

AN APPROACH TO BUILDING DOMAIN SPECIFIC SOFTWARE ARCHITECTURES
FROM SOFTWARE ARCHITECTURAL DESIGN PATTERNS

by

Julie Street Fant
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:



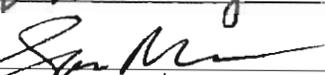
Dr. Hassan Gomaa, Dissertation Director



Dr. Alexander Brodsky, Committee Member



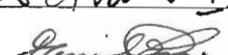
Dr. Andrew Sage, Committee Member



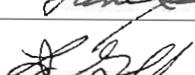
Dr. Sam Malek, Committee Member



Dr. Robert Pettit IV, Committee Member



Dr. Daniel Menascé, Senior Associate Dean



Dr. Lloyd J. Griffiths, Dean, Volgenau School of
Engineering

Date: July 28, 2011

Summer Semester 2011
George Mason University
Fairfax, VA

An Approach to Building Domain Specific Software Architectures from Software
Architectural Design Patterns

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy at George Mason University

By

Julie Street Fant
Masters of Science
George Mason University, 2004
Bachelors of Science
James Madison University, 2001

Director: Hassan Gomaa, Professor
Computer Science Department

Summer Semester 2011
George Mason University
Fairfax, VA

Copyright: 2011 Julie Street Fant
All Rights Reserved

DEDICATION

To my wonderful husband, best friend, and love of my life, John. Just like in our lives together, you have been there for me every step of the way on my PhD. Without your support, encouragement, and proof reading I would have never finished.

To my amazing son, Daniel. You bring so much joy and inspiration to my life. I hope this dissertation shows you that you can achieve anything you put your mind to.

To my mother and father. Thank you so much for instilling in me the values I need to achieve my dreams and the confidence to pursue them. This dissertation and all my other accomplishments in life are a reflection of everything you have given me.

To all my family and friends. Your words of encouragement throughout the years have touched my heart and given me the motivation to finish.

ACKNOWLEDGEMENTS

I would like to thank the following people who have helped me through the course of this dissertation:

- Dr. Hassan Gomaa your support, guidance, and patience throughout the years has enabled me to complete this PhD dissertation. Thank you so much for believing in me.
- Dr. Alexander Brodsky, Dr. Sam Malek, and Dr. Andrew Sage for serving on my committee and providing me with valuable feedback that helped shape this research.
- Dr. Robert G. Pettit IV for serving on my committee and supporting my efforts at The Aerospace Corporation. Your friendship, guidance, and help along the way were invaluable.

TABLE OF CONTENTS

	Page
LIST OF TABLES	xi
LIST OF FIGURES	xii
ABSTRACT	i
1 INTRODUCTION	1
1.1 Overview and Motivation	1
1.1 Research Problem	6
1.2 Dissertation Statement	6
1.3 Contributions	6
1.4 Organization	9
2 RELATED WORK	10
2.1 Software Architecture Design Patterns	10
2.1.1 <i>Design Pattern Composition</i>	11
2.1.2 <i>Pattern Based Development Methodologies</i>	12
2.1.3 <i>Automating Design Pattern Application</i>	13
2.1.4 <i>Other Design Pattern Processes</i>	13
2.1.4.1 Facilitating Design Pattern Application	13
2.1.4.2 Testing	14
2.1.4.3 Design Pattern Verification	14
2.1.4.4 Software Maintenance	15
2.1.4.5 Design Pattern Evolution	16
2.1.4.6 Model-Driven Architecture (MDA)	17
2.1.5 <i>Domain Specific Application of Design Patterns</i>	17
2.1.5.1 Flight Software	17
2.1.5.2 Real-time and Embedded Systems	18
2.2 Software Product Line Approaches	20
2.3 Multi-view Modeling	21
2.4 Software Architectures	22
2.5 Summary	23
3 Overview of research	25
3.1 Research Approach	25
3.2 Advantages of Research Approach	28
3.3 Validation	30

4	DISTRIBUTED, REAL-TIME AND EMBEDDED ARCHITECTURAL AND EXECUTABLE DESIGN PATTERNS	32
4.1	Introduction	32
4.2	Architectural and Executable Design Pattern Overview	32
4.2.1	<i>Architectural Design Pattern Views</i>	33
4.2.2	<i>Executable Design Patterns</i>	37
4.3	Centralized Control Architectural Design Pattern.....	40
4.4	Centralized Control Executable Design Pattern	45
4.5	DRE Design Pattern Listing	50
5	Building Domain Specific Software Product Line Architectures from Design Patterns	52
5.1	Introduction	52
5.2	Feature Modeling for Design Patterns.....	52
5.3	Feature to Design Pattern Mapping.....	54
5.4	Levels of Executable Design Pattern Modeling	58
5.4.1	<i>Software Product Line Architectural Design Patterns</i>	59
5.4.2	<i>Software Produce Line Executable Design Patterns</i>	68
5.4.3	<i>Application Architectural Design Patterns</i>	70
5.4.4	<i>Application Executable Design Patterns</i>	78
5.5	Design Pattern Interconnection	80
5.5.1	<i>Software Product Line Level Design Pattern Interconnection</i>	80
5.5.2	<i>Application Level Design Pattern Interconnection</i>	90
6	An Approach to Building Domain Specific Software Product Line Architectures from Design Patterns	91
6.1	Introduction	91
6.2	Approach to Build Domain Specific Software Architectures from Design Patterns Overview.....	92
6.3	Unmanned Spacecraft Flight Software Domain Overview	94
6.4	Unmanned Spacecraft Flight Software Product Line Problem Description.....	96
6.5	Software Product Line Engineering	98
6.6	Software Product Line Requirements Modeling Phase.....	98
6.6.1	<i>Software Product Line Feature Modeling</i>	98
6.6.2	<i>Software Product Line Use Case Activity Modeling</i>	117
6.7	Software Product Line Analysis Modeling Phase	120
6.7.1	<i>Software Product Line Feature to Design Pattern Mapping</i>	121
6.7.1.1	Flight Software Product Line Dynamic Model for Low Volume Command Execution.....	122
6.7.1.2	Flight Software Product Line Dynamic Model for High Volume Command Execution.....	124
6.7.1.3	Flight Software Product Line Dynamic Model for Time-Triggered Command Execution.....	126

6.7.1.4	Flight Software Product Line Command and Data Handling Subsystem Feature to Design Pattern Mapping Summary	127
6.7.2	<i>Software Product Line Architectural Design Pattern Modeling</i>	129
6.7.3	<i>Software Product Line Executable Design Pattern Modeling</i>	136
6.7.4	<i>Software Product Line Design Pattern Interconnection</i>	139
6.8	Software Product Line Design Modeling	143
6.8.1	<i>Software Product Line Architecture</i>	143
6.8.2	<i>Software Product Line Message Communication</i>	146
6.9	Comparison of Approaches	148
7	Application Engineering	156
7.1	Introduction	156
7.2	Problem Description	156
7.3	Application Engineering	162
7.4	Application Requirements Modeling	162
7.4.1	<i>Application Feature Modeling</i>	162
7.4.2	<i>Application Use Case Modeling</i>	170
7.4.3	<i>Application Use Case Activity Modeling</i>	171
7.5	Application Analysis Modeling	173
7.5.1	<i>Application Conceptual Static Modeling</i>	173
7.5.2	<i>Application Context Modeling</i>	175
7.5.3	<i>Application Subsystem Structuring</i>	178
7.5.4	<i>Application Design Pattern Selection</i>	181
7.5.4.1	<i>SNOE Centralized Control Architectural Design Pattern</i>	182
7.5.4.2	<i>SNOE Centralized Control Executable Design Pattern</i>	186
7.5.5	<i>Application Design Pattern Interconnection Modeling</i>	187
7.5.6	<i>SNOE Design Pattern within Layered Architecture</i>	190
8	Validation	191
8.1	DRE Executable Design Pattern Validation	191
8.2	FSW SPL Executable Design Pattern Validation	195
8.3	Application Executable Design Pattern Validation	199
8.4	FSW SPL Members' Software Architectures Validation	200
8.4.1	<i>FSW SPL Validation Activities</i>	201
8.4.1.1	<i>FSW SPL Decision Tables and Test Specifications</i>	201
8.4.1.2	<i>FSW SPL Feature and Design Pattern Test Coverage</i>	203
8.4.2	<i>SNOE Functional Validation</i>	206
8.4.2.1	<i>SNOE Test Specifications</i>	206
8.4.2.2	<i>SNOE Test Data</i>	212
8.4.2.3	<i>Test application</i>	216
8.4.3	<i>STEREO Functional Validation</i>	220
8.5	Validation Summary	225
9	Conclusions and Future Work	227

9.1	Contributions	227
a)	<i>Executable design patterns</i>	228
b)	<i>Software product lines with design pattern level variability</i>	228
c)	<i>Three levels of executable design pattern customization</i>	229
d)	<i>Design pattern integration modeling</i>	230
e)	<i>Feature and design pattern based validation</i>	231
9.2	Future Work.....	231
9.2.1	<i>Application to other subsystems in FSW SPL</i>	232
9.2.2	<i>Application to other industrial domains</i>	232
9.2.3	<i>Automation</i>	232
9.2.4	<i>Extending approach to implementation phase</i>	233
9.2.5	<i>Design pattern based performance analysis</i>	233
APPENDIX A. Distributed Real-Time and Embedded Architectural and Executable Design Patterns 234		
A.1	Hierarchical Control.....	234
A.2	Distributed Control.....	236
A.3	Command Dispatcher.....	237
A.4	Pipes and Filters	239
A.5	Master Slave.....	241
A.6	Strategy.....	242
A.7	Two Phase Commit	244
A.8	Compound Commit	245
A.9	Client Server.....	247
A.10	Publish Subscribe	248
A.11	Broadcast.....	250
A.12	Multicast.....	251
A.13	Protected Single Channel	252
A.14	Homogeneous Redundancy.....	253
A.15	Triple Modular Redundancy	255
A.16	Heterogeneous Redundancy.....	257
A.17	Monitor-Actuator	259
A.18	Sanity Check	261
A.19	Watchdog	263
A.20	Layers.....	264
A.21	Five-Layer Architecture	265
A.22	Asynchronous Message Communication.....	265
A.23	Bidirectional Asynchronous.....	266
A.24	Asynchronous Communication with Callback.....	268
A.25	Synchronous with Reply	269
A.26	Synchronous without Reply	270
A.27	Brokered Communication	271
APPENDIX B. Unmanned Spaceflight Software Product Line		273
B.1	Unmanned Spacecraft Flight Software Product Line Use Case Model	273
B.1.1	<i>Small Spacecraft System Problem Description</i>	273

B.1.2	<i>Large Spacecraft System Problem Description</i>	274
B.1.3	<i>Time-Triggered Spacecraft System Problem Description</i>	275
B.1.4	<i>Small Spacecraft System Use Case Model</i>	276
B.1.5	<i>Large Spacecraft System Use Case Model</i>	277
B.1.6	<i>Time-Triggered Spacecraft System Use Case Model</i>	280
B.1.7	<i>FSW SPL Use Case Model</i>	281
B.2	Collect and Store Spacecraft Data Use Case Activity Diagram Model	290
B.3	Unmanned Space Flight Software Product Line Conceptual Static Model ...	296
B.4	Unmanned Space Flight Software Product Line Context Model	299
B.5	Unmanned Space Flight Software Product Line Subsystem Structuring	300
B.6	Unmanned Spacecraft Flight Software Product Line Feature to Design Pattern Mapping	303
B.6.1	<i>Dynamic Model for Low Volume Command Execution with Flexible Commands</i>	303
B.6.2	<i>Dynamic Model for High Volume Command Execution with Flexible Commands</i>	306
B.6.3	<i>Dynamic Model for Time-Triggered Volume Command Execution with Flexible Commands</i>	307
B.6.4	<i>Dynamic Model for Spacecraft Clock</i>	308
B.7	Unmanned Spacecraft Flight Software Product Line Architectural Design Pattern Modeling	309
B.7.1	<i>FSW Distributed Control Architectural Design Pattern</i>	309
B.7.2	<i>FSW Hierarchical Control Executable Design Pattern</i>	311
B.8	Collect and Store Spacecraft Data Design Pattern Interconnection	313
B.9	Unmanned Spacecraft Flight Software Product Line Message Communication	316
APPENDIX C.	SNOE Case Study	318
C.1	SNOE Collect and Store Spacecraft Use Case Activity Modeling	318
C.2	SNOE Architectural and Executable Design Patterns	321
C.2.1	<i>SNOE Telemetry Client Server Executable Design Pattern</i>	322
C.2.2	<i>SNOE Payload Multiple Client Multiple Server Executable Architectural Design Pattern</i>	322
C.2.3	<i>SNOE Payload Multiple Client Multiple Server Executable Design Pattern</i>	327
C.2.4	<i>SNOE Memory Storage Device Watchdog Executable Design Pattern</i>	328
C.3	SNOE Collect and Store Spacecraft Data Design Pattern Interconnection ...	328
APPENDIX D.	Solar TERrestrial RELations Observatory (STEREO) Case Study	332
D.1	Problem Description	333
D.2	STEREO Flight Software Requirements Modeling	339
D.2.1	<i>STEREO Feature Model</i>	340
D.2.2	<i>STEREO Use Case Model</i>	347
D.2.3	<i>STEREO Use Case Activity Model</i>	348
D.2.4	<i>STEREO Conceptual Static Model</i>	350
D.2.5	<i>STEREO Context Model</i>	352

D.2.6	<i>STEREO Subsystem Structuring</i>	356
D.3	STEREO Analysis Modeling	359
D.3.1	<i>STEREO Executable Design Pattern Selection</i>	359
D.3.2	<i>STEREO Hierarchical Control Architectural Design Pattern</i>	360
D.3.3	<i>STEREO Design Pattern Interconnection</i>	361
D.3.4	<i>STEREO Design Pattern within Layered Architecture</i>	364
REFERENCES	367

LIST OF TABLES

Table	Page
Table 4-1 DRE design patterns	51
Table 5-1 Sample Feature to Design Pattern Mapping	58
Table 6-1 Feature/Use case dependencies of FSW SPL C&DH subsystem.....	114
Table 6-2 Feature conditions for FSW SPL C&DH subsystem	118
Table 6-3 Command and Data Handling Feature to Design Pattern Mapping	128
Table 6-4 Comparison pattern based SPLE with versus component/connector based SPLE	148
Table 6-5 FSW SPL Variable Component Comparison for the Command Execution use case.....	153
Table 7-1 SNOE Design Pattern Selection	181
Table 8-1 Summary of DRE executable design patterns validated	192
Table 8-2 Test case for the Client Server design pattern	192
Table 8-3 Summary of FSW SPL executable design patterns validated	195
Table 8-4 Test Case for the Houskeeping_DCClient Housekeeping_DServer design pattern	197
Table 8-5 Summary of SNOE executable design patterns validated	199
Table 8-6 Summary of STEREO executable design patterns validated	199
Table 8-7 Execute Commands Decision Table.....	202
Table 8-8 Execute Commands test specification / feature / design pattern table	204
Table 8-9 Subset of SNOE's Execute Commands Decision Table	209
Table 8-10 SNOE's LGA Reinitialize Test Specification	214
Table 8-11 Summary of SNOE's Test Specifications	217
Table 8-12 Subset of STEREO's Execute Commands Decision Table.....	220
Table 8-13 STEREO's HGA Reinitialize Test Specification	221
Table 8-14 Summary of STEREO's Test Specifications	223

LIST OF FIGURES

Figure	Page
Figure 3-1 High level view of proposed process	26
Figure 4-1 Example collaboration diagram	34
Figure 4-2 Example sequence diagram.....	35
Figure 4-3 Example component diagram.....	36
Figure 4-4 Example port design.....	37
Figure 4-5 Example state machine diagram.....	40
Figure 4-6 Collaboration diagram for Centralized Control design pattern.....	41
Figure 4-7 Representative sequence diagram for Centralized Control design pattern	42
Figure 4-8 Component diagram for Centralized Control design pattern	43
Figure 4-9 Port design for Centralized Control design pattern.....	44
Figure 4-10 State machine for Input_Component	46
Figure 4-11 State machine for Centralized Controller.....	47
Figure 4-12 State machine for Output_Component.....	48
Figure 4-13 State machine for IO_Component.....	50
Figure 5-1 Subset of FSW SPL feature model.....	54
Figure 5-2 Subset interaction diagram for the Low Volume Command Execution pattern specific feature	56
Figure 5-3 Example FSW level Centralized Control collaboration diagram.....	61
Figure 5-4 Representative interaction diagram for FSW Centralized Control	64
Figure 5-5 FSW Centralized Control execute design pattern component diagram	66
Figure 5-6 Port design for the FSW SPL Centralized Control executable design pattern	67
Figure 5-7 CDH Centralized Controller’s state machine.....	70
Figure 5-8 FSW application specific Pipes and Filters collaboration diagram	73
Figure 5-9 SNOE application specific sequence diagram for executing command to adjust attitude.....	75
Figure 5-10 SNOE Centralized Control component diagram.....	77
Figure 5-11 Subset port design for SNOE’s Centralized Control design pattern	78
Figure 5-12 Simplified Execute Commands use case activity model.....	83
Figure 5-13 Execute Commands interaction overview diagram.....	87
Figure 5-14 Subset FSW Centralized Control and FSW Spacecraft Clock Multicast interconnection.....	89
Figure 6-1 High Level View of Approach.....	94
Figure 6-2 High level FSW SPL C&DH feature model with dependencies.....	102
Figure 6-3 C&DH Feature Model.....	107
Figure 6-4 C&DH feature dependency model continued	112

Figure 6-5 Execute Commands activity diagram.....	120
Figure 6-6 Low Volume Command Execution communication diagram.....	122
Figure 6-7 High Volume Command Execution communication diagram.....	125
Figure 6-8 Time Triggered Command Execution communication diagram.....	127
Figure 6-9 FSW level Centralized Control collaboration diagram.....	130
Figure 6-10 FSW Centralized Control execute commands interaction diagram.....	133
Figure 6-11 Component diagram for FSW Centralized Control.....	134
Figure 6-12 Port design for FSW SPL Centralized Control design pattern.....	135
Figure 6-13 FSW CDH_Centralized_Controller state machine.....	137
Figure 6-14 FSW Heater_OC state machine.....	138
Figure 6-15 Interaction overview for Execute Commands use case.....	141
Figure 6-16 Interconnection between CDH_Centralized_Controller and Spacecraft_Clock_Multicast.....	143
Figure 6-17 Layered Architecture for FSW SPL.....	145
Figure 7-1 SNOE Spacecraft Structure (Laboratory For Atmospheric and Space Physics at the University of Colorado at Boulder n.d.).....	158
Figure 7-2 SNOE C&DH pattern specific feature selection.....	164
Figure 7-3 SNOE pattern variability feature selection.....	168
Figure 7-4 SNOE use case model.....	171
Figure 7-5 SNOE Execute Commands use case activity model.....	172
Figure 7-6 SNOE FSW conceptual static model.....	174
Figure 7-7 SNOE FSW system context diagram.....	176
Figure 7-8 SNOE FSW context diagram.....	177
Figure 7-9 SNOE FSW subsystem structuring.....	178
Figure 7-10 Subsystem allocation to physical devices.....	180
Figure 7-11 SNOE Centralized Control communication diagram.....	183
Figure 7-12 Execute Commands Scenario for SNOE.....	184
Figure 7-13 Component diagram for SNOE Centralized Control executable design pattern	185
Figure 7-14 SNOE's command execution interaction overview diagram.....	189
Figure 7-15 SNOE Architecture Layered View with Centralized Control Design Pattern Components.....	190
Figure 8-1 State machine for Client.....	194
Figure 8-2 State machine for Server.....	194
Figure 8-3 State machine for FSW SPL Housekeeping_DClient.....	198
Figure 8-4 State machine for FSW SPL Housekeeping_DServer.....	198

ABSTRACT

AN APPROACH TO BUILDING DOMAIN SPECIFIC SOFTWARE ARCHITECTURES USING SOFTWARE ARCHITECTURAL DESIGN PATTERNS

Julie S. Fant, PhD

George Mason University, 2011

Dissertation Director: Dr. Hassan Gomaa

Software architectural design patterns represent best practice solutions to common design challenges. However, applying design patterns in practice can be difficult because they are typically documented to be domain independent. This makes applying them in a particular domain difficult. Knowing where and at what level of abstraction software architectural design patterns should be applied in a given domain is not always clear. Currently, there are no existing approaches for building and validating domain specific software architectures that focus on reusing and composing existing software architectural design patterns. This dissertation addresses this gap by developing a software product line (SPL) based approach to building and validating domain specific software architectures from software architectural design patterns.

The key contributions of this research include: the definition of distributed real-time and embedded (DRE) executable design patterns; the definition of a SPL design approach that captures SPL variability at a higher degree of granularity using design patterns; the definition of different levels of required executable design pattern customizations; and a feature and design pattern based functional validation approach. Additionally, a domain specific SPL and two real world case studies are provided to validate and demonstrate the applicability of this approach.

1 INTRODUCTION

1.1 Overview and Motivation

Software design patterns are best practice solutions to common software problems.

Design patterns are normally captured to be domain and platform independent. There are several benefits to capturing design patterns in this manner. First, it makes them applicable across multiple domains and platforms. Second, it makes design patterns applicable at different levels of abstraction. However, there are also limitations to capturing design patterns in a domain independent fashion. First of all, it makes design patterns difficult to adopt in practice because it is not always clear at what level of abstraction in the software architecture the design pattern should be applied in a given domain. Secondly, it can be unclear as to which functionality in the software architecture can be improved using software architectural design patterns.

Furthermore, in the majority of cases, multiple design patterns can be applied in a single application. However, design pattern literature rarely provides guidance for composing multiple design patterns. When guidance is provided, such as in (Buschmann et al. 2007, Buschmann et al. 1996; Kircher & Jain 2004; D. Schmidt et al. 2000), it is typically at a

high level and only describes the relationships between patterns. This guidance does not include domain specific guidance for how specific design patterns should be composed to form a software architecture.

Furthermore design pattern guidance also does not provided a way of defining how design pattern can be systematically modified. On the other hand software product line (SPL) engineering provides a systematic means of describing variability in a software family and where variability can be introduced into an architecture. However, SPL engineering focuses on address variability at the component and connector level rather than at the design pattern level.

To address the aforementioned challenges, this research focuses on creating a systematic and repeatable approach to designing distributed, real-time and embedded (DRE) software from software architectural design patterns using software product line concepts. As part of this approach, the software architectures produced are also validated for functional correctness.

To achieve this goal, this dissertation first provides a set of design patterns that are applicable to DRE software. The architectural design patterns fully specify and capture variability in the design pattern using multiple static and dynamic architectural views. Additionally, an executable version of the design pattern is also provided that can be

systematically customized and applied to form an executable software architecture for a particular application. Their executable characteristic are also used to functionally validate software architectures produced using the proposed approach.

Second, this dissertation provides a Software Produce Line (SPL) approach to systematically apply the design patterns to a specific DRE domain. This involves building SPL architectures that utilize and methodically customize a subset of the DRE design patterns at the SPL and SPL member levels. Variability in the SPL is captured at the design pattern level, rather than the component or sub-component level. Thus, SPL variability concepts capture the domain specific integration rules that define how the various design patterns can and cannot be integrated to form SPL member software architectures. Finally, application specific architectures are derived and customized from the SPL architecture.

This research also includes a validation approach that is used to validate the functional requirements of software architectures produced following the SPL approach. Functional validation is achieved using a feature and design pattern based model based testing approach.

This research is applied and validated using the unmanned space flight software (FSW) domain. FSW is an ideal domain to apply this research for multiple reasons. First, the

amount of requirements and responsibilities placed on FSW is growing. FSW has evolved from performing simple operations to controlling a majority of the spacecraft and payloads. This research provides a systematic way to architect FSW that leverages design patterns. The systematic approach helps to ensure that all of the requirements are met. Furthermore, using design patterns makes certain that best practices are incorporated into FSW designs.

Secondly, an industry trend indicates that the number of software related anomalies is growing. It is reported that “in the period from 1998 to 2000, nearly half of all observed spacecraft anomalies were related to software” (Hecht & Buettner 2005). These software anomalies can cause mission disruption or even mission loss. In the aerospace industry these losses cannot be tolerated because of the high cost and length of time that is required to build a spacecraft. Additionally, many spacecraft support very critical missions that can be severely impacted from a small disruption of service. This research can help to alleviate the number of software related anomalies by providing design time validation through simulation. Therefore, design flaws that lead to software anomalies can be identified and remedied early.

Finally, this research aligns with industrial recommendations to help manage FSW acquisitions. In 2007, the NASA Office of Chief Engineer commissioned a multi-center study to better understand the problem and to provide recommendations which better

manage FSW complexity (NASA Office of Chief Engineer n.d.). One of NASA's recommendations is to perform early analysis and architecting of FSW (Dvorak (editor) 2009). This research can be used to satisfy this recommendation.

To apply this research to the FSW domain, a FSW SPL architecture is created that leverages and customizes the DRE architectural and executable design patterns. Then, two real world case studies from the FSW domain are derived from the FSW SPL. The two case studies vary in complexity and spacecraft structure to illustrate a wide applicability of this approach in the FSW domain. The case studies themselves will also be validated for functional correctness.

It is seen from the case studies that, using the proposed approach, FSW architectures can be systematically and repeatedly built using architectural and executable design patterns. This ensures the FSW architectural designs leverage of software architectural best practices through design patterns. Additionally, the approach ensures that FSW architectures produced will also meet their functional requirements by following the proposed validation approach.

Overall, this research provides a systematic and repeatable means to design DRE software from software architectural design patterns. When applied to the FSW domain the systematic design process along with the associated design time functional validation

should result in higher quality software architectures and subsequently reduce number of anomalies seen in flight.

1.1 Research Problem

The problem statement for this research effort is:

There are no existing systematic and repeatable approaches that leverage software product line concepts for building and validating domain specific software architectures from software architectural design patterns.

1.2 Dissertation Statement

The research statement for this effort is:

This research describes a systematic and repeatable approach that utilizes software product line concepts for composing and adapting software architectural design patterns that can be used to build and validate domain specific software architectures.

1.3 Contributions

a) Executable architectural design patterns

This PhD research addresses several research gaps. First, current work on software architectural design patterns only provides platform and domain independent specifications of the patterns. This results in a gap between the identification of the design pattern and the application of the design pattern. This is limiting during the application of design patterns because the design patterns must be re-created and customized to the particular application. This can be time consuming and there is the

potential that the design patterns are not properly applied. For example, an engineer may accidentally leave out a component required by the design pattern. To address this gap, this research provides a set of architectural and executable DRE design patterns that can be systematically applied and customized to create software architectures.

Another gap within existing design pattern literature is that it does not explicitly capture variability within the design patterns or provide guidance for how variability in design patterns can be customized for a software architecture. This additional information is needed to make applying design patterns easier within a particular. This PhD research fills this gap by modeling variability within design patterns and providing systematic steps for customizing design pattern variability for an SPL and then an SPL member.

b) Software product lines with design pattern level variability

A third research gap related to this PhD research that current work on software product line methodologies focuses on addressing variability at the component and connector level. While this approach has proven successful in many domains, it faces scalability challenges when applied in domains that have significant variability. This is because they requiring capturing all potential variability within a SPL and modeling it down to individual components and connectors. When significant architectural variability exists in an SPL this requires a large amount of components, connectors, and interactions that must be individually modeled. This can be cumbersome to develop and difficult to maintain. This research addresses the aforementioned scalability challenged by defining

an SPL methodology that captures variability at the design pattern level, as opposed to the component and connector level. Since design patterns can abstract several different individual components and connectors, less modeling is required at the SPL level. Instead some of the variability decisions are deferred to application engineering.

c) Three levels of executable design pattern customization

A third research gap addressed by this research is there are only platform and domain independent software architectural design pattern specifications to help in the application of design patterns. This means that the design patterns must be re-created every time they are applied and ad hoc customizations are made when the design patterns are applied. This research goes beyond these works by modeling three levels of architectural design patterns to progressively address variability within the patterns themselves and variability in the patterns selected for a member application. The purpose of providing three levels of executable design patterns is to facilitate reuse and customization during SPL and SPL application development. Additionally, this also saves the software engineer time when building SPL and SPL applications because they provide a good starting point, as opposed to starting completely from scratch.

d) Design pattern integration modeling

Another research gap address in this PhD research is that existing SPL approaches and multiple view modeling approaches do not provide views to illustrate how design patterns can be interconnected to achieve the functionality required of a software architecture. This PhD research addresses this gap by creating new scenario based architectural views

that capture how design patterns can be interconnected to meeting the required functionality. This view also expresses variability in the design pattern interconnections.

e) Feature and design pattern based validation approach

The last research gap this PhD address is there are currently no SPL model-based approaches for testing and validating design patterns within executable software architectures. Currently model-based SPL testing approaches focus on using modeled to develop test specifications that can be applied after the architecture is implemented. This PhD dissertation addresses the gap by extending a model-based SPL testing to test and validate executable software architectures, as opposed to code.

1.4 Organization

Chapter 1 presents the problem statement and motivation for this research. Chapter 2 presents a survey of related work. Next, Chapter 3 describes how to apply the proposed research approach. Then, Chapter 4 describes the DRE design patterns and Chapter 5 describes in detail the main contributions of this approach and overall approach in detail. Chapters 6 and 7 the approach is applied to the FSW domain. Chapter 8 describes the details of the validation. Finally, Chapter 9 provides the conclusion, the research's main contributions, and potential areas for future work.

2 RELATED WORK

This research addresses an approach to designing flight software from software architectural design patterns. The following sections of this chapter discuss a variety of works related to this dissertation as well as active research areas involving design patterns.

2.1 Software Architecture Design Patterns

Software design patterns are “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” (Gamma et al. 1995). They were first introduced by Gamma et al in 1995. Since then, researchers have continued to expand the set of design patterns. Additionally, researchers began to expand design patterns to address higher level architectural issues, called software architectural design patterns (Buschmann et al. 1996; Shaw & Garlan 1996a; D. Schmidt et al. 2000; Kircher & Jain 2004; Goma 2005).

The inherent reusability and the broad application of design patterns and architectural design pattern have attracted many researchers to their development and application. The

related works in this area can be broadly grouped into categories of design pattern composition, pattern-based development methodologies, automating design pattern application, other design pattern processes, and domain specific application. Related works in this area are also discussed.

2.1.1 Design Pattern Composition

An area of design pattern research that is closely related to this research focuses on composing design patterns together. Ram et al (Ram et al. 2004) propose an approach to handle design pattern composition for when one pattern uses another pattern or when one pattern combines with another pattern for completeness. This is accomplished by capturing design patterns as tuples consisting of the element(s) to which the design pattern applies, design pattern purpose, and design pattern unique intent. The next step is to define a set of rules for combining tuples based on the uses and combines relationship. Finally, it is required to provide guidance on how the uses and combines relationships can be statically modeled.

Reihle's research (Riehle 1997) describes some specific examples of composite patterns and describes a composition technique. The composition technique uses role diagrams to capture the design patterns and composition constraints captured in a relationship matrix, which relates every role with every other role. The matrix is then analyzed and used to determine the pattern composition.

Yacoub et al (Yacoub, Xue, et al. 2000; Yacoub, Xue,, et al. 2000) developed a tool for composing design patterns together. It first involves a pattern view, where an engineer can select and string together various design patterns. If a dependency is drawn between patterns, then a pattern interface view is required. The pattern interface view is where the dependency is refined by explicitly defining an interface between the patterns. Finally, for each pattern selected in the pattern view, a detailed design view is also created. This view captures the internals of the design pattern and the application specific information.

Finally, Bayley and Zhu (Bayley & Zhu 2008, Bayley & Zhu 2007) and Singh and Chaudhry (Singh & Chaudhary 2009b, Singh & Chaudhary 2009a) use formal methods in their approach to representing and integrating design patterns.

This research is different from the aforementioned works because it provides rules for software architectural design pattern composition using SPL concepts. This research also goes a step further and provides fully specified executable design patterns that serve as the foundation for software architectures.

2.1.2 Pattern Based Development Methodologies

The pattern based software development methodologies place emphasis on using design patterns to compose software. These works typically provide several design patterns and provide high level guidance on how to apply design patterns. Notable works in this area (Buschmann et al. 1996; D. Schmidt et al. 2000; Kircher & Jain 2004; Buschmann et al.

2007; Gomaa 2005). This PhD research is different because it provides detailed, domain specific guidance for applying design patterns use SPL concepts by extending (Gomaa 2005). Also, in addition to providing architectural design pattern specifications, this research also provided executable design patterns.

2.1.3 Automating Design Pattern Application

Historically, design patterns have mostly been captured in textual formation. However, making the leap from text to implementation can be difficult for practitioners. In recent years, researchers started focusing on automating the application of design patterns to make them easier to apply. Some works in this area include (Dascalu et al. 2005; El Boussaidi & Mili 2007; Jalil et al. 2010; Bulka 2002).

2.1.4 Other Design Pattern Processes

2.1.4.1 Facilitating Design Pattern Application

Another area of design pattern process research focuses on facilitating the use of design patterns by making them easier to use, find, and communicate. First, some research focuses on tracing design patterns to code (Landauer 2000; Baniassad et al. 2003; Dong et al. 2005, Dong et al. 2006). The goal of this research is to communicate when design patterns are used in the design and in the code.

Third, some researchers are developing approaches to help in the selection of design patterns (Noble 1998; Schulz et al. 1998; J. Wang, Song & Chung 2005d; Briand et al.

2006; Meffert 2006). The goal of this research is to facilitate the use of design pattern by aiding in their selection.

Other research is developing new ways to teach design patterns (Ouyang 2002; Al-Tahat et al. 2006; Alphonse et al. 2007; Gestwicki 2007; Wright 2007). The goal of this research is to facilitate the use of design patterns by teaching engineers about design patterns, their importance, and how to incorporate them into designs.

2.1.4.2 Testing

Another active area of design pattern process research is on testing. Researchers are examining the impact design patterns have on testing strategies while others are developing new testing strategies for design patterns (Tsai et al. 1999; Yam-Gael Gueheneuc & Albin-Amioti 2001; Dasiewicz 2005). The ultimate goal of this research is to help testers develop testing strategies that are appropriate for software with design patterns.

2.1.4.3 Design Pattern Verification

Another area that researchers are exploring in design pattern processes is the verification of design patterns. Researchers are developing tools and techniques to help verify that a design pattern is implemented correctly, and in some cases, to assist with the documentation (Cornils & Hedin 2000; Yuan et al. 2004; Dong et al. 2007; Hsueh et al. 2007). Alencar et al (Alencar et al. 1999) take it a step further by identifying undesired

interactions among the design pattern components when multiple design patterns are applied.

2.1.4.4 Software Maintenance

A fifth active area in design pattern process research is in software maintenance. First, several researchers are conducting empirical studies to examine if the explicit documentation of design patterns in software is beneficial to software maintainers (Prechelt et al. 2001, Prechelt et al. 2002; Izurieta & J. Bieman 2007; Ng et al. 2007; J. M. Bieman et al. 2003). The majority of the studies show a positive relationship between design patterns and the ability to maintain systems. Another empirical study examines the evolution of design patterns in terms of how frequently design patterns change as well as identification of which classes are impacted by the change (Aversano, Canfora, et al. 2007). This study found that design patterns closely tied to the application changed more frequently and the classes impacted by the change are closely tied to the type of change. Finally, another case study explores the value of design patterns in reengineering software (Masuda et al. 2000). It shows that the use of design patterns can improve the extensibility of a reengineered piece of software.

Secondly, other research focuses on developing reverse engineering approaches to extract design patterns from existing code (Tonella & Antoniol 1999; Albin-Amiot et al. 2001; Balanyi & Ferenc 2003; Arcelli, Masiero & Raibulet 2005; Arcelli, Masiero, Raibulet, et

al. 2005; Basu et al. 2005; Costagliola et al. 2005; Ferenc et al. 2005; W. Wang & Tzerpos 2005; Kaczor et al. 2006; Kim & Lu 2006; Pappalardo & Tramontana 2006; Shi & Olsson 2006; Dong et al. 2007; H. Lee et al. 2007; Dong & Y. Zhao 2007). The goal of this research is to extract design patterns from existing code to help software maintainers understand the design and to help them make appropriate changes.

Finally, another active area of research is in refactoring and reengineering. Researchers are developing approaches to systematically and/or automatically refactor legacy code to use design patterns (Cinneide & Nixon 2001; Jeon et al. 2002; Tahvildari & Kontogiannis 2002a, Tahvildari & Kontogiannis 2002b). The final goal of this research is to improve the maintainability of legacy code by leveraging the power of design patterns.

2.1.4.5 Design Pattern Evolution

Design evolution is another active area in design pattern process research. The goal of design evolution research is to develop techniques that make changing software easier. With respect to design patterns, researchers are developing approaches to automate changes to design patterns and frameworks (Mens & Tourwe 2001; C. Zhao et al. 2007; Dong et al. 2006). Researchers are also studying the causes and impacts of design pattern evolution (Aversano, Cerulo, et al. 2007).

2.1.4.6 Model-Driven Architecture (MDA)

Research in design pattern processes is also occurring within the Model-Driven Architecture (MDA) paradigm. Active research is being performed on building code generators and transformation techniques that automatically apply design patterns during the model-to-code transformations (Ohtsuki et al. 1999; Xue-Bin et al. 2007).

2.1.5 Domain Specific Application of Design Patterns

2.1.5.1 Flight Software

Currently, there are very few works that discuss building flight software from software architectural design patterns. Herrmann & Schöning (Herrmann & Schöning 2000) discuss how the abstract factory pattern and the facade pattern can be applied to telemetry processing. They illustrate their approach using a small satellite called, CHAMP that uses the Packet Telemetry Recommendation of the European Space Agency (ESA). Katwijk et al (Katwijk et al. 2001) developed a general architecture of a distributed real-time system using the Layers Pattern. They then show how it can be applied to satellite systems.

Another similar work that applies design patterns to flight software is NASA Goddard Space Flight Center's work on the Core Flight Software System (CFS) (Wilmot 2005, Wilmot 2006; Goddard Space Flight Center 2009). The CFS concept consists of "three major components, a small runtime flight executive, an expandable catalog/library of

reusable software components, and an integrated development environment” (Wilmot 2006). When CFS is complete, flight software engineers will be able to design and build flight software on top of the small runtime flight executive called Core Flight Executive (cFE) using components from the reuse catalog. The proposed CFS architecture leverages some design patterns including the Layers and the Publish and Subscribe patterns. Currently only the cFE is available (Goddard Space Flight Center 2010).

Finally, Bennett et al (Bennett et al. 2005, Bennett et al. 2008) are working a related research called the Mission Data System (MDS) project. The MDS provides a system level control architecture, framework, and systems engineering methodology for developing state-based models for planning and execution. The PhD dissertation can be used to complement and support this work.

2.1.5.2 Real-time and Embedded Systems

There is more research of design pattern application to real-time and embedded systems than space flight software. First, some researchers focus on how to apply existing design patterns in the real-time and embedded domain. Selic (Selic 2004, Selic 1996a, Selic 1996b) describes a set of micro-patterns can be combined in different ways to form software architectures for real-time systems. Benowitz and Niessner (Benowitz & Niessner 2003) present design patterns that can be used with the Real-time Specification for Java (RTSJ). Zalewski (Zalewski 1999, Zalewski 2002, Zalewski 2001) proposes an

approach to designing real-time software using design patterns by treating real-time systems as control systems. Bellebia and Douin (Bellebia & Douin 2006) describe how existing design patterns can be used to build middleware for embedded systems. Fliege et al (Fliege et al. 2005) propose an approach to apply design patterns to real-time systems that captures both the design patterns and architecture in Specification and Description Language (SDL).

Other researchers are focusing on developing new software design patterns specifically for the real-time and embedded system domain. Selic (Selic 1996a) describes the recursive control pattern that “provides a means for dealing with what are traditionally considered ancillary software functions” (Selic 1996a). Pont and Banner (Pont & Banner 2004) are developing design patterns for time-triggered architectures. Esmailzadeh et al (Esmailzadeh et al. 2007) present a design pattern they developed specifically for a neural network on SoC-based embedded systems. Kalinsky (Kalinsky 2002) developed software design patterns for high availability real-time software. Gay et al (Gay et al. 2007) have developed design patterns for TinyOS, an operating system for wireless network embedded systems. Dupire and Fernandez (Dupire & Fernandez 2001) developed a modified version of the command pattern called the command dispatcher pattern which is suitable for real-time systems.

2.2 Software Product Line Approaches

Software product lines (SPLs) are “a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” (Clements & Northrop 2002). The potential productivity gains resulting from a SPL’s reusable assets have attracted many researchers to their development and application. There are several notable SPL design approaches including (Clements & Northrop 2002; Pohl et al. 2005; Gomaa 2005). These approaches have proven successful in many applications. In fact, the Software Engineering Institute provides a listing of successful SPL case studies that utilize their approach (Anon n.d.). The key to SPL success is in their ability to develop assets for a family of software systems that addresses variability. This involves using commonality/variability analysis to identify the features, which are requirements. Some features will be required by all SPL members. Other features are optional or alternatives meaning that they are only required by some SPL members. After the features are determined the next step is to develop a SPL software architecture that addresses these features. The SPL software architecture must also reflect the variability in the SPL’s common and variable features. Variability in the software architecture is typically managed at the subsystem, component, and connector level. Common subsystems, components, and connectors are required by all SPL members. Optional subsystems, components, and connectors are only required by some SPL members. Finally, variant subsystems, components, and connectors are different versions

that are required by different SPL members. Finally, SPL approaches also involve application engineering where individual SPL members are derived from the SPL assets.

This research leverages existing SPL research by utilizing SPL concepts found in (Gomaa 2005). It extends SPL approaches by managing variability at the software architectural design pattern level rather than at the component and connector level. This approach to SPLs effectively reduces variability through the incorporation of architectural design patterns in SPL architectures and giving the application developer the capability of customizing the patterns to the needs of the application. This is because design patterns contain customizable components, connectors, and interactions rather than specific components, connectors, and interactions. Therefore several different combinations of specific components, connectors, and interactions are abstracted into one design pattern and do not need to be individually modeled. Thus less modeling is required at the SPL level since features have dependencies on design patterns rather than multiple individual components and connectors. The tradeoff is that the application engineering process does require additional modeling since the application specific components, connectors, and interactions must be derived from the design patterns. However, guidance is provided to help assist the application engineer and ensure the SPL architecture is maintained.

2.3 Multi-view Modeling

Another major area of related research involves multi-view modeling. This type of modeling focuses on expressing software architectures using multiple views to fully

specify the architecture from different perspectives. Typical views include requirements, static, dynamic, and deployment. First, requirement views capture the requirements of the system. Next, static views capture structural properties of the software architecture. Dynamic views capture object interactions within the software architecture. Finally, deployment views document how the software is deployed on a hardware configuration. Notable works in this area include (Nuseibeh et al. 1993; Gomaa & Shin 2004; O’Hara-Schettino & Gomaa 1998).

This research leverages existing research in multi-view modeling research by utilizing architectural views found in (Gomaa 2005). It extends this work by developing new views in UML that are necessary to express software architectures that are composed of design patterns.

2.4 Software Architectures

Finally the last area of related works is on software architectures. Research in this area focuses on the development and documentation of software architectures. This research focuses on techniques to determine and document the software components and connectors that compose software architectures. This includes performing requirements, static, and dynamic analysis on the proposed software system to determine the required components and connectors. Prominent works in this area include (Shaw & Garlan 1996b; Taylor et al. 2009; Clements et al. 2002; Gomaa 2000; Gomaa 2011; R. Allen & Garlan 1997; R. J. Allen 1997; Garlan et al. 1997; Magee et al. 1995; ISO/IEEE 2011;

Bass et al. 2003). In addition to showing how the software architectures meet their functional requirements, researchers are also developing ways to analyze quality attributes, such as software performance, in software architectures to determine if they meet their quality attribute requirements early in the software lifecycle. Examples of this type of research include (Gomaa 2005; Smith & Williams 2002; Pettit IV & Gomaa 2007; Graf 2004; Hakansson et al. 2004; D. C. Petriu 2005; Woodside et al. 2005; Harel 1997; Bass et al. 2003).

This research leverages existing software architecture research by utilizing architectural analysis techniques found in (Gomaa 2000; Gomaa 2011). It extends this work by developing guidance for successfully customizing components and connectors in DRE software architectural design patterns for the SPL software architecture and then SPL member software architectures.

2.5 Summary

This chapter summarizes the works related to this PhD research. These works include various areas of software design pattern research, software product line (SPL) approaches, multi-view modeling, and software architecture development approaches. This PhD research extends these related works by developing an approach to building domain specific software architectures from software architectural design patterns. This research utilizes existing SPL concepts to capture how the software architectural design patterns can and cannot be incorporated together to form software architectures. This

research leverages existing software architectural design patterns and extends them by creating executable versions that can be systematically applied to the SPL and then SPL member software architectures. Additionally, this research follows practices for multi-view modeling. This area of related works is extended in this PhD research since additional architectural views were needed to express the design pattern interconnections within software architectures. Furthermore this PhD research leverages existing software architecture development approaches. However, this PhD research extends this work by developing guidance for customizing components and connectors in the executable software architectural design patterns for the SPL and then SPL member software architectures.

3 OVERVIEW OF RESEARCH

3.1 Research Approach

This research presents a systematic and repeatable approach for modeling domain specific distributed, real-time, and embedded (DRE) software from software architectural design patterns. The proposed approach is a Software Product Line (SPL) based approach where SPL variability is captured at a higher degree of granularity using software architectural design patterns. This enables SPL variability concepts to capture the integration rules for how design patterns can and cannot be composed. Additionally, it provides a systematic approach to customizing domain independent design patterns at the SPL and application levels.

A high level view of the proposed SPL approach is depicted in Figure 3-1, where processes are rectangles and repositories are cylinders. The approach taken for this research first involves pre-selecting software architectural design patterns that are applicable to DRE software. Next, it involves building DRE level architectural design patterns and executable design patterns. The architectural design patterns fully specify

and capture variability within the design patterns using multiple static and dynamic architectural views. The executable design patterns are executable version of the design pattern, using state machines, that is intended to facilitate functional validation. The DRE level architectural and executable design patterns are created only once and are reused and modified by all domain specific DRE SPLs, as shown using an DRE Design Architectural and Executable Design Patterns repository in Figure 3-1.

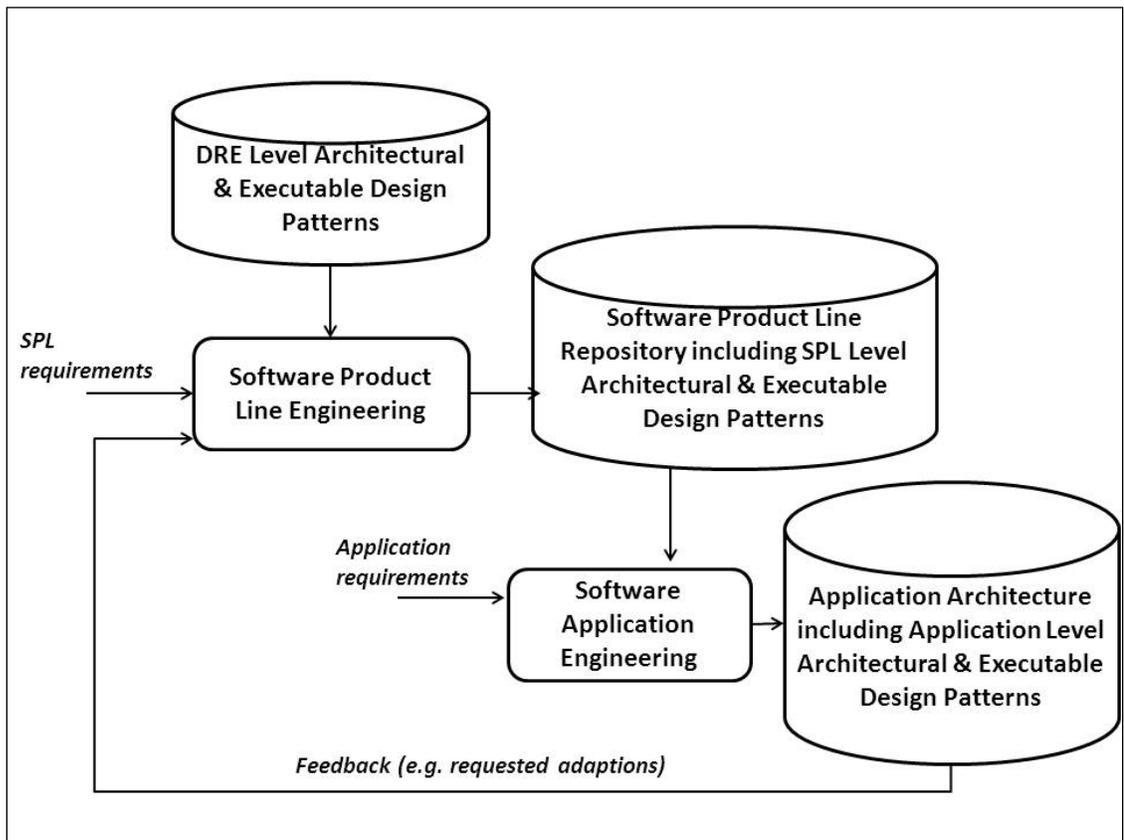


Figure 3-1 High level view of proposed process

Then, this research is applied to a specific DRE domain, as shown in the Software Product Line Engineering process in Figure 3-1. This involves building a domain specific SPL architecture that utilizes and customizes a subset of the DRE architectural and executable design patterns. In the SPL, the features are mapped to design patterns rather than components. Design patterns contain customizable components, connectors, and interactions rather than specific components, connectors, and interactions. Therefore several different combinations of specific components, connectors, and interactions are abstracted into one design pattern and do not need to be individually modeled. Thus variability is captured with a high degree of granularity and less modeling is required at the SPL level since features have dependencies on design patterns rather than multiple individual components and connectors. The tradeoff is that the application engineering process does require additional modeling since the application specific components, connectors, and interactions must be derived from the design patterns. However, guidance is provided to help assist the application engineer and ensure the SPL architecture is maintained. The SPL also captures the integration rules that define how the various design patterns can and cannot be integrated to form an overall software architecture.

During the SPL engineering phase, the DRE architectural and executable design patterns are customized with domain specific knowledge to create SPL level architectural and executable design patterns. The architectural design patterns are updated to reflect the

SPL specific components and variability. The executable design patterns are also updated to reflect the SPL specific components.

The last major process shown in Figure 3-1, is the Application Engineering Process where this research is applied to create a specific application. The application's architecture is derived from the SPL architecture based on the application's requirements. The specific SPL architectural and executable design patterns selected for the application are customized to form the foundation for the application's architecture. Finally, the application specific adaptations to the architectural design patterns include updating the architectural specification to show the variant and optional components used by the application. The executable design patterns are also updated to reflect the variant and optional components and interactions.

3.2 Advantages of Research Approach

The approach described in this research has a number of distinct advantages. First, and for most, a benefit of this research is capturing SPL variability at a higher degree of granularity than traditional SPLs. This is accomplished by capturing variability at the architectural design pattern level, rather than the component and sub-component level used in typical SPL approaches. Variability captured at a higher degree of granularity increases architectural flexibility and reduces the amount of modeling required in the SPL. Thus this approach provides a scalable way to manage variability in domains with significant architectural variability.

Second, the method for capturing architectural design patterns, executable design patterns, and constructing software architectures from these design patterns does not deviate from the existing UML 2.0 industrial standard (Rumbaugh et al. 2004; OMG 2005). The use of standards enables software engineers to quickly understand and apply this research. Furthermore, this research also uses existing commercial UML Computer Aided Software Engineering (CASE) tools. This enables this research to be easily applied to industrial applications.

Another benefit of this research is that the design patterns used for the SPL and applications are derived from design patterns for DRE software. The research contains a set of DRE architectural and executable design patterns that can be customized to any DRE domain. This ultimately helps save SPL engineers time when developing their SPL architectures because the DRE foundation for the design patterns is provided. This increases the applicability and practicality of this research.

Additional benefits of this research are that the design patterns used are pre-selected to address a wide variety of DRE systems. Again, this saves software engineers time because they do not need to sift through design pattern literature to find the appropriate design patterns.

Other benefits of this research are that the executable design patterns are systematically applied to form the foundation of the domain specific SPL and application specific architectures. The executable nature of the design patterns facilitates the functional validation of the architecture.

Finally, this research supports a real world problem and it aligns with an industrial recommendation for improving space flight software acquisition. This research has the potential to help many FSW programs improve the quality of their software architectures and subsequently reduce the number of FSW anomalies.

3.3 Validation

The proposed approach is validated by ensuring functional correctness of the individual DRE, SPL, and application executable design patterns as well as the executable application software architectures. The individual DRE executable design patterns are validated using executable object modeling with statecharts (Harel 1997). The flow of events and messages is followed through the design pattern's state machines to ensure everything functions as expected. Any problems discovered are corrected before the design pattern is applied to a SPL. Second, the SPL executable design patterns are also validated. The validation process is the similar to the validation process at the DRE level. However the design patterns must function as required by the SPL features. Thirdly, the application executable design patterns are also validated. The validation process is the similar to the validation process the SPL level. However the design

patterns must function as required by the application's features. Finally, executable application software architectures are validated using a feature and design pattern model based testing approach. This approach extends a feature driven model based testing technique (Olimpiew 2008; Olimpiew & Gomaa 2009) to explicitly address design patterns.

Validation for this research occurs through the FSW domain using two proof of concept, real world case studies. The FSW SPL is described in Chapter 6 and Appendix B. The case studies include modeling a small, spin stabilized spacecraft in a low earth orbit (Chapter 7 and Appendix C) and a large, three-axis stabilized spacecraft in heliocentric orbit (Appendix D). These case studies were selected because they validate the feasibility of applying the research to a wide range of FSW. The FSW SPL and case studies will be built following the approach described in this research. The cases studies will use several of the architectural and executable design patterns proposed in this research. The case studies will be validated for functional correctness following the proposed feature and design pattern based validation approach.

4 DISTRIBUTED, REAL-TIME AND EMBEDDED ARCHITECTURAL AND EXECUTABLE DESIGN PATTERNS

4.1 Introduction

This chapter presents an overview and the definitions for the architectural and executable design patterns used to build domain specific Distributed, Real-time and Embedded (DRE) software architectures. First, this chapter provides an overview of architectural design patterns and the executable design patterns. This includes a description of architecture views used to describe DRE architectural design patterns. Additionally, it describes the executable design patterns and how they are made executable. It then illustrates the complete set of architectural views and an executable version using the Centralized Control design pattern. Finally it lists all the architectural and executable design patterns created as part of this research. A brief description of each architectural and executable design pattern is found in Appendix A.

4.2 Architectural and Executable Design Pattern Overview

This research utilizes architectural and executable design patterns to serve as customizable templates that establish the foundation for domain specific DRE software

architectures. The variability within the design patterns is intended to be customized to a particular domain and application by software engineers. In order to use design patterns for the foundation of software architectures they must fully define the design pattern's objects, object interactions, object communication within the design pattern, and variability within the design pattern. This information is captured in the architectural design patterns using multiple architectural views. The executable design patterns contain an executable version of the design with more detail that can be systematically incorporated to make the software architectures executable.

This research utilizes the Unified Modeling Language (UML) to capture software architectural views. UML contains several diagrams to capture the software from different views. However, only the UML diagrams needed to appropriately specify the design patterns are used. This subsection describes the architectural views and the executable version in more detail.

4.2.1 Architectural Design Pattern Views

The first architectural view needed to specify the design pattern must define the objects involved in a design pattern need to be defined. UML Collaborations are “a description of a collection of objects that interact to implement some behavior within a context” (Rumbaugh et al. 2004). Therefore collaborations provide a great view to define the objects and behavior within a design pattern from a very high level. The collaboration diagram uses roles to define a description of objects that can participate in the

collaboration. Variability is captured using SPL stereotypes and multiplicities. The SPL stereotypes used are from the PLUS method (Gomaa 2005). For example, <<kernel>> is used to indicate a feature is common to all applications and <<optional>> is used to describe a feature that is provided by some applications.

Figure 4-1, illustrates a sample collaboration diagram. The entire collaboration is encapsulated using the collaboration notation, which is a dashed oval with the name on top. As seen in Figure 4-1, this collaboration is composed of two roles, which are role_0 and role_1. Additionally, role_0 and role_1 has some interaction because a connector is modeled between them. According to the multiplicities, there can only be one role_0. However this is variability in the amount of role_1s as indicated with its multiplicity and <<optional>> stereotype.

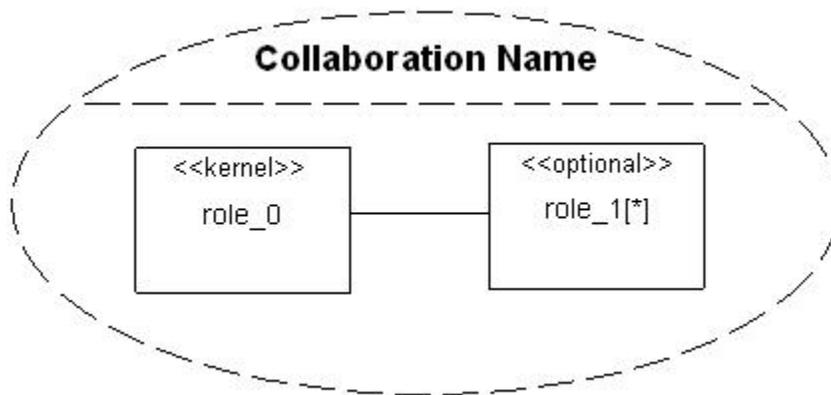


Figure 4-1 Example collaboration diagram

Secondly, an architectural view of how the objects can interact within the design pattern needs to be established. At the DRE level, there is significant variability in the object interactions since the actual number of the objects is variable. Therefore only a representative set of object interactions can be defined for the DRE architectural design patterns. Furthermore, the specific type of message communication is variable at the DRE. Therefore no assumptions are made about the message communication type. The design pattern's object interaction view will be captured using UML sequence diagrams (Rumbaugh et al. 2004). An example sequence diagram is depicted below in Figure 4-2. The top of the diagram shows two object roles. In this sequence, role_0 calls role_1's message_0 asynchronously. Then role_1 invokes role_0's message_1 synchronously.

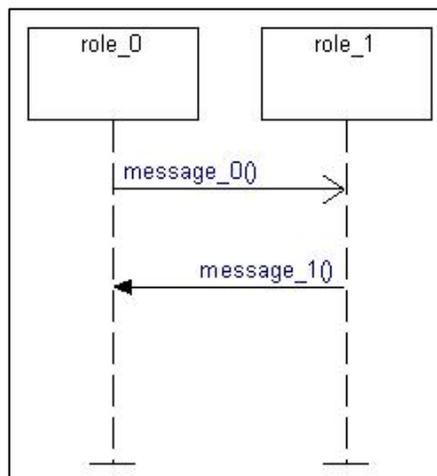


Figure 4-2 Example sequence diagram

The last architectural view needed to specify the design pattern must define the communication within the design pattern. The purpose of this view is to show the components with the required and provided ports they use to communicate with the other components. The component diagram (Rumbaugh et al. 2004) shows the components with required ports, provided ports, and connectors that inter-connect the components. At the DRE level, there is variability in the actual connector type, such as asynchronous message queue. To capture this variability, the DRE design patterns will only indicate where connector objects are needed. The variability in the components is modeled with SPL stereotypes.

An example component diagram is depicted in Figure 4-3. This figure shows how two components are interconnected. The connector is a bidirectional connector that joins Class_2's complex port RClass1 to Class_1's complex port PClass1.

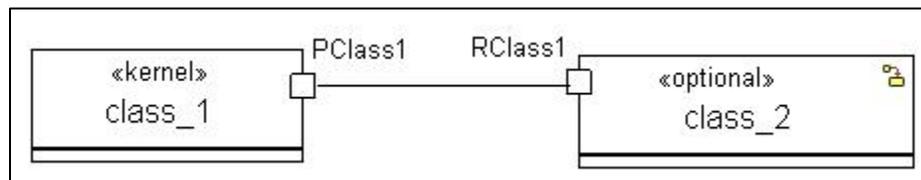


Figure 4-3 Example component diagram

To make the diagram readable, the port design is modeled on a separate component diagram in Figure 4-4. Class_1's port provides the iClass1 interface and requires the iClass2 interface. Class_2's port provides the iClass2 interface and requires the iClass1 interface. Examination of the port design shows that the PClass1 and RClass1 ports are compatible because the required interfaces can be connected to the provided interfaces.

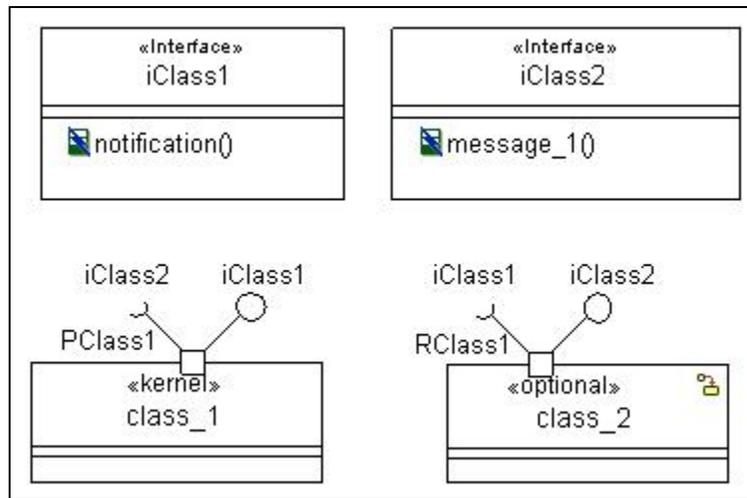


Figure 4-4 Example port design

4.2.2 Executable Design Patterns

After the design pattern is specified, the next step is to make an executable version of the design pattern. In order to accomplish this, internal behavior for each of the active objects in the design patterns needs to be defined. The internal behavior must capture which actions the object performs, such as executing an algorithm. If an object's actions

include communicating with other object, the port and message being sent must be defined. Additionally, the functionality in the passive objects must also be specified.

UML state machine diagrams are the perfect match to capture internal behavior. Additionally, state machines can be executed using Harel's executable state chart semantics (Harel 1997). State machines are composed of states, which represent the object's internal state. Transitions between states are specified using transitions that capture the trigger that causes the transition and any actions to be performed as part of the transition. Additionally, actions which the object performs can also be captured within the state using entry, exit, or do activities.

This research uses IBM Rational Rhapsody to build and execute the state machines. Therefore the actions performed are captured using IBM Rational Rhapsody's action language and event handling infrastructure. IBM Rational Rhapsody uses custom action language, which is a subset of the C++ language, to capture actions and to execute the model (IBM Corporation 2009). Thus, this action language is used to implement the objects actions. The action language is similar to C++, except there are a few additional reserved words and functions. For example, OUT_PORT is a reserved word used to specify the port in which messages are sent and GEN is a reserved word used to generate asynchronous messages as events. The messages must be specified on the consumer's provided interface in order to be invoked. When an event is generated, IBM Rational

Rhapsody event handling infrastructure handles the routing of events from the producer to the consumer. The details are found in (IBM Corporation 2009). When the consumer component receives the event, the appropriate state transition is taken and actions within that state are performed.

Figure 4-5 depicts an example state machine diagram for Class_2. This state machine is composed to two main states, which are Idle and Processing. The beginning state is Idle. If message_1 is received while Class_2 is in this state, then Class_2 will transition into Processing. In this state the object performs an output action and generates the complete event to indicate that processing is complete. This information is captured as an action within the Processing state using IBM Rational Rhapsody action language, as seen in Figure 4-5. This causes Class_2 to transition back to the Idle state. During this transition, the Class_2 also sends a notification message Class_1. The notification message is a provided interface as seen in Figure 4-4. This information is captured as a transition action shown after the trigger called complete. The semantics used are from IBM Rational Rhapsody action language where OUT_PORT specifies the port to use and GEN sends the event.

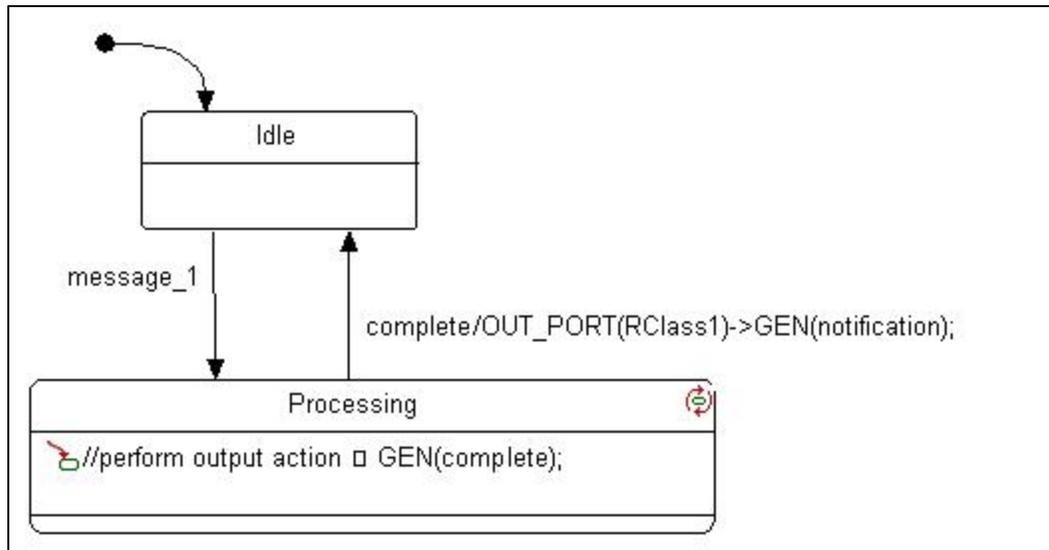


Figure 4-5 Example state machine diagram

4.3 Centralized Control Architectural Design Pattern

The Centralized Control design pattern (Gomaa 2005) involves one control component that provides overall control of the system by conceptually executing a state machine. This design pattern is included because it provides a control structure for DRE systems where overall control is centralized. This enables all the state dependent control to be contained in a single component, which makes it easy to understand and maintain. This pattern is suitable for small control applications because the centralized control can become a bottleneck on large systems.

The first architectural view captures the components that participate in the executable design pattern and their variability. This information is modeled using a collaboration diagram. The collaboration diagram for the Centralized Control executable design pattern is depicted in Figure 4-6. The design pattern is composed of four general purpose components. There is variability in the amount and type of Input_Components, Output_Components, and IO_Components that can be used when this pattern is applied. Therefore this variability is modeled using the <<optional>> stereotype and as a zero-or-more multiplicity in the collaboration diagram.

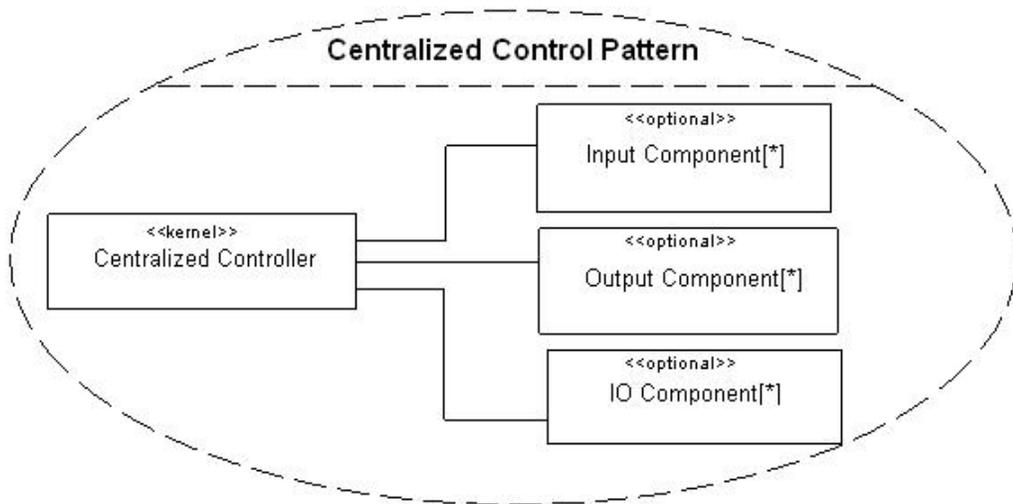


Figure 4-6 Collaboration diagram for Centralized Control design pattern

The next architectural view captures the dynamic interactions between the components are captured using interaction diagrams. Figure 4-7 depicts a representative object interaction sequence for the Centralized Control design pattern. In this interaction sequence, an external event from the external environment, called ENV, triggers the Input_Component to send an inputEventNotification to the Centralized_Controller. Based on this data, the Centralized_Controller then determines the appropriate response. In this particular scenario, the Centralized_Controller sends a command to the Output_Component to perform an action.

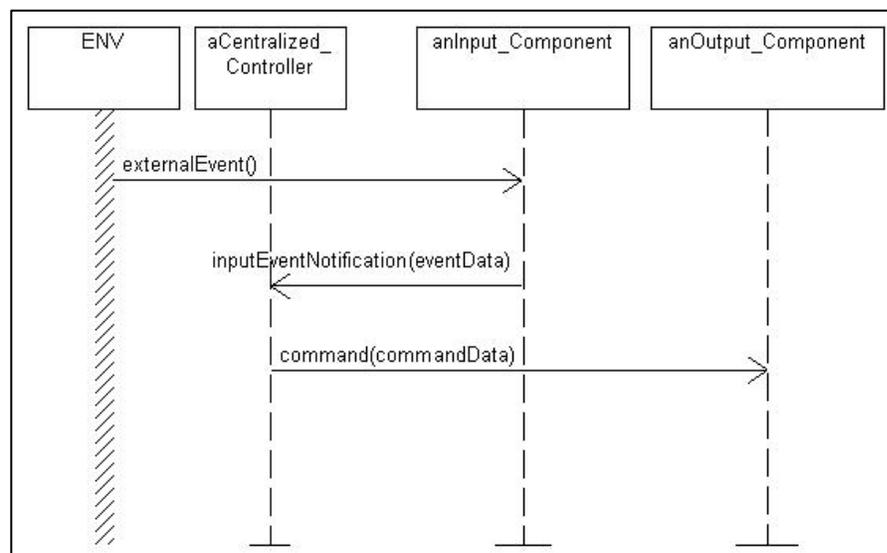


Figure 4-7 Representative sequence diagram for Centralized Control design pattern

Finally, the last architectural view captures the communication between the components in the design pattern using component diagrams. The component diagram for the Centralized Control design pattern is depicted in Figure 4-8. This diagram shows there is a connector between the Centralized_Controller's ROuput required port and the Output_Component's ROutputComp provided port. This connector is used to send output commands to the Output_Components. There is also a connector between the Centralized_Controller's RIO required port and the IO_Component's PIOComp provided port. This connector is used to send output commands to and receive inputs from the IO_Components. The last connector is between the Centralized_Controller's required port RInput and the Input_Component's provided port RInput. This connector is used to send data to the Centralized_Controller and when applicable, requesting data from the Input_Component.

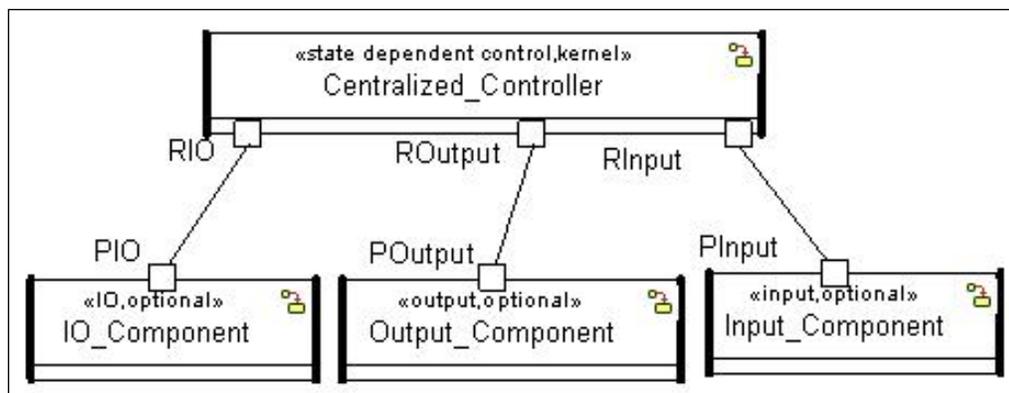


Figure 4-8 Component diagram for Centralized Control design pattern

In order for the ports to be connected, the required and provide interfaces between the ports must be compatible. This means the required interfaces on the port must match the provided interfaces on the connected port, and vice versa. Additionally, the variability in the interfaces must also be addresses. For example, Figure 4-9 depicts the port design of the Centralized Control design pattern. This shows that the connected ports have compatible required and provided interfaces. Since there is data type variability at the DRE level, the data types are modeled using the RhpAddress data type, which is IBM Rational Rhapsody's language independent object superclass similar to void* in C++.

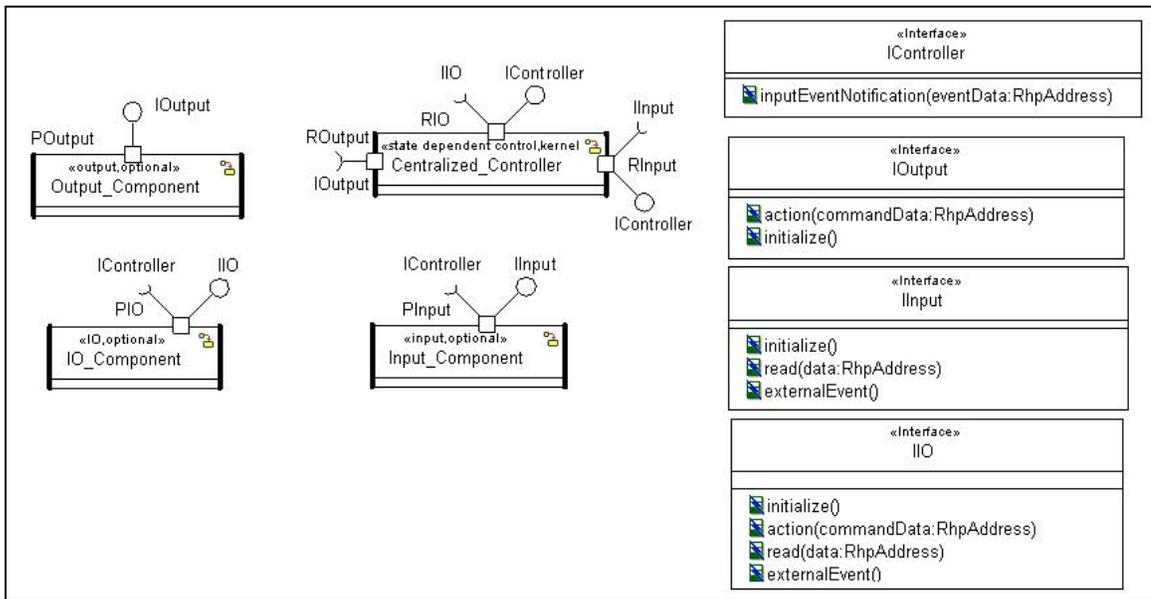


Figure 4-9 Port design for Centralized Control design pattern

4.4 Centralized Control Executable Design Pattern

The executable design patterns provide an executable version of the design pattern using state machines. The state machines capture the objects internal behavior and are made executable using executable object modeling with statecharts (Harel 1997). The executable version of the Centralized Control design pattern assumes there are one Input_Component, one Output_Component, and one IO_Component. Additionally, it assumes asynchronous message communication for all communication. Furthermore, it assumes that all inputs from the Input_Component result in a command to the Output_Component and all inputs from the IO_Component result in commands back to the IO_Component. This assumption enables all components in the design pattern to be exercised.

The state machine for the Input_Component is depicted in Figure 4-10. In the Idle state the component waits for an externalEvent from its external hardware device or read requests from the Centralized_Controller. When an externalEvent is received, the Input_Component transitions into the Preparing_Notification state, where it prepares a message to send to the Centralized_Controller. At the DRE level, the actions performed to create a message are variable; therefore it is modeled using a code stub as seen in Figure 4-10. Once the message is ready, the Input_Component generates an inputNotification event and transitions back to the Idle state. On the transition, an action is performed. This action involves sending the event data as an inputEventNotification

message to the Centralized_Controller. As seen in Figure 4-9, the inputEventNotification is a provided method of the Centralized_Controller and the port used corresponds to the ports in the component diagram in Figure 4-8. A similar set of actions is performed in response to a read event message, however the requested data is collected and sent back the Centralized_Controller.

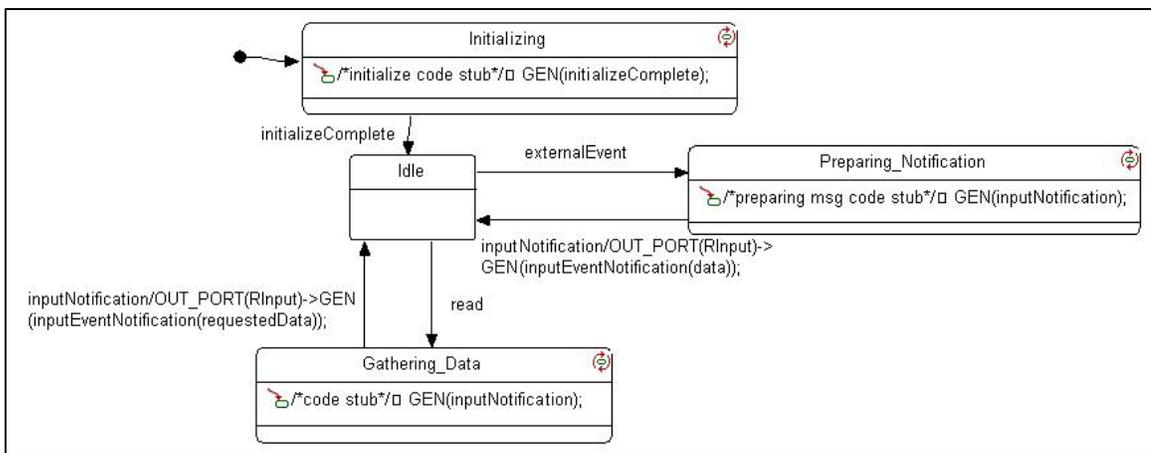


Figure 4-10 State machine for Input_Component

The state machine for the Centralized_Controller is depicted in Figure 4-11. The Centralized_Controller also has two states, which are Idle and Responding. In the Idle state the Centralized_Controller waits for inputEventNotifications from Input_Components and IO_Components. When an inputEventNotification is received, it transitions into the Responding state. Upon entry into this state, the

Centralized_Controller determines the appropriate response based on the input received. In this executable version, it is assumed that all inputs from the Input_Component result in a command to the Output_Component and all inputs from the IO_Component result in a command back to the IO_Component. Once the response is determined, the Centralized_Controller then sends a command to the appropriate component using the provided ports and methods defined in Figure 4-8 and Figure 4-9. This information is contained as an entry action within the Controlling state; however it is not depicted in Figure 4-11 in an effort to make the figure readable. Finally, after sending the command messages, the Centralized_Controller generates a processingComplete event and transitions back into the Idle state to wait for the next inputEventNotification.



Figure 4-11 State machine for Centralized Controller

Next, the state machine for the Output_Component is depicted in Figure 4-12. The Output_Component begins in the Idle state where it waits for commands from the Centralized_Controller. Once a command message is received, the Output_Component transitions into the Executing_Command state where it performs the appropriate actions

on the external hardware. At the DRE level, the actions the Output_Component performs are variable, therefore it is modeled using a code stub as seen in Figure 4-12. Once complete, it generates the processingComplete event and transitions back to the Idle state to wait for the next command.

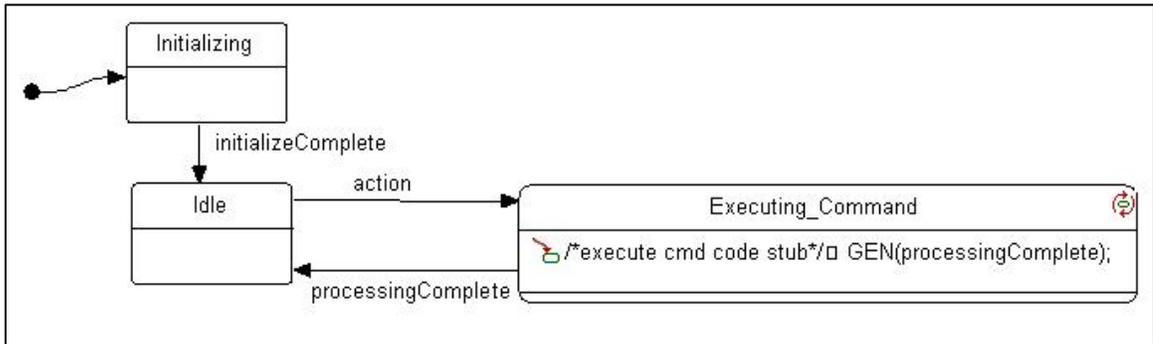


Figure 4-12 State machine for Output_Component

The last state machine for the Centralize Control design pattern is the IO_Component, which is depicted in Figure 4-13. The state machine for the IO_Component is slightly more complicated because it acts as both an input and IO component. The IO_Component begins in the Idle state within the Working state. In the Idle state the IO_Component waits for commands from the Centralized_Controller. When an action message is received, it transitions into the Executing_Command state where it performs the appropriate actions on the external hardware. At the DRE level, the actions the IO_Component performs are variable, therefore it is modeled using a code stub. The

code stub is modeled as an on entry action into the Executing_Command state, but not depicted in Figure 4-13 to keep the diagram readable. After it performs the necessary actions, it generates the processingComplete event and transitions back to the Idle state to wait for the next command. When a read message is received, a similar set of states and transitions occurs, however, it occurs in the Gathering Data state. The IO_Component is also responsible for listening to externalEvents from the hardware. Therefore if an externalEvent event is received, the IO_Component stops its current action in the Working state and transitions into the Preparing_Notification state. In the Preparing_Notification state it prepares a message to send to the Centralized_Controller. At the DRE level, the actions performed to create a message are variable. Thus this is modeled using a code stub in the on entry section of the Preparing Notification state, but not graphically depicted to keep the diagram readable. Once the message is ready, the IO_Component then sends the inputEventNotification message to the Centralized_Controller through the RIO port and transitions back to its previously interrupted location within the Working state.

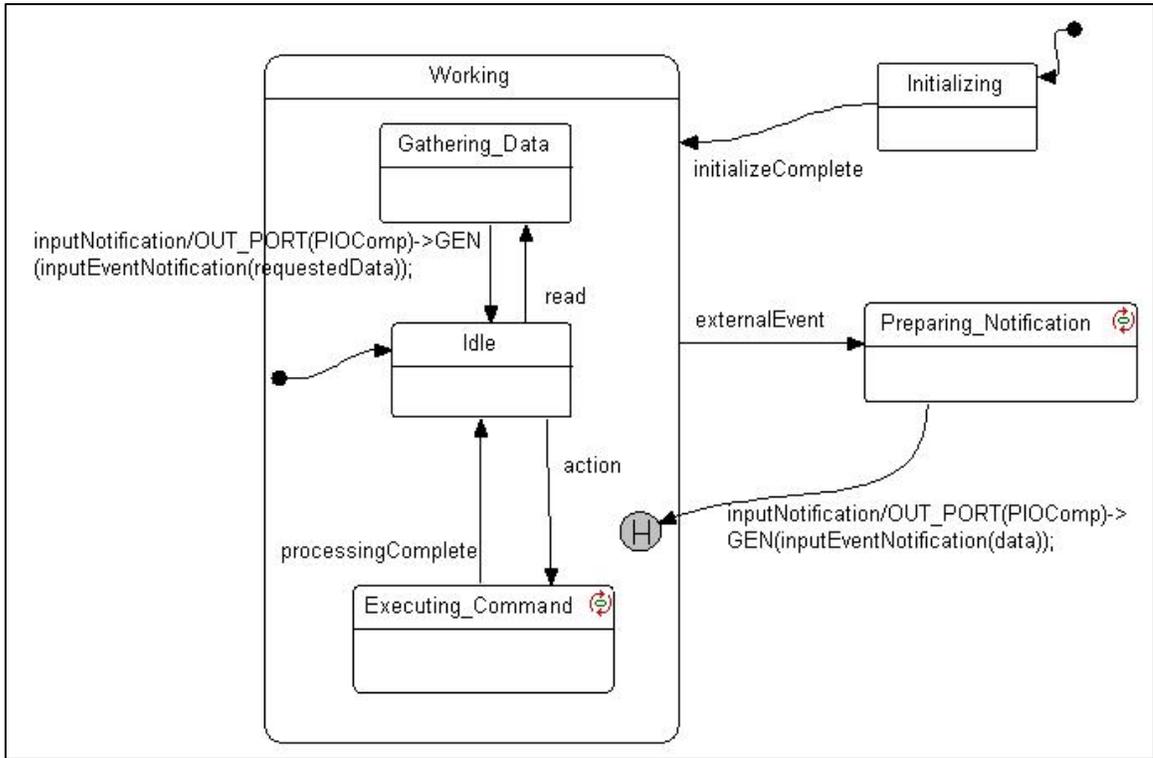


Figure 4-13 State machine for IO_Component

4.5 DRE Design Pattern Listing

In addition to the Centralized Control design pattern, the other DRE design patterns created as part of this research are listed in Table 4-1 and additional details are found in Appendix A.

Table 4-1 DRE design patterns

DRE Design Patterns	
Hierarchical Control Pattern (Gomaa 2005)	Distributed Control Pattern (Gomaa 2005)
Command Dispatcher design pattern (Dupire & Fernandez 2001)	Strategy design pattern (Gamma et al. 1995)
Pipes and Filters architectural design pattern, also referred to as Channel, (Buschmann et al. 1996; Douglass 2003)	Two Phase Commit design pattern (Gomaa 2005)
Master Slave design pattern (Buschmann et al. 1996).	Compound Transaction design pattern (Gomaa 2005)
Client Server design pattern, including the Multiple Client Multiple Server and Multitier Client Server variations (Gomaa 2005)	Publish Subscribe design pattern (Gomaa 2005; Buschmann et al. 1996; Gamma et al. 1995), also called Observer and Subscription/Notification
Broadcast design pattern (Gomaa 2005)	Protected Single Channel (Douglass 2003)
Multicast design pattern (Gomaa 2005)	Homogeneous Redundancy design pattern (Douglass 2003)
Layers (Gomaa 2005; Douglass 2003; Buschmann et al. 1996)	Triple Modular Redundancy design pattern (Douglass 2003)
Five-Layer Architecture (Douglass 2003)	Monitor-Actuator design pattern (Douglass 2003)
Asynchronous Message Communication (Gomaa 2005)*	Sanity Check design pattern (Douglass 2003)
Bidirectional Asynchronous Message Communication (Gomaa 2005)*	Watchdog design pattern (Douglass 2003)
Asynchronous Communication with Callback (Gomaa 2005)*	Safety Executive design pattern (Douglass 2003)
Synchronous with Reply (Gomaa 2005) *	Brokered Communication (Gomaa 2005; Buschmann et al. 1996; Douglass 2003)
Synchronous without Reply (Gomaa 2005)*	Heterogeneous Redundancy design pattern (Douglass 2003)

*executable versions were not created; relied on the message queues provided by IBM Rational Rhapsody's execution environment

5 BUILDING DOMAIN SPECIFIC SOFTWARE PRODUCT LINE ARCHITECTURES FROM DESIGN PATTERNS

5.1 Introduction

This chapter describes the novel pieces required to build domain specific distributed real-time and embedded (DRE) software architectures from design patterns. This approach utilizes architectural and executable design patterns and software product line (SPL) concepts to capture rules for integrating the design patterns to form domain specific software architectures.

5.2 Feature Modeling for Design Patterns

This research uses a feature model to identify the common and variable features in the SPL. A feature is a requirement or characteristic of some applications in a SPL. Feature modeling is not new. In fact, this research derives features from use cases and is modeled as described in the PLUS method (Gomaa 2005). However, feature modeling has differences when mapping to design patterns.

First, there are coarse grained features, called pattern specific features, which differentiate one pattern specific feature from another. For example, the FSW SPL has a pattern specific feature called Low Volume Command Execution and another pattern specific feature called High Volume Command Execution. These features provide similar functionality, however, they are differentiated in the amount of commands they are required to process.

Then, there are fine grained features, called pattern variability features, which influence the variability within pattern specific features. For example, in the FSW SPL there is a pattern variability feature called Heater. This pattern variability feature relates to variability in the Low Volume Command Execution pattern specific feature because it influences whether or not the FSW will execute commands involving a heater. The pattern specific features will ultimately be mapped to design patterns that support the feature. Pattern variability features will not be mapped to design patterns. Rather they will be modeled as having dependencies on pattern specific features.

Feature modeling is illustrated using a subset of the FSW SPL's feature model in Figure 5-1. This example shows a pattern specific feature group that has variability captured in three alternative features. Additionally, it shows that the optional Heater pattern variability feature influences variability in the Command Execution feature group since it influences whether or not the FSW is required to process commands for a heater.

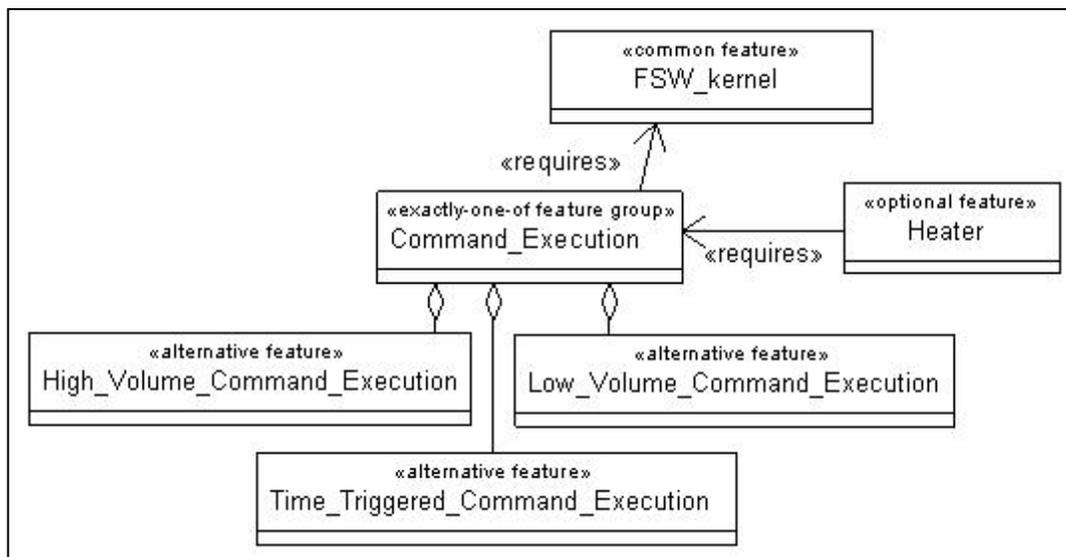


Figure 5-1 Subset of FSW SPL feature model

5.3 Feature to Design Pattern Mapping

The feature to design pattern mapping is where this approach is also novel and differentiates itself from other SPL approaches. It identifies where software architectural design patterns can be used in the analysis model and then relates them back to the SPL features. Mapping SPL features to design patterns has several benefits. First, it ensures that SPL and SPL members leverage the benefits of software architectural design patterns. Second, the feature to design pattern mapping captures the rules for selecting design patterns. For example, consider two alternative pattern specific features that are each mapped to a different design pattern. Since these features are alternatives they cannot both be selected for a particular application. Therefore the design patterns that are

mapped to these features cannot be combined because the design patterns are alternatives. Finally, capturing SPL variability using design patterns, as opposed to components and connectors, results in an SPL with variability captured at a higher level of granularity. This results in architectural flexibility and less modeling at the SPL because the design patterns contain customizable components and connections rather than specific classes. This provides a more scalable approach to handling SPL domains with a significant amount of variability.

The feature to design pattern mapping is created following a systematic process. During the dynamic modeling, each pattern specific feature in the feature model follows these steps:

1. Create an interaction diagram based on the pattern specific feature's associated use case.
2. For each pattern variability feature that has a dependency on the pattern specific feature, model the related variability in the interaction diagram.
3. Analyze the object interactions and identify any design patterns that can be applied.

The process for deriving the feature to design pattern mapping is illustrated using the FSW SPL's Low Volume Command Execution pattern specific feature. This pattern specific feature is derived from the Execute Commands use case where the command volume variation point is low and strict temporal predictability variation point is false.

Figure 5-2 show a subset of the communication diagram for feature. In this example, due to the small amount of commands that need processing, one controller receives a set of ground commands from the ground station. It then determines the sequence of the actions to ensure that it does not put the spacecraft in an unsafe state. Next it executes the commands by invoking the appropriate actions on the output and IO components identified in the pattern specific feature. The Low Volume Command Execution pattern specific feature states the IO and output components include a Memory Storage Device IO component and an Antenna output component.

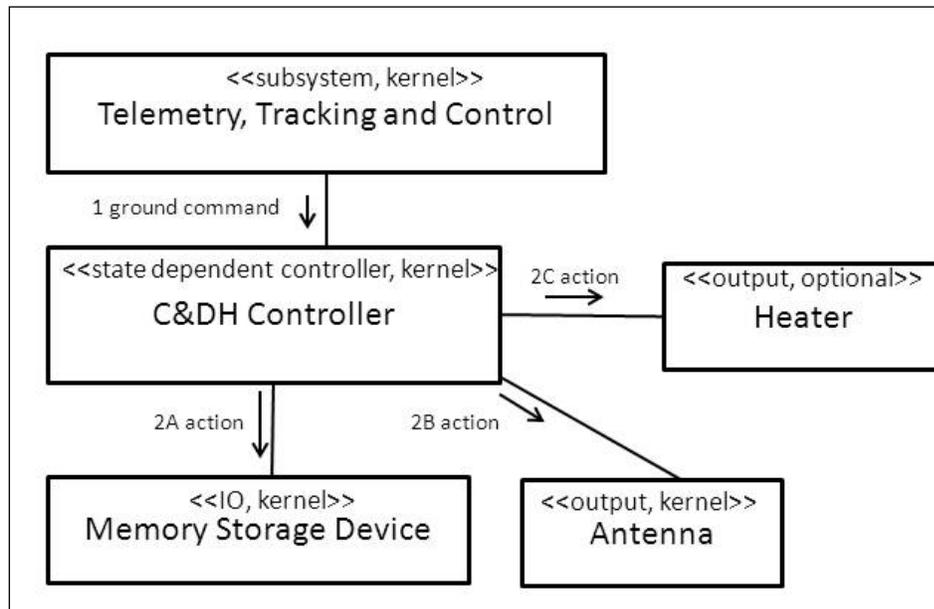


Figure 5-2 Subset interaction diagram for the Low Volume Command Execution pattern specific feature

Next, the variability from the pattern variability features is added to the interaction diagram. The Low Volume Command Execution pattern specific feature is influenced by the Heater pattern variability feature. The Heater pattern variability feature captures whether or not a spacecraft is required to have a heater to adjust its temperature. This variability is captured using an optional Heater component on the interaction diagram.

Finally, the interactions involved in the dynamic model are analyzed and applicable design patterns are identified. In the Low Volume Command Execution pattern specific feature, the interactions are consistent with Centralized Control design pattern (Gomaa 2011). Therefore this pattern specific feature is mapped to the Centralized Control design pattern.

The feature to design pattern mapping is documented in tabular format. The table captures the pattern specific features and their associated design pattern or pre-integrated design pattern combination. Pattern variability features are captured along and the variability of the features is also documented. Table 5-1 illustrates a sample feature to design pattern mapping for the FSW SPL.

Table 5-1 Sample Feature to Design Pattern Mapping

Feature Group	Variability	Feature	Design Pattern/Pattern Combination
<<exactly-one-of feature group>> Command Execution	alternative	High Volume Command Execution	Hierarchical Control
	alternative	Low Volume Command Execution	Centralized Control
	alternative	Time Triggered Command Execution	Distributed Control
N/A	optional	Heater	

5.4 Levels of Executable Design Pattern Modeling

Another novel contribution of this research is the inclusion of variable architectural and executable design patterns. The purpose of the architectural design patterns is to specify the design pattern and to identify variability within the design patterns. The purpose of the executable design patterns is to provide an executable version of the design pattern, which can be systematically incorporated into executable software architectures. To help guide engineers in the application of design patterns, this research developed three levels of architectural and executable design patterns modeling. These levels include a DRE level, SPL level, and application level. The specifics of the DRE architectural and executable design patterns are described in Chapter 4. The SPL and application levels of architectural and executable design patterns are described in more detail in the following subsections.

5.4.1 Software Product Line Architectural Design Patterns

The second level of architectural and executable design patterns is at the SPL level. This level contains the initial customizations of the DRE level architectural executable design patterns to meet the needs of the SPL features. The purpose of the SPL design patterns is to capture customizations that can be applied across a SPL to facilitate the application of design patterns during the application engineering phases. This is accomplished by customizing the DRE design pattern to the SPL pattern specific feature it is mapped to and capturing the variability from other pattern variability features that relate to variations in the design pattern. This is a systematic process that involves customizing the architectural views and variability within the design pattern. The steps involved in customizing each architectural view are described below in more detail.

First, the architectural view used is a collaboration diagram, which models the components that participate in the design pattern. To update a DRE level collaboration diagram to a SPL level collaboration diagram, the following steps are performed:

1. For each kernel component with a multiplicity of one and its variants
 - a. Update the component name to reflect the needs of the design pattern's pattern specific feature
 - b. If a pattern variability feature contains alternatives for this component, then variant components are created for each alternative
2. For each kernel component with a multiplicity of one or many and its variants

- a. Determine the number of this type of component that is required for the needs of the design pattern's associated pattern specific or pattern variability feature
 - b. For each type of this component required, create an SPL specific version with a name and multiplicity that is reflective of the SPL features
 - c. If a pattern variability feature contains alternatives for this component, then variant components are created for each alternative
3. For each optional component with a zero or many multiplicity and its variants
 - a. Determine the number of that component which is required based on the design pattern's associated pattern specific feature or the pattern variability feature pointing to a variation point in the design pattern
 - b. If the component is not required, then remove from collaboration diagram
 - c. If the component is required, for each type of this component required, create an SPL specific version with a name and multiplicity that is reflective of the appropriate SPL pattern base or pattern variability features

A FSW SPL is used to illustrate the process of updating the DRE collaboration diagram to a SPL collaboration diagram. This example involves converting the DRE level Centralized Control design pattern into the FSW SPL's Centralized Control design pattern, which is mapped to the Low Volume Command Execution pattern specific feature and influenced by numerous pattern variability features including the Memory

Storage Device exactly one of feature group and Attitude Control Device at least one of feature group. First, the DRE level Centralized Control executable design pattern has one kernel component with a multiplicity of one, which must be updated with a SPL name based on the features. This design pattern is being applied within the command and data handling (C&DH) subsystem to perform that processing and execution of ground commands. Therefore the component is updated to be called the CDH_Centralized_Controller as seen in Figure 5-3.

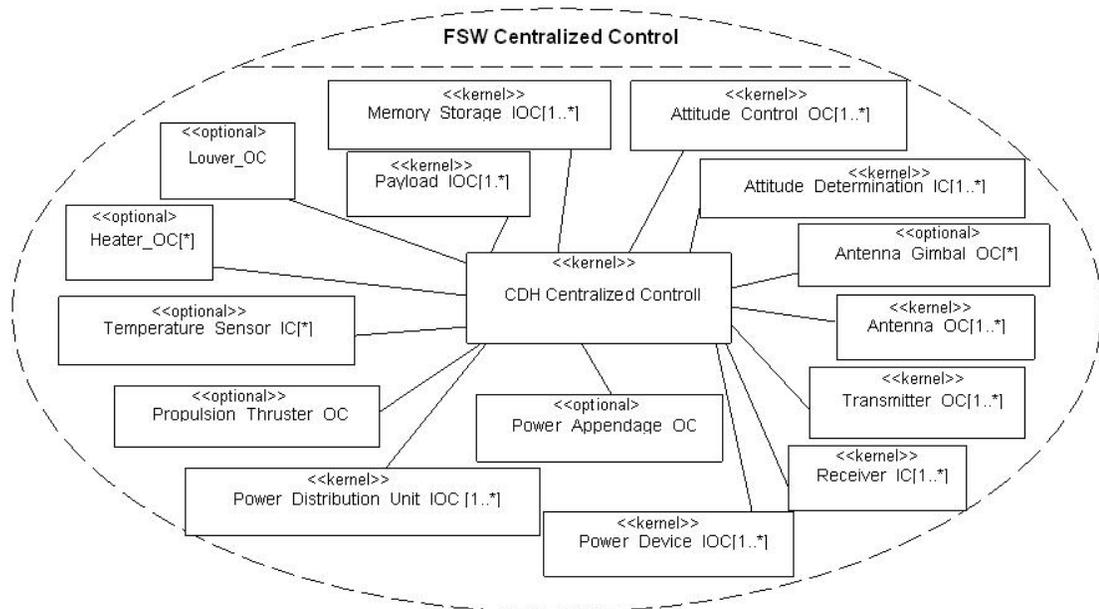


Figure 5-3 Example FSW level Centralized Control collaboration diagram

Second, the DRE level Centralized Control executable design pattern contains an optional IO_Component with zero or many multiplicities. Based on the pattern specific and pattern variability features it is determined that this component is required because the FSW SPL has four features relating to IO devices. The Low Volume Command Execution pattern specific feature requires all FSW to interface with one or more memory storage devices, power distribution units, power devices, and payload devices. Therefore four versions of the IO_Component are created and given SPL names reflective of the SPL's devices. Additionally, the multiplicity is updated to one or many and the variability is updated to kernel to reflect the needs the feature. Additionally, the Memory Storage Device Type, Power Device Type, and Payload Device Type pattern variability features also point to variability in this design pattern. Each of these features contains a set of alternative device types that can be selected for an SPL member. To reflect this information, the different device types are modeled as variant components and use the variant stereotype. However, they are not graphically depicted in Figure 5-3 to keep the diagram readable. This same process is followed for the other optional components with zero or many multiplicities in the DRE Centralized Control collaboration diagram. While this pattern may look overly centralized, the Centralized Control design pattern is only used when there are a small amount of commands to execute and hardware to control. Therefore, when this pattern is customized to an application, only an application specific subset of the optional devices is selected.

Next, the DRE level architectural design patterns contain interaction diagrams illustrating representative object interactions. The process to update a DRE interaction diagram is as follows:

1. Identify the interaction sequence or sequences from the design pattern's associated pattern specific feature and pattern variability features
2. For each object interaction sequence identified
 - a. Create an interaction diagram using the SPL specific components
 - i. If the precise sequence of component interactions is known, then they should be modeled
 - ii. If the precise sequence of component interactions is application specific, then a representative set of object interactions is captured
 - b. If the interaction diagram contains the precise sequence of component interactions, then the specific type of message communication used is identified and updated on the diagram during design modeling phase

The Centralized Control design pattern for the FSW SPL is used to illustrate the process of updating the DRE level interaction diagram to a SPL collaboration diagrams. This design pattern is mapped to the Low Volume Command Execution pattern specific feature and several other pattern variability features related to variability in the design pattern. The Low Volume Command Execution pattern specific feature captures one scenario where the FSW processes and executes a set of ground commands, which involves invoking actions on the input, output, and IO components. The type and amount

of input, output, and IO components is influenced by several other features, such as the Memory Storage Device pattern variability feature group that captures the alternative memory storage devices that can be used. Due to the amount of variability in this scenario, only a representative set of object interactions is created for this use case. The set of representative object interactions for this executable design pattern is depicted in Figure 5-4. Notice, the SPL components are used rather than the general purpose DRE level components. The variant components are not shown in Figure 5-4 to keep the diagram readable. A set of alternative branches is used to represent that different actions will result in different actions being performed on different devices.

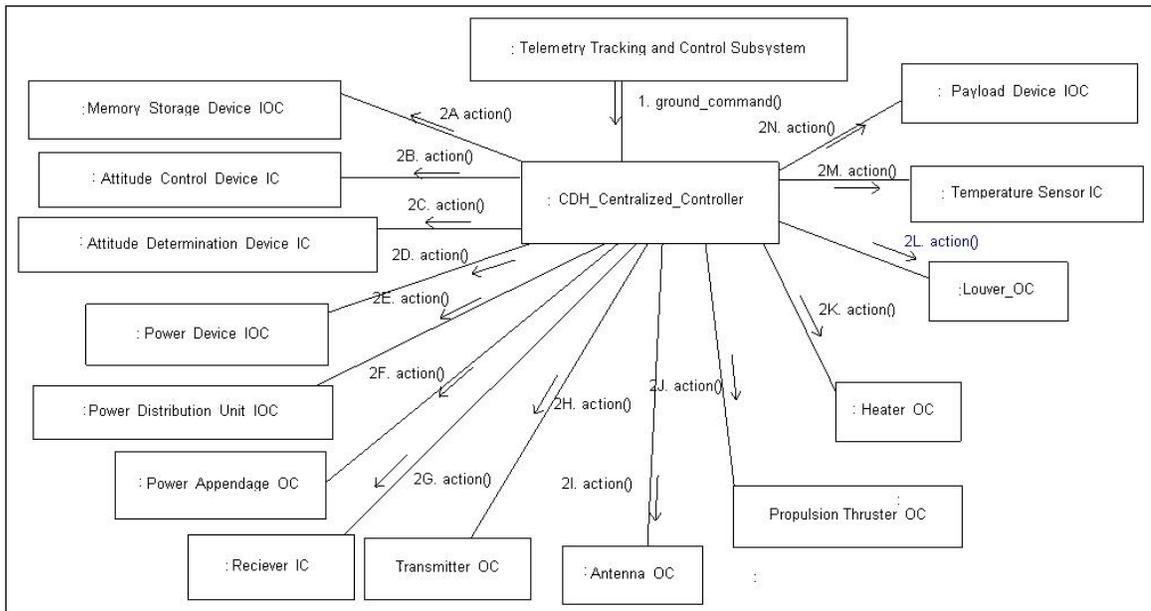


Figure 5-4 Representative interaction diagram for FSW Centralized Control

Finally, the last architectural view that needs to be updated is the communication between the components in the design pattern, which is captured in the component diagram. The SPL level component diagram is derived from the DRE level component diagram, by following this systematic process:

1. Components are updated to reflect the SPL components, as identified in the collaboration diagram
2. Connectors from the DRE level components are maintained, to be consistent with the design pattern. However, the port names are updated to reflect the SPL component
3. Interfaces are updated to reflect SPL components. Changes include
 - a. Updating and adding SPL specific methods based on the needs of the pattern specific or pattern variability features
 - b. Updating and adding SPL data types methods based on the needs of the pattern specific or pattern variability features

This process is illustrated in Figure 5-5 using the FSW SPL's Centralized Control executable design pattern. Figure 5-5 shows that the general purpose Input Component is customized to the FSW SPL specific input components, which are the Receiver_IC, Temperature_Sensor_IC, and Attitude_Determination_IC. Additionally, the names of the required and provided ports are also updated to reflect the FSW SPL specific components. The same changes have also been applied to the output and IO components.

All the variant input, output, and IO devices are also modeled in the component diagram.

However the variants are not shown in Figure 5-5 to keep the diagram readable.

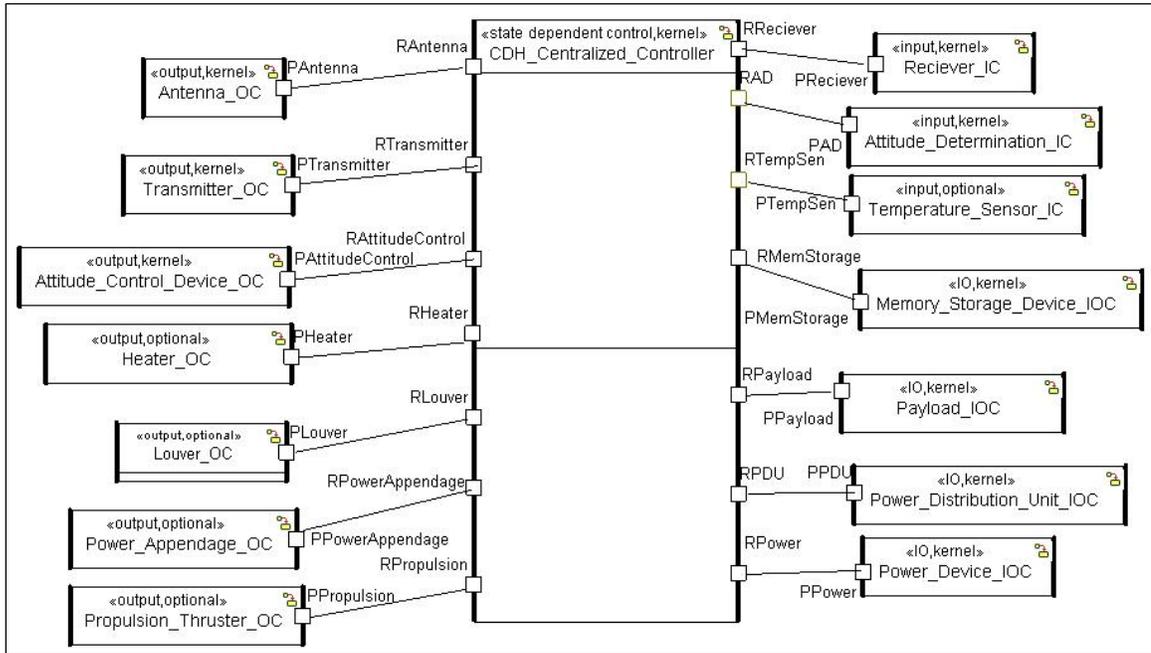


Figure 5-5 FSW Centralized Control execute design pattern component diagram

Finally, the required and provided interfaces between the ports must be customized to reflect the SPL customizations. These changes are illustrated on the port design component diagram. Figure 5-6 shows a subset of the port design for the FSW Centralized Control executable design pattern template. This shows that the connected

ports have compatible required and provided interfaces that are reflective of the FSW SPL.

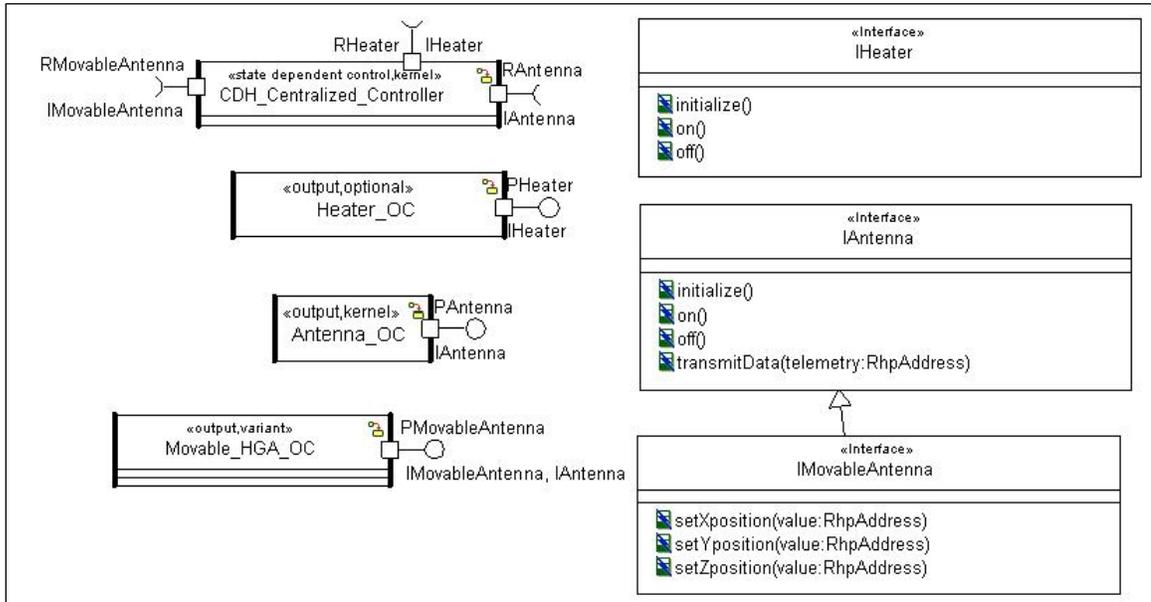


Figure 5-6 Port design for the FSW SPL Centralized Control executable design pattern

For example, instead of listing a general output component from the Centralized Control executable design pattern, the Heater_OC, Antenna_OC, and variant Moveable_Antenna_OC are shown, along with an updated port names. Their interfaces are updated versions of the IOutput interface from the Centralized Control DRE architectural design pattern. The general purpose action() method is replaced with FSW SPL specific methods for the Heater_OC, Antenna_OC, and variant

Moveable_Antenna_OC must provide based on the SPL features. For instance, the Low Volume Command Execution pattern specific feature requires the ability to turn the heaters and antenna on and off. Therefore the FSW SPL methods are added that provide this functionality.

5.4.2 Software Produce Line Executable Design Patterns

In addition to updating the design pattern's specification, the DRE executable design pattern also must be updated for the SPL. This involves systematic process involves:

1. Documenting assumptions made about the variability and object interactions
2. If a pattern specific feature or pattern variability feature extends the behavior of the object, then this is modeled as a new state.
3. If a pattern specific feature or pattern variability feature refines some behavior, then this can be modeled in substates within an existing state.
4. If a pattern specific feature or pattern variability feature requires the component to send message to another component, then this is modeled as an action within a state or on a transition. If there is variability in the object communication then assumptions should be made about the communication and documented.
5. If a pattern specific feature or pattern variability feature contains application specific logic, such as a processing algorithm, can be model as an action or activity within a state.

The process for updating a state machine is illustrated using CDH Centralized Controller, which is the FSW SPL component for the DRE level Centralized Controller. According

to the Low Volume Command Execution pattern specific feature, the CDH Centralized Controller must manage and take into account the spacecraft mode during command execution. The mode management function extends the behavior of the Centralized Controller. Therefore common modes, including launch mode, safe mode, and normal mode are modeled as new states at the highest level in the state machine hierarchy, as seen in Figure 5-7. Within each of these states are the original Idle and Controlling states from the DRE level Centralized Controller. Also according to the Low Volume Command Execution pattern specific feature, the CDH Centralized Controller must validate all ground commands or responses to onboard events to ensure that it does not put the spacecraft in an unsafe state. This functionality refines the behavior of the Controlling state. Therefore this logic is modeled as substates within each of the Controlling state. These substates are modeled within each of the Controlling States, but to make the diagram readable they are only depicted in the Normal Mode's Controlling state in Figure 5-7. The specific algorithms and logic used within the Validating Command, Executing Command Logging Commands and Rejecting commands substates is application specific. Therefore stub placeholders are models as actions within these states.

However, in order to make the design pattern executable, assumptions must be made about communication between the CDH_Centralized_Controller and the input, output, and IO components. For example, it is assumed that commands will be sent as integers.

A command with a value of eight will result in the CDH_Centralized_Controller invoking the Heater_OC's on() method through the RHeater port, as defined in Figure 5-5 and Figure 5-6. This information is captured within the Controlling states, but not depicted in Figure 5-7 to keep the diagram readable.

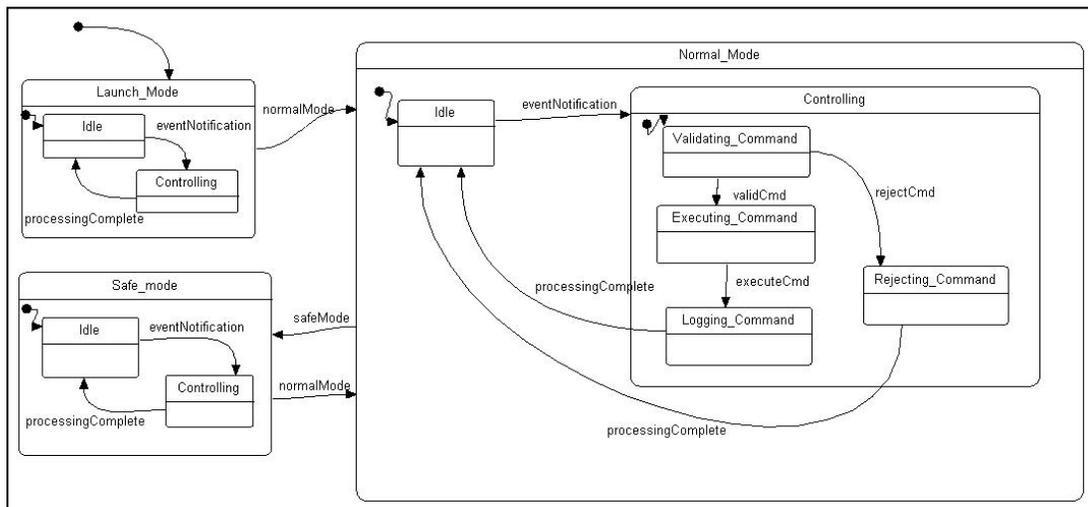


Figure 5-7 CDH Centralized Controller's state machine

5.4.3 Application Architectural Design Patterns

The final level of architectural design patterns is the application level. This level builds on the SPL architectural design patterns, where the SPL architectural design patterns are customized to meet the needs of a specific SPL member. This is a systematic process that involves customizing the architectural views and variability within the design

pattern. The steps involved in customizing each view and the executable version is described below in more detail.

First, the SPL architectural design patterns use a collaboration diagram to model the components that participate in the design pattern. To update a SPL level collaboration diagram to an application level collaboration diagram, the following steps are performed:

1. For each kernel component with a multiplicity of one, if the component has variants then only use the application specific variant, this is determined from the pattern variability feature selection.
2. For each kernel component with a multiplicity of one or many
 - a. Determine the number of this type of component that is required for the needs of the design pattern's associated pattern specific or pattern variability feature selection.
 - b. If the component has variants then only use the application specific variants, which is determined from the pattern variability feature selection
 - c. If the component has an application specific variant that could not be defined in the SPL, then create an application specific version of the variant component.
3. For each optional component with a zero or many multiplicity

- a. Determine the number of this type of component that is required for the needs of the design pattern's associated pattern specific or pattern variability feature selection.
- b. If the component is not required, then remove from collaboration diagram.
- c. If the component is required and has variants then only use the application specific variant, which is determined from the pattern variability feature selection.
- d. If the component is required and has an application specific variant that could not be defined in the SPL, then create an application specific version of the variant component.

The FSW SPL is used to illustrate the process of customizing a SPL collaboration diagram into an application specific diagram. Figure 5-8 depicts the Centralized Control design pattern collaboration diagram for the Student Nitric Oxide Explorer (SNOE), which is a SPL member that involves a small spin stabilized spacecraft. First, the kernel CDH Centralized Controller component with a multiplicity of one does not have any variants since the variability in this feature is due to the components being controlled. Next, each of the SPL kernel components with multiplicities of one or many are customized based on the pattern specific and pattern variability feature selection. For example, instead of using the Attitude_Determination_IC, the collaboration diagram is customized to just use the particular attitude determination device variants and multiplicities. Figure 5-8 shows SNOE uses multiple horizon crossing indicators and one

magnetometer to collect measurements to determine the spacecraft's attitude. Additionally, SNOE contains four payload devices that are unique to the application, which could not be defined in the SPL. Therefore these application specific variants of the Payload_IOC component are created. Finally each of the optional component with a zero or many multiplicity is customized based on the pattern specific and pattern variability feature selection. SNOE did not select any of the pattern variability features associated with the optional components therefore they are removed from the diagram.

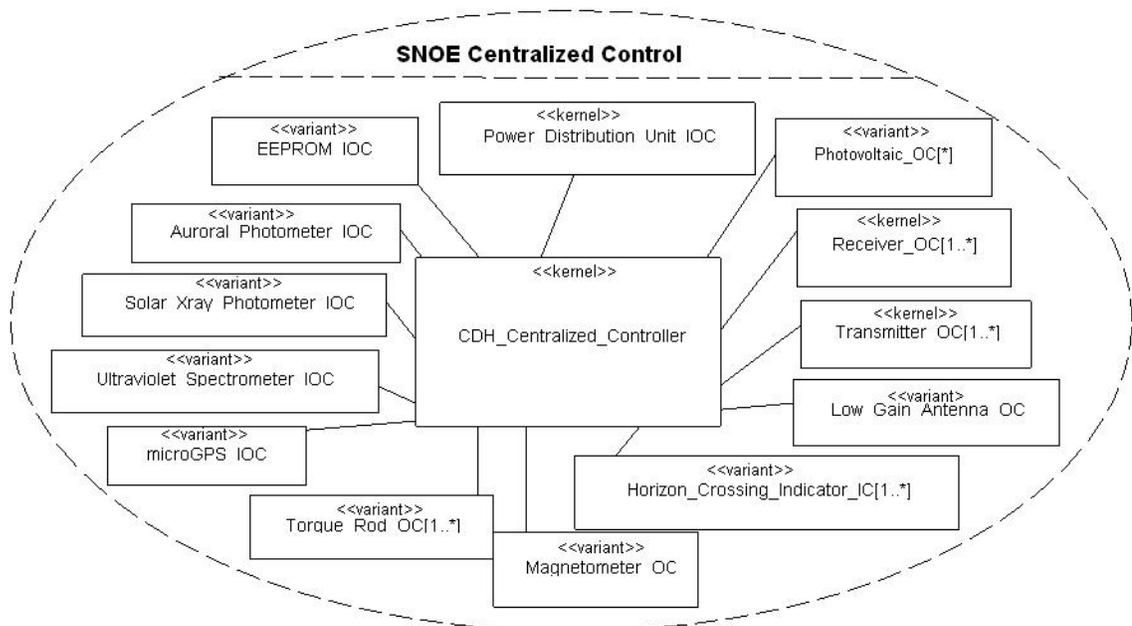


Figure 5-8 FSW application specific Pipes and Filters collaboration diagram

Next, the SPL level design pattern's interaction diagrams are customized for the application. To customize the interaction diagram the following steps are followed:

1. For each interaction diagram that contains the precise sequence of object interactions, no customizations are required.
2. For each interaction diagram that contains a representative sequence
 - a. Update the components participating in the interaction diagram to reflect the application specific components and variant selected, as defined in the collaboration diagram.
 - b. Update the sequence of messages to be precise sequence used in the application based on the pattern specific and pattern variability feature selections.

For example, Figure 5-9 shows an application specific sequence diagram for SNOE to reinitialize its low gain antenna based on a ground command. This interaction diagram is based on the FSW SFL execute commands interaction diagram in Figure 5-4, which contains a representative set of object interactions. In this example, the CDH_Centralized_Controller receives a ground command to reinitialize the spacecraft's low gain antenna from the Telemetry Tracking and Control subsystem as an inputEventNotification. Since the specific antenna variant is known, the specific interactions with this output component can now be modeled. In the example in Figure 5-9, the CDH_Centralized_Controller reinitializes the low gain antenna by invoking a series of actions on the Low_Gain_Antenna_OC output component.

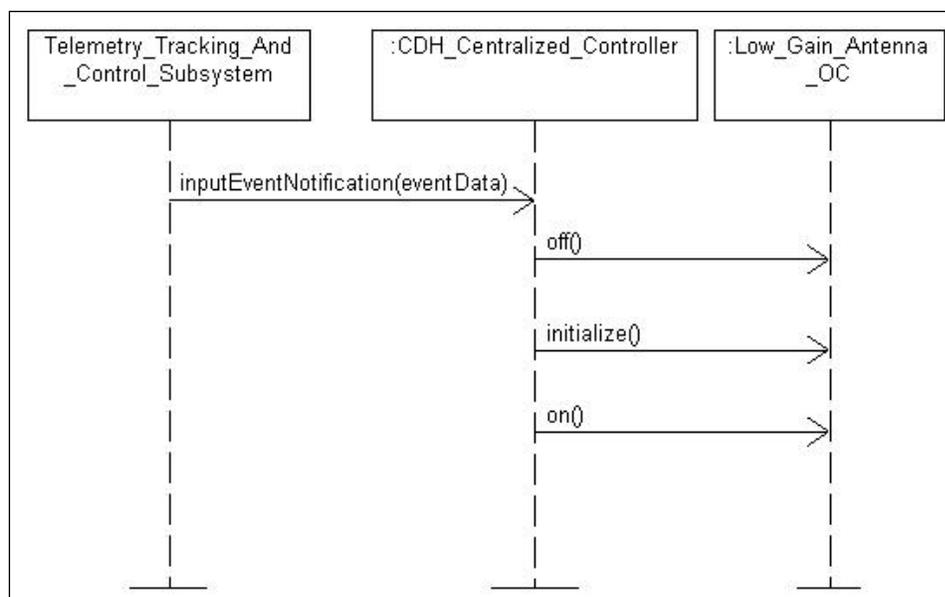


Figure 5-9 SNOE application specific sequence diagram for executing command to adjust attitude

Finally, the last step in updating the design patterns architectural views is to customize component communication for the application. The component diagram is derived from the SPL level component diagram using the following process:

1. Components are updated to reflect the application specific variants, as identified in the collaboration diagram.
2. For each application specific variant that was not captured in the SPL
 - a. Connectors from the SPL level components base class are maintained, to be consistent with the design pattern. However, the port names are updated to reflect the application component.

- b. Interfaces are updated to reflect application specific components. Changes include
 - i. Updating and adding SPL specific methods based on the needs of the application.
 - ii. Updating and adding SPL data types methods based on the needs of the application.

This process illustrated using SNOE from the FSW SPL. SNOE's component diagram for the Centralized Control design pattern is shown in Figure 5-10. First, it can be seen that the FSW SPL components are replaced with SNOE's variant selection. For example, the FSW SPL Antenna_OC component is replaced with SNOE's variants, which is the Low_Gain_Antenna_OC component.

Second, optional components not used are removed. For example, SNOE does not utilize a heater, thus the Heater_OC is removed. Finally, SNOE's unique payload variant components, which were not defined in the FSW SPL, are also added to the diagram. These include the Solar_Xray_Photometer_IOC, microGPS_IOC, Auroral_Photometer_IOC, and Ultra_Violet_Spectrometer_IOC. They maintain the connector to the CDH_Centralized_Controller, which was from the Payload_IOC component. However, their port names are also updated to be reflective of their names.

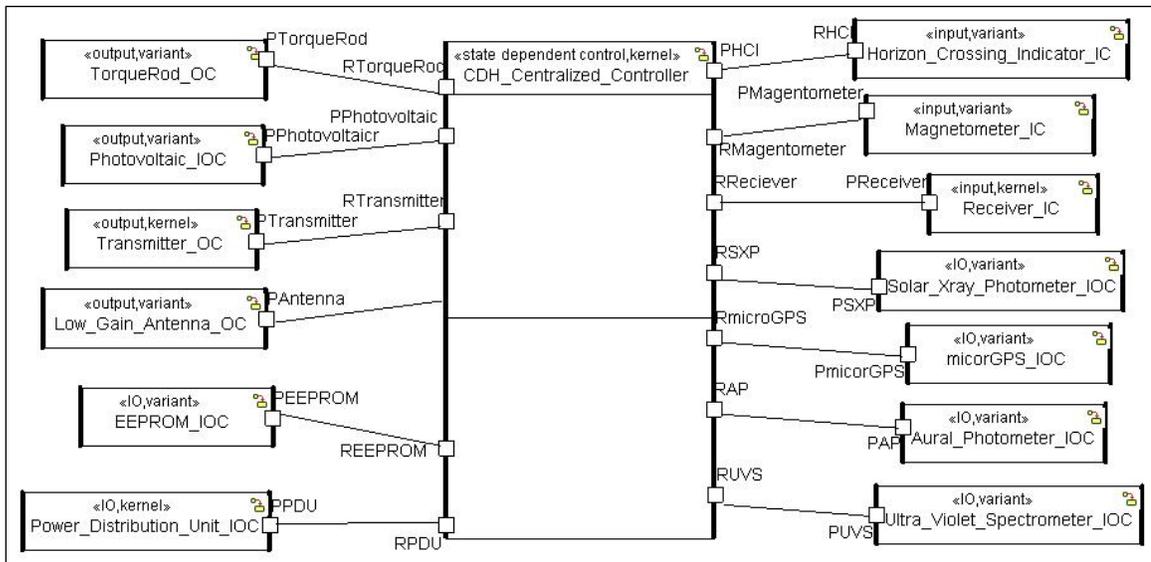


Figure 5-10 SNOE Centralized Control component diagram

The design of the ports is also updated to reflect the application specific information. A subset of the port design for SNOE’s Centralized Control design pattern is shown in Figure 5-11. The Low_Gain_Antenna_OC is SNOE’s selected variant for the Antenna_OC. Therefore the Low_Gain_Antenna_OC and its interface are used. SNOE’s unique payload variants should also be updated to include their application specific interfaces.

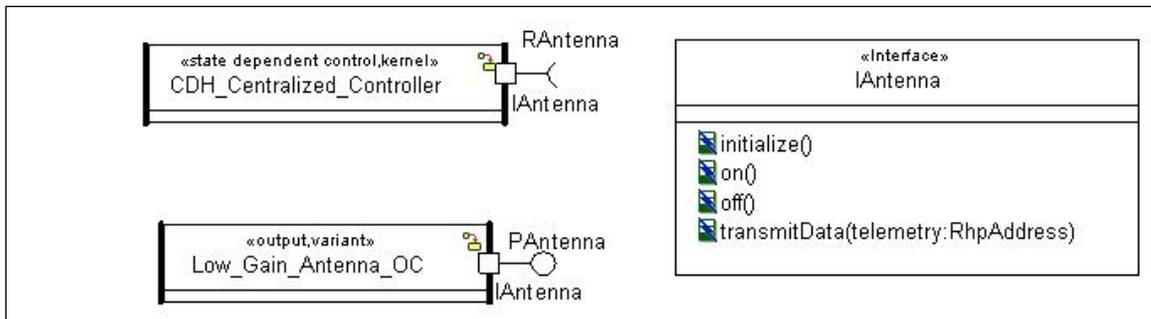


Figure 5-11 Subset port design for SNOE's Centralized Control design pattern

5.4.4 Application Executable Design Patterns

In addition to customizing the design pattern's architectural views, the executable version of the design pattern also must be updated for the application. This involves systematic process involves:

1. For each selected component or variant captured in the SPL
 - a. If the selected pattern variability features influences the message communication with other components, then this is modeled as an action within a state or on a transition.
 - b. If the selected pattern variability features relate to application specific logic, such as a processing algorithm, then this can be model as an action or activity within a state.
2. For each application specific variant not captured in the SPL

- a. If the application variant extends the behavior of the object, then this is modeled as a new state.
- b. If the application variant refines some behavior, then this can be modeled in substates within an existing state.
- c. If the application variant requires the component to send message to another component, then this is modeled as an action within a state or on a transition.
- d. If the application variant contains application specific logic, such as a processing algorithm, it can be model as an action or activity within a state.

The process for updating a state machine is illustrated using SNOE's CDH Centralized Controller, which is a customized version of the FSW SPL component shown in Figure 5-7. SNOE does not refine the behavior therefore the state machine is the same as the SPL CDH Centralized Controller in Figure 5-7. However, now the precise set of components that the CDH_Centralized_Controller communications with is known.

Therefore the precise object communication between the CDH_Centralized_Controller and the IO components is added as actions within the Executing Command substate.

The methods and ports used to communicate with other objects are defined in Figure 5-10 and Figure 5-11. The syntax of these messages is OUT_PORT(*portname*)->GEN(*method*), where *portname* is replaced with the actual port name and *method* is replaced with the actual method name.

5.5 Design Pattern Interconnection

The final novel contribution of this research is capturing the design pattern interconnections. This is used to illustrate how the design patterns are integrated to form software architectures for the SPL and application. A use case scenario driven approach is used to interconnect design patterns to achieve the SPL functionality. Then during application engineering just the selected design patterns will be used. The steps modeled during the SPL engineering and application engineering phases are described in the subsequent subsections.

5.5.1 Software Product Line Level Design Pattern Interconnection

First, use case models are commonly used to capture and document SPL requirements. However, the sequence of events and its variability is not very precise. This makes determining how the design patterns are integrated together to achieve a use case difficult. Therefore activity diagrams, which are functional models, are used to make the sequencing of activities in a use case description more precise. Use case activity diagrams are built as described in (Gomaa & Olimpiew 2010; Olimpiew 2008). Using this approach, SPL variability is captured in these diagrams in two ways. First, the flow in activity can be associated with the SPL features when a feature influences the sequence of events. This is called a feature based condition. It is modeled as a decision node with the feature conditions. When traversing the activity diagram for a SPL member, the path taken corresponds to the SPL member's feature selection. To differentiate feature conditions from execution conditions, quotations are used around the feature. Second,

steps with variability, which are denoted using the <<adaptable>> stereotype, are successively refined in separate sub-activity diagrams until all individual variations are modeled. However, there is a subtle difference when managing variability at a higher level of granularity. The approach in this paper will only refine adaptable steps when a step is impacted by a small number of variation points. If a significant amount of variability in an adaptable step exists, as seen by the multitude of variation points impacting the step, then sub-activity diagrams for all possible combinations will not be created. Instead the all the potential individual interactions will remain abstracted at this higher level of granularity. This strategy makes modeling activity diagrams more scalable.

Use case activity modeling is demonstrated using a simplified version of the FSW SPL's Execute Commands use case. This use case involves executing commands from the ground station to ensure the spacecraft is not put into an unsafe state and the actions taken are appropriate for the spacecraft's mode. The simplified activity diagram for this use case is depicted in Figure 5-12. The sequence in this use case activity diagram is first impacted by the Command Execution feature condition. This feature based condition is based on the Command Execution pattern specific feature group, which contains three alternative features. If a SPL member realizes the Low Volume Command Execution pattern specific feature then the path through the use case for this SPL will follow the path marked with the CommandExecution = "LowVolume." Thus this SPL would take

steps one and two through the activity diagram. In step one, the FSW updates the spacecraft's time. This activity is performed because the current time is needed to determine a command sequence where all command deadlines can be met. In step two, the FSW must validate, determine a command sequence, and execute a low volume of commands that is appropriate for the spacecraft's mode. For instance, if the FSW receives a collect payload data command in launch mode, then this command should be rejected. This is because launch mode only involves initializing the spacecraft and getting the spacecraft into orbit. Payload data should not be collected until the spacecraft is in its desired orbit and in normal mode. Step two is influenced by several variation points, including the Modes and the Spacecraft IO device. The Spacecraft IO device variation points include Antennas, Antenna Gimbals, Memory Storage Devices, Power Appendages, Power Devices, Attitude Control Devices, Attitude Determination Devices, Payload Devices, Thrusters, Heaters, Louvers, and Temperature Sensors.

Next, if a SPL member realizes the High Volume Command Execution pattern specific feature then the path through the use case for this SPL will follow the path marked with the CommandExecution = "HighVolume." This SPL member will take steps one and three through the activity diagram as seen in Figure 5-12. Steps one and three are similar to steps one and two, except a high volume of commands is required to be executed which will require interacting with a larger amount of IO devices.

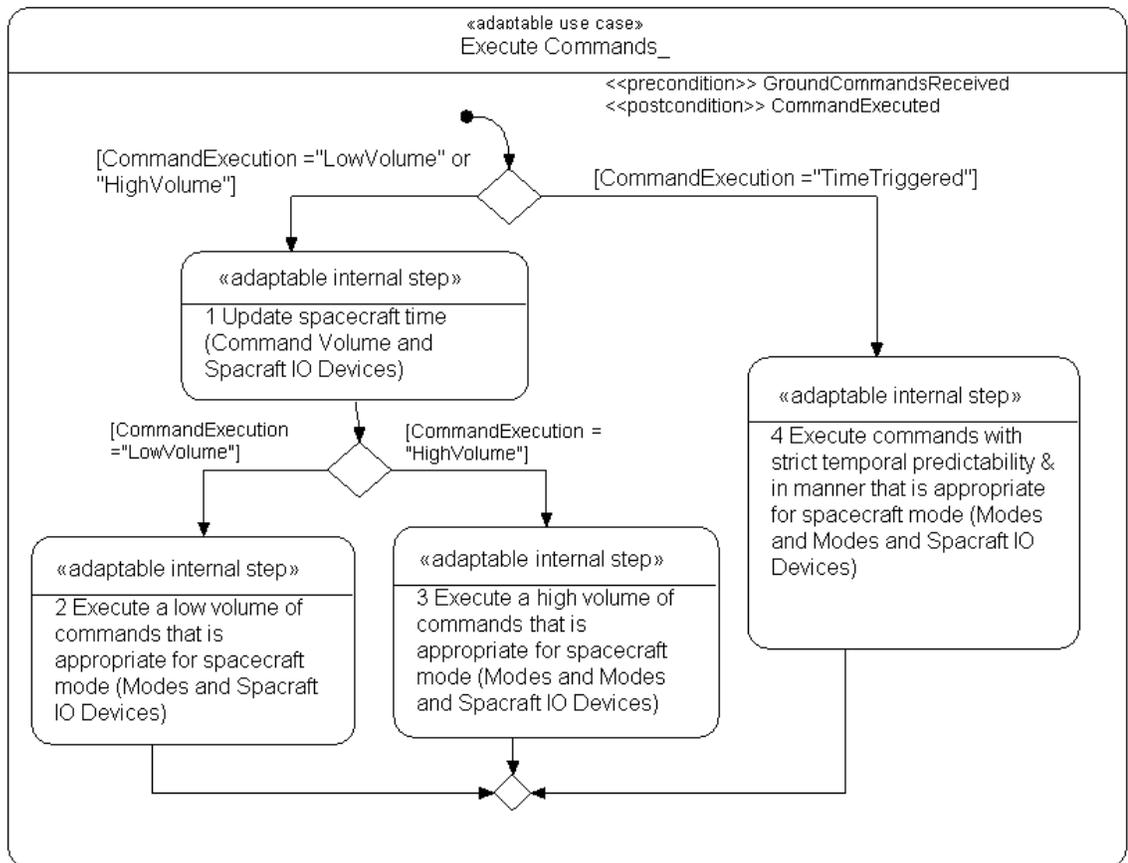


Figure 5-12 Simplified Execute Commands use case activity model

Finally, the CommandExecution = “TimeTriggered” path is taken when SPL members realized the Time Triggered Command Execution pattern specific feature. Then step four is used as seen in Figure 5-12. In this step, commands must be executed with strict temporal predictability. Strict temporal predictability is best achieved using a time triggered architecture, which time is provided and components perform actions during

predefined time intervals. Therefore the FSW does not manage time so step one is not included on this path.

The use case variation points that impact the adaptable steps are listed in parenthesis after the step's description, as seen in Figure 5-12. To fully specify the use case scenario, each adaptable step should have a separate sub-activity diagram illustrating the different variants and non-adaptable steps. However, adaptable steps 1-4, have a significant amount of variability as seen by the multitude of variation points impacting the steps. Therefore sub-activity diagrams for all possible combinations of these adaptable steps will not be created. Instead, all the potential individual interactions will remain at this higher level of granularity.

For each SPL use case scenario, an SPL interaction overview diagram is created based on the use case activity diagram. An interaction overview diagram is created by using the same control flow as the use case activity diagrams. However, rather than representing each step as an activity, it will show a reference to the interaction diagram from the SPL executable design pattern that supports that step or a reference to another interaction overview diagram where the step is refined. On feature based condition paths, the design pattern use to achieve one or more of the steps along the path can be determined from the feature to design pattern mapping.

After an interaction overview diagram is created the design pattern interconnections are determined. When two design patterns appear sequentially in the interaction overview diagram this implies they must communicate with each other and must be interconnected. Therefore these design patterns are interconnected by adding ports and connectors between the components in the design patterns that perform the communication.

Finally after the design patterns are interconnected, this information is used to help to identify where the design patterns and their components fit into the overall software architecture. This is accomplished by analyzing the communication within a design pattern and the design pattern interconnections. For instance if a component from one design pattern utilizes a service from a component in another design pattern, then in a layered architecture the component using the service is in a higher layer than the component providing the service.

The design pattern integration process is illustrated using the simplified FSW SPL Execute Commands use case. An interaction overview diagram is created based on the FSW SPL Execute Commands use case activity diagram in Figure 5-12. This is accomplished by using the same control flow in the use case activity diagram. Since the Execute Commands use case activity diagram begins with a feature based decision, the interaction overview diagram will also begin with this same decision point, as seen in Figure 5-13. Then the steps along the path are converted to reference the interaction

diagram from the SPL executable design pattern that supports that step. Each of the alternative steps in the activity diagram in Figure 5-12 are updated to their supporting design patterns using the feature to design pattern mapping, as seen in Figure 5-13. For example, on the far left the path's feature condition is `CommandExecution = "LowVolume."` Using the feature to design pattern mapping, the Low Volume Command Execution feature is mapped to the FSW Centralized Control design pattern. Then based on the dependencies in the feature model, it can be determined that the FSW Centralized Control design pattern has a dependency on the Spacecraft Clock pattern specific feature. Therefore the design pattern associated with the the Spacecraft Clock pattern specific feature will also be used in this scenario.

Step one involves sending a time update. This step is supported by the Spacecraft Clock pattern specific feature, which is mapped to the FSW Spacecraft Clock Multicast executable design pattern. Therefore this step is replaced with a reference to the FSW Spacecraft Clock Multicast's interaction diagram. Step two along this path involves executing a small number of commands. This feature is supported by the Low Volume Command Execution feature, which is mapped to the FSW Centralized Control executable design pattern. Therefore this step is replaced with a reference to the FSW Centralized Control's interaction diagram, as seen in Figure 5-13.

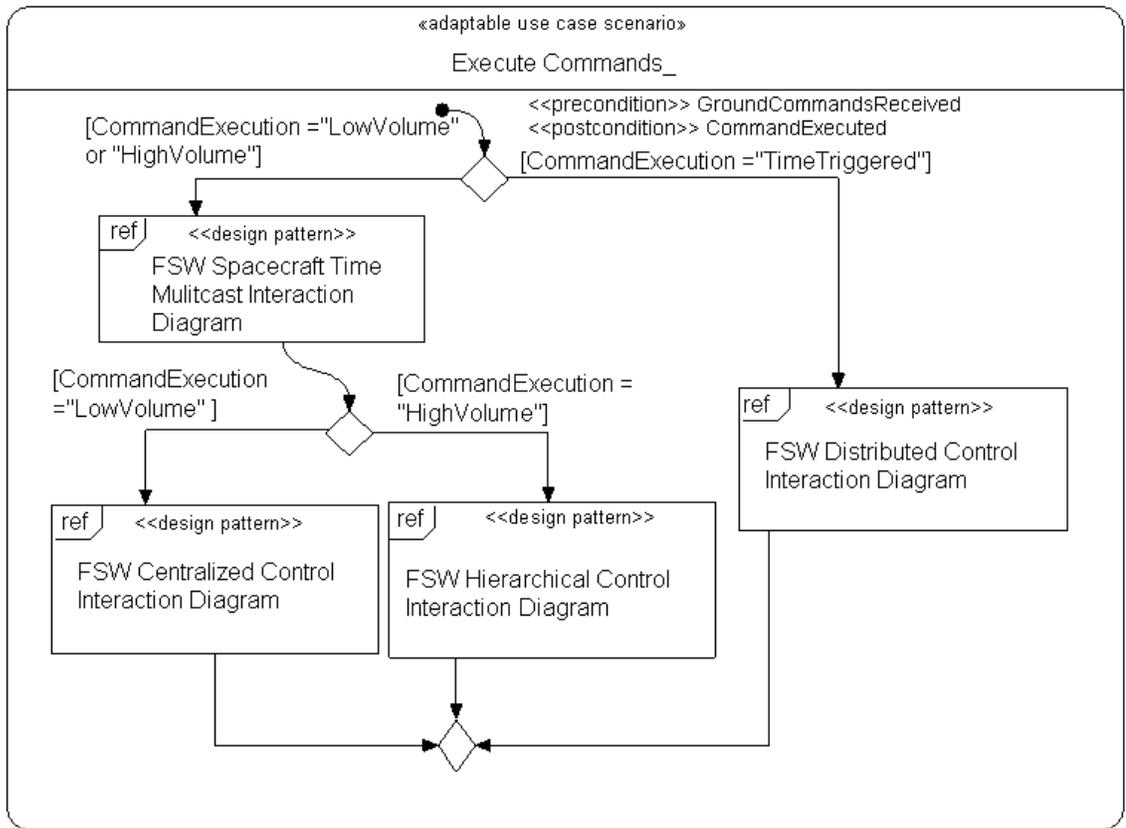


Figure 5-13 Execute Commands interaction overview diagram

After the interaction overview diagram is created, it is then analyzed to identify where design patterns are interconnected. If there are two sequence design patterns along an execution path, then these design patterns must be interconnected. In the simplified Command Execution use case interaction diagram, the FSW Spacecraft Clock Multicast design pattern interconnects with the FSW Centralized Control and FSW Hierarchical Control design patterns.

Once the design patterns that interconnect are identified, the specific components within these patterns that communicate are identified and interconnected with ports and connectors. First, consider the case where the FSW Spacecraft Clock Multicast executable design pattern interconnects with the FSW Centralized Control executable design pattern. The Spacecraft_Clock_Multicast component is responsible for updating the time and sending time updates to a predetermined set of consumers. In order to properly determine the command execution sequence the CDH_Centralized_Controller, from the FSW Centralized Control executable design pattern, needs to know the current time. Additionally, the input, output, and IO components from the FSW Centralized Control executable design pattern also need time updates because they need to time-tag their data. Therefore, ports and connectors are created between the Spacecraft_Clock_Multicast component and CDH_Centralized_Controller, input, output, and IO components. For example, the interconnection between the Spacecraft_Clock_Multicast component and CDH_Centralized_Controller is shown in Figure 5-14. Since the Spacecraft_Clock_Multicast component is sending input to the CDH_Centralized_Controller, a required port called RController is added to the Spacecraft_Clock_Multicast component. It is connected to the CDH_Centralized_Controller's provided port called PController.



Figure 5-14 Subset FSW Centralized Control and FSW Spacecraft Clock Multicast interconnection

Next, the interconnection between the FSW Spacecraft Clock Multicast executable design pattern and the FSW Hierarchical Control executable design pattern is examined. In the FSW Spacecraft Clock Multicast executable design pattern, the Spacecraft_Clock_Multicast component is responsible for updating the time and sending time updates to a predetermined set of consumers. In the FSW Hierarchical Control executable design pattern, CDH_Coordinator needs a time update in order to determine the command execution sequence. Additionally, all the localized controllers need the current time in order to know when to execute a command. Additionally, the input, output, and IO components from the FSW Centralized Control executable design pattern also need time updates because they need to time-tag their data. Thus port and connectors are created between the Spacecraft_Clock_Multicast component and the CDH_Coordinator, all the localized controller components, all the input, output, and IO components.

5.5.2 Application Level Design Pattern Interconnection

During the application engineering process, the design pattern interconnection is updated based on the application's selected features. Any design patterns that are not used based on the selected features should be removed from the interaction overview diagram.

Additionally, the associated ports and connectors should also be removed.

For example, consider the simplified Execute Commands interaction overview diagram in Figure 5-13. SNOE, the small spacecraft used in section 5.4.3, will only utilize the first path simplified Execute Commands activity diagram since SNOE selected to use the Low Volume Command Execution feature. Therefore the other two paths are removed for SNOE. Additionally, the ports and connectors between the components in the FSW Spacecraft Clock executable design pattern and the FSW Hierarchical Control executable design pattern are removed.

6 AN APPROACH TO BUILDING DOMAIN SPECIFIC SOFTWARE PRODUCT LINE ARCHITECTURES FROM DESIGN PATTERNS

6.1 Introduction

This chapter describes an approach for building domain specific software architectures from software architectural design patterns. To accomplish this goal, the research approach is applied to the unmanned spacecraft flight software (FSW) domain. First, this chapter provides an overview of the FSW domain and illustrates why this is an ideal domain to apply this research. Then, this chapter describes how to construct FSW command and data handling software architectures from the architectural and executable design patterns following the research approach. The command and data handling subsystem represents sufficient complexity and diversity of component interactions to adequately illustrate the process of building flight software from software architectural design patterns. This chapter focuses on the SPL engineering while the emphasis in Chapter 7 and Appendices C and D focus on application engineering. Finally, this chapter illustrates the scalability benefits of the approach by comparing it with a traditional SPL approach using the FSW SPL.

6.2 Approach to Build Domain Specific Software Architectures from Design Patterns Overview

This subsection describes how these novel contributions from Chapter 5 are used together in an overall approach to build domain specific DRE software architectures from software architectural design patterns. This approach utilizes SPL concepts and processes to build domain specific software architectures and to capture domain specific rules for how the design patterns can and cannot be composed to form the foundation for software architectures. SPL concepts are a good match for this approach because they can be leveraged to capture the variability across a domain. The SPL concepts used in this approach are based on the Product Line UML-Based Software Engineering (PLUS) method (Gomaa 2005).

A high level view of the proposed approach is depicted below in Figure 6-1, where processes are rectangles and repositories are cylinders. The overall approach is an evolutionary SPL engineering process, where SPLs evolve through several iterations. Within this process, there are two major processes which are software product line engineering and software application engineering. These phases are described below in more detail.

First, the SPL engineering process involves identifying and analyzing the commonality and variability across the SPL. The inputs to this phase are the SPL requirements and the

DRE architectural and executable design patterns, previously described in Chapter 4. This process has three phases, which include the requirements modeling, analysis modeling, and design modeling phases. During these phases, the SPL architecture is defined and developed. As part of this process, software design patterns that can be used in the software architecture are identified and mapped back to SPL features using the feature to design pattern mapping. Additionally, the DRE executable and architectural design patterns are customized to become SPL level architectural and executable design patterns. The SPL engineering process is described in more detail in the remaining subsections.

The second major process is the application engineering process, which involves deriving an individual SPL member's software architecture from the SPL assets. This process also has three phases, which are the requirements, analysis, and design modeling. The artifacts from these phases are derived from the SPL assets by selecting the kernel and variable assets that meet the SPL member's requirements. The entire application engineering process is described in more detail in Chapter 7.

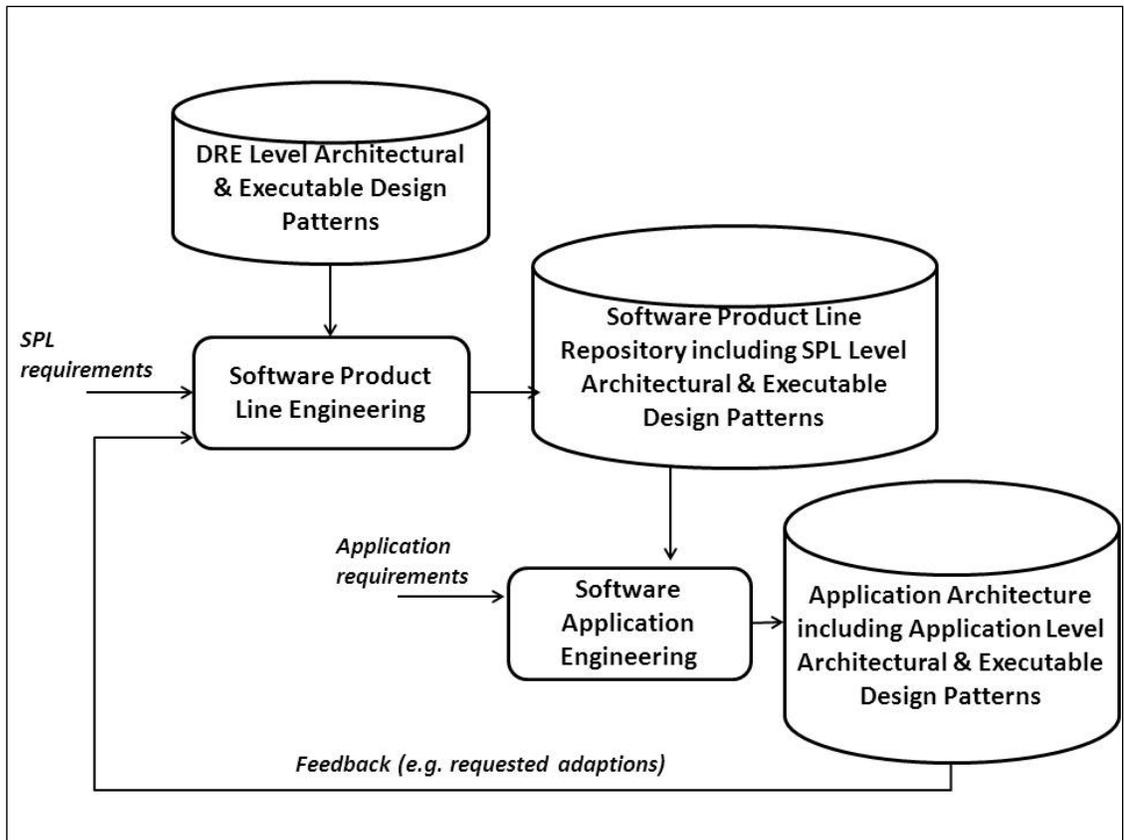


Figure 6-1 High Level View of Approach

6.3 Unmanned Spacecraft Flight Software Domain Overview

The unmanned spacecraft flight software (FSW) domain involves using software in unmanned spacecraft to achieve a variety of missions. Spacecraft missions span scientific missions, such as the Student Nitric Oxide Explorer (SNOE) that measures nitric oxide and its variability in the Earth’s atmosphere (Laboratory For Atmospheric and Space Physics at the University of Colorado at Boulder n.d.), to navigational

missions, such as the Global Positioning System (GPS) that provides reliable location information anywhere on Earth (Strom 2002).

The role of FSW has evolved from performing simple operations to now controlling the majority of the spacecraft and its payloads. This increased responsibility has led to additional FSW complexity and a greater number of software related anomalies. It is reported that “in the period from 1998 to 2000, nearly half of all observed spacecraft anomalies were related to software” (Hecht & Buettner 2005). These anomalies can lead to mission disruption or even mission failure. In the FSW domain, these types of anomalies can be devastating for a mission especially given the significant length of time that is required to build a new or replacement spacecraft.

The FSW industry is in need of sound software engineering practices to help them better manage the complexities of FSW and avoid onboard anomalies. In fact, the National Aeronautics and Space Administration (NASA) commissioned a study on FSW that examined flight software complexity and provided a series of recommendations to better manage the associated challenge. One recommendation from this report included the need to perform early analysis and architecting on FSW acquisition efforts (Dvorak (editor) 2009).

Since the FSW industry has a strong need to apply early analysis and architecting on FSW, it is an ideal domain to apply the research proposed in this PhD dissertation. The design pattern specific SPL approach described in Chapter 5 is a great match for the FSW domain. This is because it provides a systematic approach for designing FSW that utilizes best practice software design patterns. Additionally, the proposed approach lends itself to performing design time functional validation. This design time validation can help identify issues early in the software lifecycle when changes are less costly.

6.4 Unmanned Spacecraft Flight Software Product Line Problem Description

The initial iteration of FSW SPL consists of controlling an unmanned spacecraft of 100kg or above to achieve a space mission. The FSW SPL does not include spacecrafts under 100kg such as Pico spacecrafts and manned spacecraft such as space shuttles. These types of spacecraft can be analyzed for applicability in the FSW SPL in future iterations.

Capabilities of FSW systems vary widely. For example, a spacecraft may rely extensively on the ground station to control the spacecraft and may not require a significant amount of hardware to perform its mission. Another spacecraft may utilize onboard autonomy to control the spacecraft and may have a significant amount of hardware to perform its mission. The functionality of other spacecraft can vary anywhere in between these extremes. The major spacecraft design decisions that influence the FSW include:

- Spacecraft versus ground capability tradeoff – influences how much capability will be housed onboard versus performed at the ground station
- Payload devices– influences the amount of payload command and control the FSW needs to effectively operate the payload devices
- Attitude control type –the selected stabilization technique effects control algorithms, hardware, and commanding
- Orbit type – influences the amount of guidance and control and propulsion required
- Thermal control type – influences how much of a role, if any, the FSW plays in thermal control
- Mission criticality – effects the amount of hardware redundancy and FSW reliability

The FSW SPL is developed using a reverse engineering SPL approach, where existing systems are analyzed. Since there is a wide variety of FSW systems, systems on the far ends of the FSW SPL scope will be analyzed. These systems include a small spacecraft that relies heavily on the ground commanding to manage spacecraft operations and a large spacecraft in deep space that performs the majority of spacecraft's operations autonomously. Additionally, to ensure the FSW SPL is robust enough to handle future spacecraft technologies, the FSW SPL will also be engineered using a spacecraft system with a time-triggered architecture (Walko 2009; TTech n.d.; Gwaltney & Briscoe 2006; Gwaltney & Briscoe 2005).

6.5 Software Product Line Engineering

The SPL engineering process involves identifying and analyzing the commonality and variability across the SPL. This process leverages SPL stereotypes defined in (Gomaa 2005), where assets that are common to all SPL members are marked as <<kernel>> and assets that only some SPL members provide are denoted using the <<optional>> and <<alternative>> stereotypes. Additionally, if variability exists within the asset then <<param-vp>> is added to the stereotype. Detailed descriptions of the SPL engineering phases are described in the subsequent subsections.

6.6 Software Product Line Requirements Modeling Phase

The first phase in the SPL engineering process is the requirements modeling phase. The purpose of this phase is to scope the SPL and to define the SPL requirements. The artifacts from this phase are the use case model, feature model, and use case activity model. The process to develop the feature model and use case activity model is illustrated using the FSW SPL. Additional information about the use case model is found in Appendix B.

6.6.1 Software Product Line Feature Modeling

After the use cases are identified, the feature model for the SPL can be derived from these use cases. The most important requirements modeling artifact is the feature model. The feature model depicts all the required and variable features that SPL members

provide (Gomaa 2005). However, as discussed in Chapter 5, feature modeling has differences when mapping features to design patterns. For each pattern, there needs to be a primary feature to differentiate this feature from others, as well as other features to address variability within the pattern. Coarse grained features, called pattern specific features, capture the variability in the SPL's required functionality. Fine grained features, called pattern variability features, influence the variability within pattern specific features. This type of feature relationship is modeled as an association using the <<requires>> stereotype. The feature model for the FSW SPL command and data handling subsystem is derived from the SPL use case model. The details of this process are described below in more detail.

The processes for building a feature model is illustrated using the FSW SPL command and data handling (C&DH) subsystem. A high level feature model for the C&DH is created that depicts the dependencies just between the features and feature groups. This model for the C&DH is depicted in Figure 6-2. Each of the features and feature group in this model is described below

- **Memory Storage Device Type pattern variability feature group.** This group captures the variability in the different types of onboard memory that a spacecraft utilizes, which are derived from Memory Storage Device variation points in the Command Execution, Collect and Store Spacecraft Data and Perform Fault Detection use cases. This feature is related to variability in the Memory Storage Device Fault

Detection, Command Execution, Housekeeping Data Collection, and Telemetry Storage and Retrieval pattern specific features because it influences the type of fault detection performed, how the memory storage device is commanded, the type of housekeeping data collected, and the amount memory storage device hardware that is used to store/retrieve data, respectively.

- **Memory Storage Device Fault Detection pattern specific feature.** This kernel pattern specific feature is derived from the Perform Fault Response use case, where the FSW periodically checks the memory storage device for faults. This feature is impacted by the variability in the Memory Storage Device Type feature group.
- **Telemetry Storage and Retrieval pattern specific feature group.** This group captures alternative ways the data is stored and retrieved from the Collect and Store Spacecraft Data use case and its Data Volume variation point.
- **Telemetry Formation pattern specific feature group.** This group captures the variability in how the spacecraft formats raw spacecraft data into telemetry packets. This feature group is derived from the Collect and Store Spacecraft Data use case and its variation points.
- **Telemetry Formation Reliability pattern specific feature group.** This group captures what checks if any the FSW needs to perform during the telemetry formation process. It is derived from the Perform Fault Management use case. This pattern specific feature group requires the Telemetry Formation pattern specific feature group because it provides a layer of fault detection to the telemetry formation process.

Therefore it cannot function properly without the Telemetry Formation pattern specific feature group.

- **Spacecraft Clock pattern specific feature.** This pattern specific feature is derived from the Execute Commands use case's Spacecraft Clock variation point and it captures whether or not the FSW is required to maintain the spacecraft clock.
- **Command Execution pattern specific feature group.** This pattern specific feature group captures the variability in how the spacecraft will execute the commands from the ground station. It is derived from the Execute Commands use case's Command Volume, Strict Temporal Predictability, and Modes variation points.
- **Housekeeping Data Checks optional pattern specific feature.** This feature is derived from the Perform Fault Response use case and captures whether or not the FSW is required to monitor and check the housekeeping data for potential problems. The Housekeeping Data Checks feature depends on the Housekeeping Data Collection feature group. This is because in order to monitor the housekeeping data, the housekeeping data must first be collected.
- **Housekeeping Data Collection pattern specific feature group.** This feature group captures the various optional ways housekeeping data collection is performed in the FSW SPL. This group is derived from the Collect and Store Space Data use case and its Collection Trigger variation point.

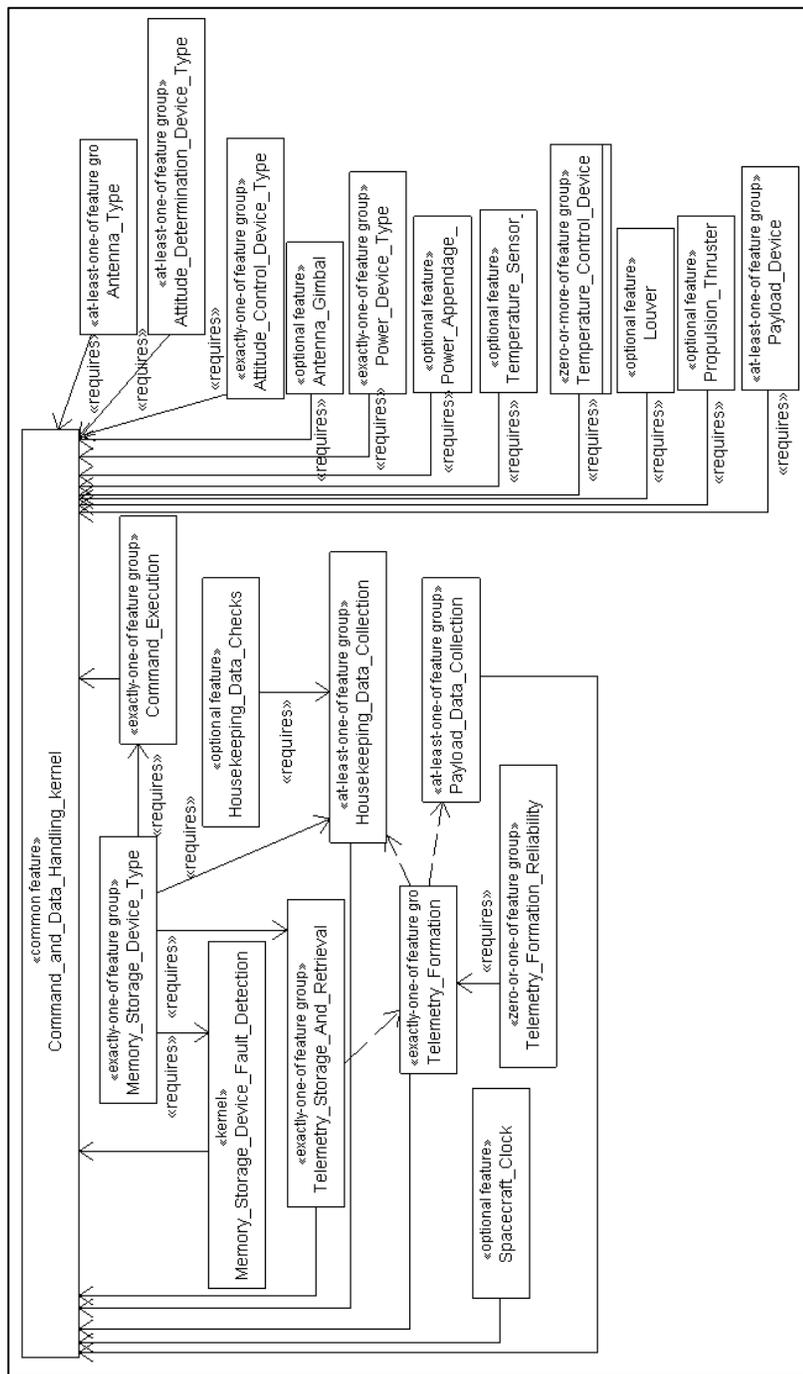


Figure 6-2 High level FSW SPL C&DH feature model with dependencies

- **Payload Data Collection pattern specific feature group.** This pattern specific feature group is derived from the Collect and Store Spacecraft Data use case and its Collection Trigger variation point. This feature captures the various optional ways payload data collection is performed in the FSW SPL.
- **Other subsystem pattern variability features and feature groups.** This set of features is depicted on far left column in Figure 6-2. These pattern variability features and features groups belong to other subsystems. However, they are shown in the C&DH feature model because they relate to variability in the C&DH subsystem. These pattern variability features will influence the type and amount of commanding that is required by the Command Execution pattern specific feature group. They influence the type and amount of housekeeping data that is collected in the Housekeeping Data Collection pattern specific feature group. Finally, they influence the type and amount of data collected in the Payload Data Collection pattern specific feature group.

Next, a feature dependency model is created, which illustrates the optional and alternative features in the feature groups and all the features. The C&DH feature dependency diagram is depicted in Figure 6-3 and Figure 6-4, however some of the relationships previously modeled are not depicted to keep the diagrams readable. A description of these features depicted in Figure 6-3 is described below:

- **Command Execution pattern specific feature group.** This group contains a set of alternative pattern specific features are derived from the Execute Commands use case.

The first alternative pattern specific feature is the High Volume Command Execution which is derived when the Command Volume variation point is set to high, Strict Temporal Predictability is set to false, and Command Flexibility is set to false. Next, there is the Low Volume Command Execution pattern specific feature which is derived when Command Volume is set to low, Strict Temporal Predictability is set to false, and Command Flexibility is set to false. The third alternative pattern specific feature in this group is the Temporal Predictable Command Execution which is used when Command Volume is either high or low, Strict Temporal Predictability is set to true, and Command Flexibility is set to false. Next is the High Volume Command Execution with Command Flexibility which is derived when the Command Volume is set to high, Strict Temporal Predictability is set to false, and the Command Flexibility is set to true. The fifth alternative pattern specific feature is the Low Volume Command Execution with Command Flexibility feature. This feature is derived when the Command Volume is set to low, Strict Temporal Predictability is set to false, and the Command Flexibility is set to true. Finally the Temporal Predictable Command Execution with Command Flexibility pattern specific feature is derived when the Command Volume is either high or low, Strict Temporal Predictability is set to true, and Command Flexibility is set to true. The High Volume Command Execution, Low Volume Command Execution, High Volume Command Execution with Command Flexibility, and Low Volume Command Execution with Command Flexibility alternative pattern specific features all have a dependency on the Spacecraft Clock

pattern specific feature because they utilize an event driven platform where global time must be provided by the software.

- **Telemetry Formation pattern specific feature group.** This feature group contains four alternative pattern specific features which are derived from the Collect & Store Data use case. First, the High Volume Formation is derived when the Data Volume variation point is set to high and the Algorithm Flexibility variation point is set to false. Next, the Flexible High Volume Formation alternative pattern specific feature is derived when the Data Volume variation point is set to high and the Algorithm Flexibility variation point is set to true. The third alternative pattern specific feature in this group is Low Volume Formation. This pattern specific feature is derived when the Data Volume variation point is set to low and the Algorithm Flexibility variation point is set to false. Finally, the Flexible Low Volume Formation pattern specific feature is the last feature in this group. It is derived when the Data Volume variation point is set to low and the Algorithm Flexibility variation point is set to true.
- **Telemetry Formation Reliability pattern specific feature group.** This group contains a set of two pattern specific alternatives. First, the Extensive Telemetry Formation Checks pattern specific feature is derived from the Perform Fault Management use case where the FSW is required to perform multiple checks on the data during the telemetry formation process to detect problems with the data. The second alternative pattern specific feature is Quick Telemetry Formation Check feature, which is derived from the Perform Fault Management use case where the

FSW is required to ensure that the overall process is performing within an acceptable range by performing a quick check on the results.

- **Payload Data Collection pattern specific feature group.** This group captures the different the strategies for collecting the payload data. This is an at-least-one-of feature group meaning that all SPL members must minimally use one payload data collection strategy. However, multiple strategies can be used if different payload devices need different collection strategies. This pattern specific feature group contains three optional features as derived from the Collect & Store Data use case. The first is the Ground Driven Payload Data Collection pattern specific feature. This pattern specific feature is used when the ground station controls when and what type of payload data is collected. The next optional pattern specific feature is Event Driven Payload Data Collection, which involves having the payload data sent out to a predetermined group of receivers during some event, such as a time event or buffer full event. Finally the last optional pattern specific feature in this group is the Event Driven Payload Data with Dynamic Collection feature. This feature involves sending payload to a dynamic set of receivers during some event, such as a time event or buffer full event.

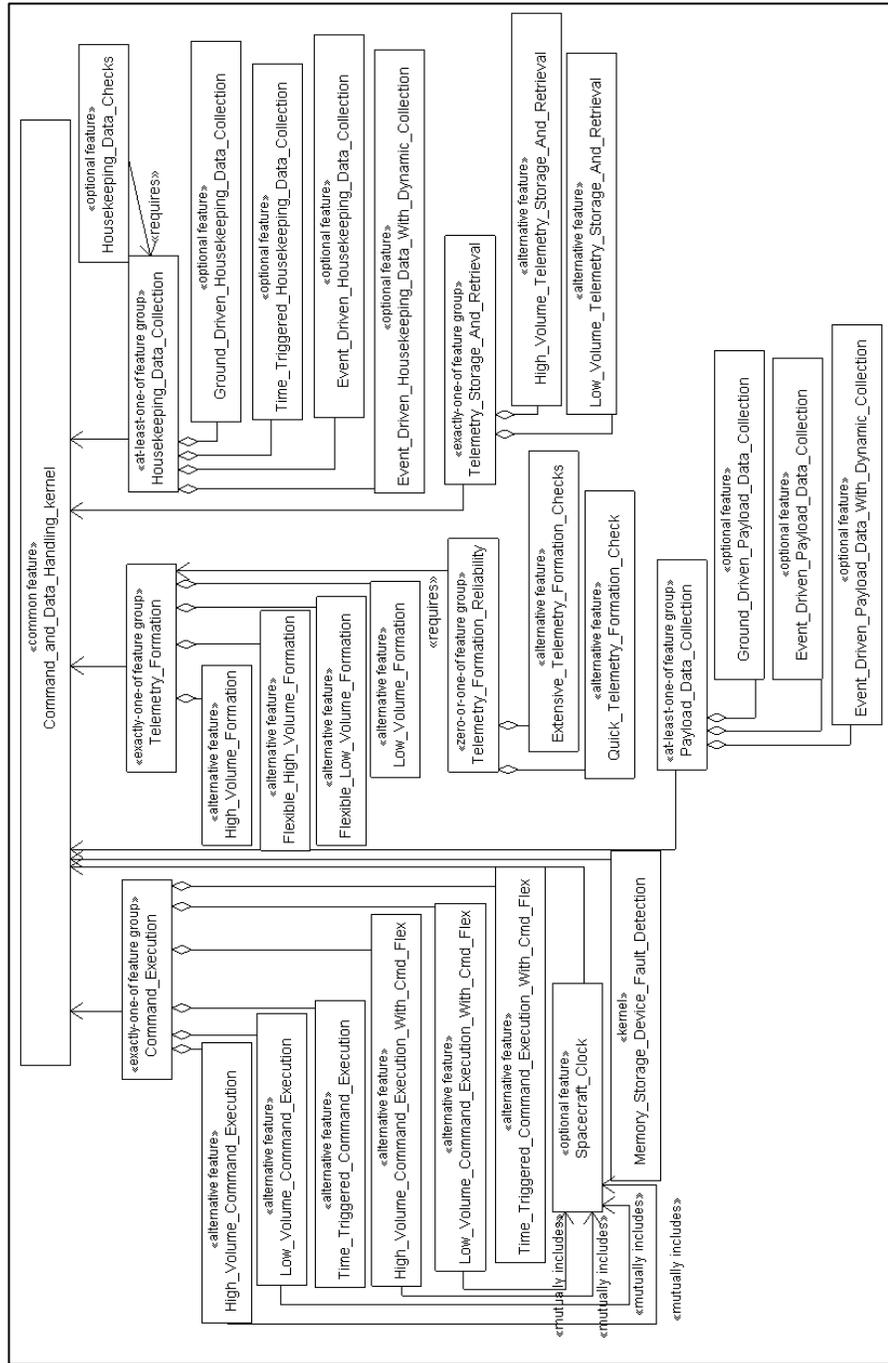


Figure 6-3 C&DH Feature Model

- **Housekeeping Data Collection pattern specific feature group.** This is an at-least-one-of feature group meaning that all SPL members must minimally use one housekeeping data collection strategy. However, multiple strategies can be used if different hardware devices require different collection strategies. This feature group contains four optional pattern specific features. The Ground Driven Housekeeping Data Collection, Event Driven Housekeeping Data Collection, and Event Driven Housekeeping Data with Dynamic Collection pattern specific features have the same collection strategies used in the Payload Data Collection feature group, except housekeeping data is collected instead of payload data. The last optional pattern specific feature in this ground is the Time-Triggered Housekeeping Data Pushing, which contains the collection strategy that should be used in a time-triggered architecture where housekeeping data is broadcast to all nodes at predefined time intervals.
- **Telemetry Storage/Retrieval pattern specific feature group.** This is an exactly-one-of feature group therefore all SPL members must provide one feature from this group. This feature group provides the structure and strategy for storing telemetry. The first alternative feature in this group is the High Volume Telemetry Storage and Retrieval pattern specific feature. This pattern specific feature is selected when the spacecraft is required to store a high volume of data. The other alternative pattern specific feature is Low Volume Telemetry Storage and Retrieval, which is selected when the spacecraft is required to store a low volume of data.

Next Figure 6-4 illustrates just the pattern variability features without their dependencies on pattern specific features. Many of these pattern variability features are associated with other subsystems, however they are include and described since they influence the C&DH subsystem. The pattern variability features in Figure 6-4 are described below in more detail:

- **Memory Storage Device Type pattern variability feature group.** This group which captures the alternative types of onboard memory storage devices. The alternative devices are derived from the Command Execution, Collect and Store Spacecraft Data, and Perform Fault Management’s Memory Storage Device variation point. There are several options for memory storage devices. Tape recorders, Erasable Programmable Read-Only Memory (EEPROM), FLASH Memory, and Random Access Memory (RAM) (NASA Jet Propulsion Laboratory 2011). Thus, all these devices are modeled as alternative features.
- **Antenna Type pattern variability feature group.** This group which captures the alternative types of antennas. The alternative devices are derived from the Command Execution, Collect Housekeeping Data, Uplink/Downlink Data use cases’ Antenna Type and Antenna Number variation points. For example, the Small Spacecraft utilizes a low gain antenna. Alternatively, the Large Spacecraft requires a high gain antenna, medium gain antenna, and low gain antenna. Therefore the optional features include the high-gain antenna (HGA), medium gain antenna (MGA) and the low-gain antenna (LGA).

- **Antenna Gimbal pattern variability feature.** The Antenna Gimbal pattern variability feature shows whether or not gimbals are used to make the antenna movable. This feature is derived from the Command Execution, Collect Housekeeping Data, Uplink/Downlink Data use cases' Antenna Type variation point. It has a dependency on the Antenna Type pattern variability feature group because it requires an antenna in order to make it moveable.
- **Power Device Type pattern variability feature group.** This group shows the different alternative power device types that can be selected for a spacecraft. The types of power devices are derived from the Command Execution, Collect and Store Spacecraft Data, and Control Power use cases' Power Device variation point. While both the Large and Small Spacecraft use photovoltaic devices to generate power, radioisotope thermoelectric generators are also commonly used to generate power (NASA Jet Propulsion Laboratory 2011). Therefore these two alternatives features are modeled.
- **Power Appendage pattern variability feature.** This feature captures whether or not the FSW requires the ability to maneuverer the power devices. Thus there is a dependency on the Power Device Type pattern variability feature group. The Power Appendage pattern variability feature is derived from the derived from the Command Execution, Collect and Store Spacecraft Data, and Control Power use cases' Power Appendage variation point

- **Temperature Control Device pattern variability feature group.** This pattern variability group captures the variability in the types of temperature control devices. For instance, Large Spacecraft system utilizes heaters. However, louvers are also used on spacecraft to control temperature (NASA Jet Propulsion Laboratory 2011). Therefore these two devices are modeled as optional features. This pattern variability feature is derived from the Command Execution, Collect and Store Spacecraft Data, and Maintain Temperature use case's Temperature Control variation point.
- **Temperature Sensor Type pattern variability feature group.** This pattern variability group captures the variability in the types of temperature sensors. For instance, Large Spacecraft system utilizes thermometers. However, thermistors are also used on spacecraft to measure temperature (NASA Jet Propulsion Laboratory 2011). Therefore these two devices are modeled as alternative features. This pattern variability feature is derived from the Command Execution, Collect and Store Spacecraft Data, and Maintain Temperature use case's Temperature Sensor variation point.
- **Propulsion Thruster pattern variability feature.** This feature captures whether or not the FSW is required to interface and control the propulsion thruster. It is derived from the Command Execution and the Collect and Store Spacecraft Data use case's Propulsion Thruster variation point, as well as the Maintain Flight Path and Maintain Orbit use cases.

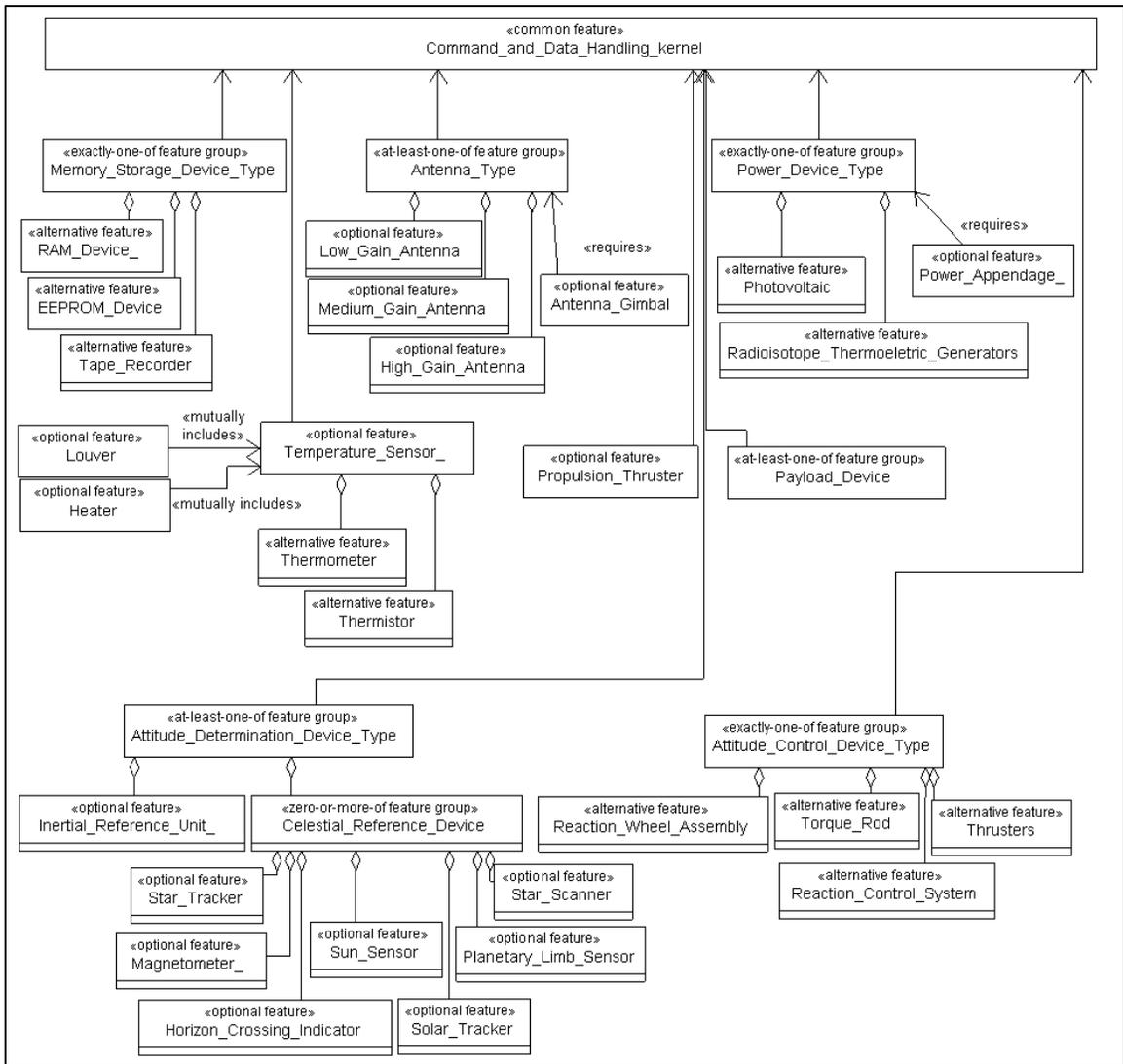


Figure 6-4 C&DH feature dependency model continued

- Payload Device pattern variability feature group.** This feature group captures the variability in the payload devices. Payloads are unique to each spacecraft mission; therefore the specific alternative features cannot be modeled in the FSW SPL. This

- feature group is derived from the Command Execution, Collect and Store Spacecraft Data, and Control Payload use case's Payload Device variation point.
- **Attitude Determination Device Type pattern variability feature group.** This group captures the possible for attitude determination devices. For example the Large Spacecraft uses both an inertial reference unit (IRUs) and celestial reference devices, while the Small Spacecraft only uses celestial reference devices. There are several types of celestial reference devices. For example, the Small Spacecraft utilizes a magnetometer and a horizon crossing indicator and the Large Spacecraft uses a star tracker and sun sensor. Other commonly used attitude determination devices including star scanner, solar trackers and planetary limb sensors (NASA Jet Propulsion Laboratory 2011). Therefore these are all modeled as optional devices. This feature is derived from the Command Execution, Collect and Store Spacecraft Data, and Control Attitude use cases' Attitude Determine Device variation point.
 - **Attitude Control Device Type pattern variability feature group.** This group captures the variability in the attitude control devices. For instance, the Small Spacecraft uses a torque rod while the Large Spacecraft uses a reaction control system. Other possible options include reaction wheel assembly (RWA) and small thrusters, but not both. Since only one of the attitude control devices can be selected per spacecraft they are modeled as alternative features. This feature is derived from the Command Execution, Collect and Store Spacecraft Data, and Control Attitude use cases' Attitude Control Device variation point.

The final summary of the feature/use case dependencies of the FSW SPL command and data handling subsystem is depicted in Table 6-1.

Table 6-1 Feature/Use case dependencies of FSW SPL C&DH subsystem

Feature Name	Feature Category	Use Case Name	Use Case Cat./vp	Variation Point Name
High Volume Command Execution	alternative	Execute Commands	vp	Command Volume, Strict Temporal Predictability, Modes, Spacecraft Clock, Command Flexibility, Antenna Number, Antenna Type, & Memory Storage Devices Power Appendage & Power Devices Attitude Control Devices & Attitude Determination Devices Payload Devices
		Control Power	vp	
		Control Attitude	vp	
		Operate Payload Control Orbit Manage Temperature	vp	
Low Volume Command Execution	alternative	Execute Commands	vp	Command Volume, Strict Temporal Predictability, Modes, Spacecraft Clock, Command Flexibility, Attitude Control Devices, Attitude Determination Devices, Payload Devices, Memory Storage Devices, Antenna Number, Antenna Type, Power Devices, Temp. Control, Temp. Sensors, Thrusters, and Power Appendage
		Execute Commands	vp	
		Execute Commands	vp	
		Execute Commands	vp	
Temporal Predictable Command Execution	alternative	Execute Commands	vp	Command Volume, Strict Temporal Predictability, Modes, Spacecraft Clock, Command Flexibility, Antenna Number, Antenna Type, & Memory Storage Devices Power Appendage & Power Devices Attitude Control Devices & Attitude Determination Devices Payload Devices
		Control Power	vp	
		Control Attitude	vp	
		Operate Payload Control Orbit Manage Temperature	vp	
High Volume Command Execution with Command Flexibility	alternative	Execute Commands	vp	Command Volume, Strict Temporal Predictability, Modes, Spacecraft Clock, Command Flexibility, Antenna Number, Antenna Type, & Memory Storage Devices Power Appendage & Power Devices Attitude Control Devices & Attitude Determination Devices Payload Devices
		Control Power	vp	
		Control Attitude	vp	
		Operate Payload Control Orbit Manage Temperature	vp	

Low Volume Command Execution with Command Flexibility	alternative	Execute Commands	vp	Command Volume, Strict Temporal Predictability, Modes, Spacecraft Clock, Command Flexibility, Attitude Control Devices, Attitude Determination Devices, Payload Devices, Memory Storage Devices, Antenna Number, Antenna Type, Power Devices, Temp. Control, Temp. Sensors, Thrusters, and Power Appendage
Temporal Predictable Command Execution with Command Flexibility	alternative	Execute Commands	vp	Command Volume, Strict Temporal Predictability, Modes, Spacecraft Clock, Command Flexibility, Antenna Number, Antenna Type, & Memory Storage Devices
		Control Power	vp	Power Appendage & Power Devices
		Control Attitude	vp	Attitude Control Devices & Attitude Determination Devices
		Operate Payload	vp	Payload Devices
		Control Orbit	vp	
		Manage Temperature	vp	Temp. Control and Temp. Sensors
Housekeeping Data Monitoring	optional	Perform Fault Response	vp	Fault Response Capability
Memory Storage Device Fault Detection	kernel	Perform Fault Response	vp	Memory Storage Devices
Spacecraft Clock	optional	Execute Commands	vp	Spacecraft Clock
High Volume Telemetry Formation	alternative	Collect & Store Data	vp	Data Volume
Flexible & High Volume Telemetry Formation	alternative	Collect & Store Data	vp	Data Volume Algorithm Flexibility
Low Volume Telemetry Formation	alternative	Collect & Store Data	vp	Data Volume
Flexible & Low Volume Telemetry Formation	alternative	Collect & Store Data	vp	Data Volume Algorithm Flexibility
Extensive Telemetry Formation Checks	alternative	Collect & Store Data	vp	Check Type
Quick Telemetry Formation Check	alternative	Collect & Store Data	vp	Check Type
Ground Driven Payload Data Formation	optional	Collect & Store Data	vp	Collection Trigger
Event Driven Payload Data Collection	optional	Collect & Store Data	vp	Collection Trigger
Event Driven Payload Data with Dynamic Collection	optional	Collect & Store Data	vp	Collection Trigger
Ground Driven Housekeeping Data Collection	optional	Collect & Store Data	vp	Collection Trigger

Time Triggered Housekeeping Data Collection	optional	Collect & Store Data	vp	Collection Trigger
Event Driven Housekeeping Data Collection	optional	Collect & Store Data	vp	Collection Trigger
Event Driven Housekeeping Data With Dynamic Collection	optional	Collect & Store Data	vp	Collection Trigger
High Volume Telemetry Storage and Retrieval	alternative	Collect & Store Data	vp	Data Volume Memory Storage Device
Low Volume Telemetry Storage and Retrieval	alternative	Collect & Store Data	vp	Data Volume Memory Storage Device
Tape Recorder Device	alternative	Collect & Store Data	vp	Memory Storage Device
EEPROM Device	alternative	Collect & Store Data	vp	Memory Storage Device
RAM Device	alternative	Collect & Store Data	vp	Memory Storage Device
FLASH Device	alternative	Collect & Store Data	vp	Memory Storage Device
Low Gain Antenna	optional	Execute Commands	vp	Antenna Type and Antenna Number
Med.Gain Antenna	optional	Execute Commands	vp	Antenna Type and Antenna Number
High Gain Antenna	optional	Execute Commands	vp	Antenna Type and Antenna Number
Antenna Gimbal	optional	Execute Commands	vp	Antenna Type and Antenna Number
Photovoltaic	alternative	Execute Commands	vp	Power Type
Radioisotope Thermoelectric Generators	alternative	Execute Commands	vp	Power Type
Power Appendage	optional	Execute Commands	vp	Power Appendage
Heater	optional	Execute Commands	vp	Heater
Louver	optional	Execute Commands	vp	Louver
Thermometer	alternative	Execute Commands	vp	Temperature Sensor
Thermistor	alternative	Execute Commands	vp	Temperature Sensor
Propulsion Thruster	optional	Execute Commands	vp	Propulsion Thruster
Inertial Reference Unit	optional	Execute Commands	vp	Attitude Determination Device
Star Tracker	optional	Execute Commands	vp	Attitude Determination Device
Magnetometer	optional	Execute Commands	vp	Attitude Determination Device
Horizon Crossing Indicator	optional	Execute Commands	vp	Attitude Determination Device
Sun Sensor	optional	Execute Commands	vp	Attitude Determination Device
Solar Tracker	optional	Execute Commands	vp	Attitude Determination Device
Star Scanner	optional	Execute Commands	vp	Attitude Determination Device
Planetary Limb Sensor	optional	Execute Commands	vp	Attitude Determination Device
Reaction Wheel Assembly	alternative	Execute Commands	vp	Attitude Control Device
Torque Rod	alternative	Execute Commands	vp	Attitude Control Device
Reaction Control System	alternative	Execute Commands	vp	Attitude Control Device
Thrusters	alternative	Execute Commands	vp	Attitude Control Device

6.6.2 Software Product Line Use Case Activity Modeling

After the use cases and features are determined, the next step in is to develop customizable activity diagrams from the SPL use case model. Use case activity diagrams provide a more precise description of the use case by depicting the use case in terms of activity nodes and decision nodes. The process for creating these diagrams was previously described in section 5.5.1. These activity diagrams will be later used to assist in the interconnecting of design patterns and the functional validation process.

The process for creating use case activity models is described below for the FSW SPL. This is accomplished by creating an activity diagram is created for each use case associated with the FSW SPL command and data handling subsystem. In addition, to showing the flow of actions in a use case, the activity diagrams also must explicitly associate the features in the FSW SPL command and data handling feature model with activities in the activity diagrams. Table 6-2 shows a list of feature conditions created to represent features in the feature model of the FSW SPL command and data handling subsystem. These will be used to associate the features to the activities and to control the execution of activities in the activity diagram depending on feature selections.

The command and data handling subsystem in the FSW SPL is associated with three use cases. Thus three use case activity diagrams are created for the FSW SPL's command and data handling subsystem. The process for creating activity diagrams is illustrated using the Execute Commands use case. However, additional activity diagrams can be found in Appendix B.

Table 6-2 Feature conditions for FSW SPL C&DH subsystem

Feature Condition	Feature Selections
CommandExecution	{LowVolume, HighVolume, TimeTriggered, LowVolumeWithCmdFlex, HighVolumeWithCmdFlex, TimeTriggeredWithCmdFlex}
HousekeepingDataMonitor	{T,F}
MemoryStorageDeviceFaultDetection	{T}
MaintainSpacecraftClock	{T,F}
TelemetryFormation	{HighVolume, FlexHighVolume, LowVolume, FlexLowVolume}
TelemetryFormationReliability	{SignificantChecks, QuickCheck, F}
GroundDrivenPayloadDataCollection	{T,F}
EventDrivenPayloadDataCollection	{T,F}
EventDrivenPayloadDataWithDynamicCollection	{T,F}
GroundDrivenHousekeepingDataCollection	{T,F}
TimeTriggeredHousekeepingDataCollection	{T,F}
EventDrivenHousekeepingDataCollection	{T,F}
EventDrivenHousekeepingDataWithDynamicCollection	{T,F}
TelemetryStorageandRetrieval	{LowVolume, HighVolume}
MemoryStorageDevice	{Tape, RAM, EEPROM}
AntennaType	{HGA, LGA, MGA}
AntennaGimbal	{T, F}
PowerDevice	{Photovoltaic, Radiosotope Thermoelectric Generators}
PowerAppendage	{T, F}
Heater	{T, F}
Louver	{T, F}
TempSensor	{Thermometer, Thermistor}
AttitudeDeterminationDeviceType	{IRU, StarTracker, Magnetometer, HCI, SunSensor,

	SolarTracker, PlanetaryLimbSensor, StarTracker}
AttitudeControlDeviceType	{RWA, TorqueRod, RCS, Thrusters}

The Execute Commands use case involves executing commands from the ground station to ensure the spacecraft is not put into an unsafe state and the actions are appropriate for the spacecraft's mode. The activity diagram for this use case is depicted in Figure 6-5.

This scenario involves six alternative paths that are based on the Command Execution feature group. For example, if CommandExecution = "LowVolume" then steps one and two are taken through the activity diagram. The variation points that impact the activity node are listed in parenthesis in the activity node. To fully specify the use case scenario, each adaptable step should have a separate sub-activity diagram illustrating the different variants and steps. However, due to significant amount of variability in adaptable steps 2, 4, 5, 7, 9, and 10, as seen by the multitude of variation points impacting the steps, sub-activity diagrams for all possible combinations will not be created. Instead, all the potential individual interactions will remain abstracted at this level. Sub-activity diagrams are also not created for adaptable steps 1,3, 6, and 8. This is because they are impacted by the Payload Devices variation point. The variants of the Payload Device variation point are not known at the SPL level since payload devices are unique to each spacecraft. Therefore the sequencing logic of the variants cannot be modeled.

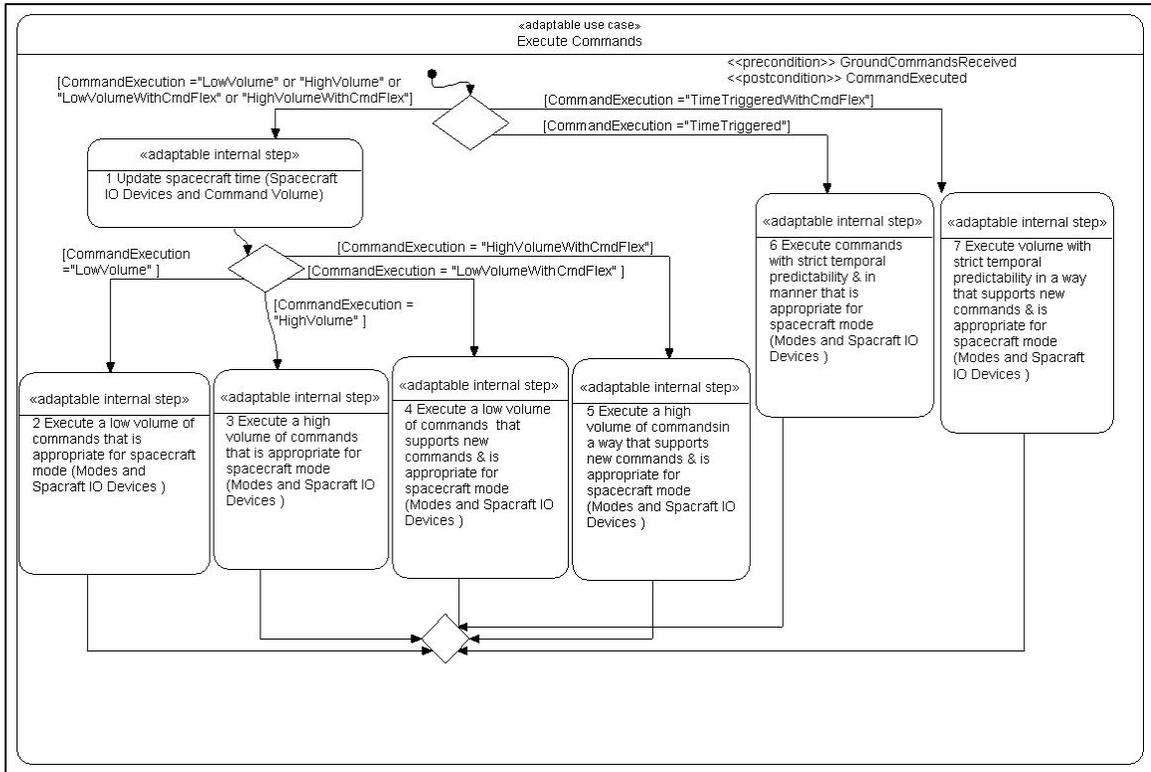


Figure 6-5 Execute Commands activity diagram

6.7 Software Product Line Analysis Modeling Phase

The next phase in the SPL engineering process is the analysis modeling phase. This phase involves creating the static and dynamic models of the SPL. The key outputs from this phase include the conceptual static model, context model, subsystem structuring, feature to design pattern mapping, and SPL level architectural and executable design patterns. The process to feature to design pattern mapping and SPL level architectural

and executable design patterns is illustrated using the FSW SPL. The remaining artifacts from this phase are found in Appendix B.

6.7.1 Software Product Line Feature to Design Pattern Mapping

One of the key artifacts in the SPL analysis modeling phase is the feature to design pattern mapping. The feature to design pattern mapping is where this approach is truly novel and differentiates itself from other SPL approaches. It identifies where design patterns can be used in the analysis model and then relates them back to the SPL features. Using design patterns instead of classes provides the SPL with extra architectural flexibility because the SPL's variability is captured at a higher degree of granularity. This is because the design patterns contain customizable components and interconnections rather than specific components. Therefore several different combinations of specific components and interconnections are abstracted into one design pattern, resulting in a higher degree of granularity. Additionally, feature to design pattern mapping captures the rules for integrating design patterns. For example, if two alternative features are each mapped to a different design pattern, then these design patterns cannot be integrated because they are alternatives. The process for deriving the feature to design pattern mapping is described above in section 5.3.

The feature to design pattern mapping process demonstrated using three of the FSW SPL's pattern specific features. Additional dynamic models can be found in Appendix B.

6.7.1.1 *Flight Software Product Line Dynamic Model for Low Volume Command Execution*

First, the objects that participate in the Low Volume Command Execution pattern specific feature are modeled and examined. This pattern specific feature is derived from the Execute Commands use case where the Command Volume variation point is low, Command Flexibility variation point is false, and Strict Temporal Predictability variation point is false. Figure 6-6 shows a communication diagram for feature. In this set of interactions, due to the small amount of commands that need processing, one controller receives a set of ground commands from the Telemetry Tracking and Control Subsystem. It then determines the sequence of the actions to ensure that it does not put the spacecraft in an unsafe state.

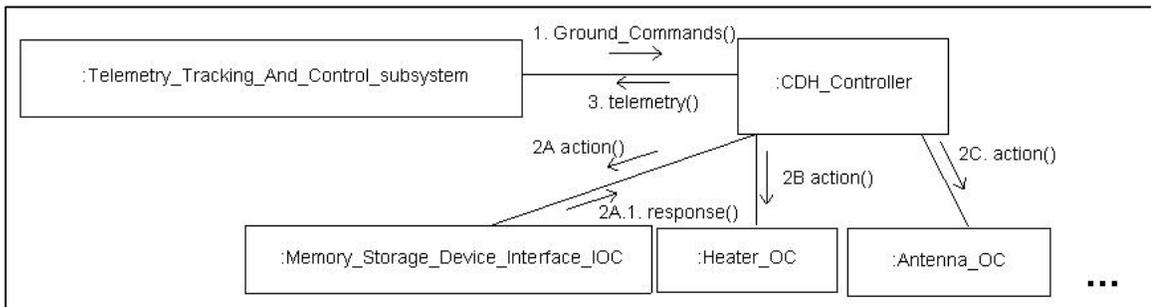


Figure 6-6 Low Volume Command Execution communication diagram

Then it executes the commands by invoking the appropriate actions on the output and IO components identified in the pattern specific feature. The Low Volume Command Execution pattern specific feature includes invoking actions on the kernel input, output and IO, such as the Memory Storage Device IO component and an Antenna output component.

Next, the variability from the pattern variability features is added to the interaction diagram. The Low Volume Command Execution pattern specific feature is influenced by multiple pattern variability features including the Antenna Gimbal, Power Appendage, Propulsion Thruster pattern variability features. These features result in the need for additional components and they are added to the communication diagram as seen in Figure 6-6. The Low Volume Command Execution pattern specific feature is influence by the Temperature Sensor and Temperature Control Device pattern variability feature groups, which result in the need for additional components and variant components. In order to keep the diagram manageable, only the base classes are depicted in Figure 6-6. Finally, the Low Volume Command Execution pattern specific feature is impacted by other pattern variability feature groups, such as the Memory Storage Device pattern variability feature group, which results in a set of variant components. Again, to keep the diagram manageable, only the base classes are modeled.

Finally, the interactions involved in the communication diagram are analyzed and applicable design patterns are identified. In the Low Volume Command Execution pattern specific feature, the interactions are consistent with Centralized Control design pattern (Gomaa 2011). Therefore this pattern specific feature is mapped to the Centralized Control design pattern.

6.7.1.2 Flight Software Product Line Dynamic Model for High Volume Command Execution

The objects that participate in the High Volume Command Execution pattern specific feature are modeled and examined. This pattern specific feature is derived from the Execute Commands use case where the Command Volume variation point is high, Command Flexibility variation point is false, and Strict Temporal Predictability variation point is false. Figure 6-7 shows a communication diagram for the High Volume Command Execution pattern specific feature. In this set of interactions, one coordinator receives a set of ground commands from the Telemetry, Tracking, and Control subsystem and determines the execution sequence. However, due to the large volume of commands that need to be executed, it cannot execute all the commands without becoming a system bottleneck. Therefore the coordinator sends the commands to localized controllers to execute. The localized controllers execute the commands by invoking the appropriate actions on the output and IO components identified in the pattern specific feature. The High Volume Command Execution pattern specific feature includes invoking actions on

the kernel input, output and IO, such as the Memory Storage Device IO component and an Antenna output component.

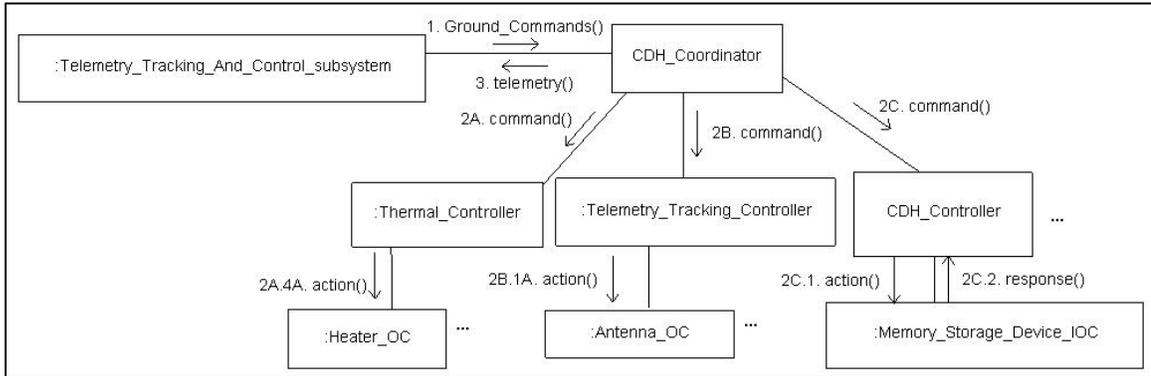


Figure 6-7 High Volume Command Execution communication diagram

Next, the variability from the pattern variability features is added to the interaction diagram. The High Volume Command Execution is impacted by the same pattern variability features as the Low Volume Command Execution pattern specific feature. Therefore the same additional components and variants are added to the communication diagram as seen in Figure 6-7.

Finally, the interactions involved in the communication diagram are analyzed and applicable design patterns are identified. The interactions and distribution of workload in this feature are consistent with the Hierarchical Control design pattern (Gomaa 2011).

Therefore the High Volume Command Execution pattern specific feature is mapped to the Hierarchical Control design pattern.

6.7.1.3 Flight Software Product Line Dynamic Model for Time-Triggered Command Execution

Next, the object interactions in the Time-Triggered Command Execution pattern specific feature are modeled and examined. This pattern specific feature is derived from the Execute Commands use case where the Command Volume variation point is either high or low, Command Flexibility variation point is false, or Strict Temporal Predictability variation point is true. The Time-Triggered Command Execution's communication diagram is shown in Figure 6-8. In this instance, ground commands need to be executed with strict temporal predictability. This is best achieved using multiple controllers on a time trigger architecture that execute commands and send control data to the other controllers at predictable time intervals.

Next, the variability from the pattern variability features is added to the interaction diagram. The Time-Triggered Command Execution is impacted by the same pattern variability features as the Low Volume Command Execution pattern specific feature. Therefore the same additional components and variants are added to the communication diagram as seen in Figure 6-8.

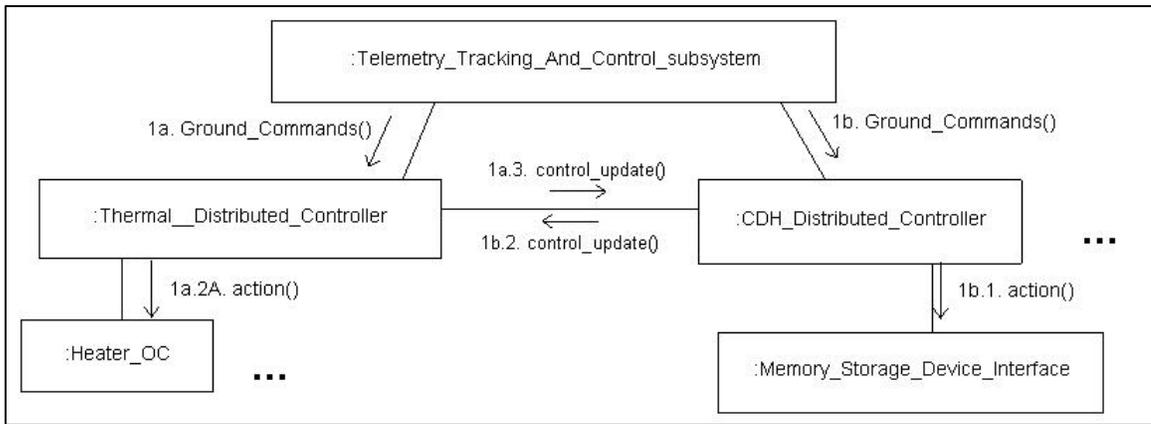


Figure 6-8 Time Triggered Command Execution communication diagram

Finally, the interactions involved in the communication diagram are analyzed and applicable design patterns are identified. The interactions in this feature are consistent with Distributed Control design pattern (Gomaa 2011). Thus the Time-Triggered Command Execution pattern specific feature is mapped to the Distributed Control design pattern.

6.7.1.4 Flight Software Product Line Command and Data Handling Subsystem

Feature to Design Pattern Mapping Summary

A final summary of the feature to design pattern mapping for the FSW SPL's command and data handling subsystems features are listed in Table 6-3. Each of the command and data handling features is mapped to a single design pattern or pre-integrated design pattern that supports the feature's functionality.

Table 6-3 Command and Data Handling Feature to Design Pattern Mapping

Feature Group	Variability	Feature	DRE Design Pattern
<<exactly-one-of feature group>> Command Execution	alternative	High Volume Command Execution	Hierarchical Control
	alternative	Low Volume Command Execution	Centralized Control
	alternative	Time-Triggered Command Execution	Distributed Control
	alternative	High Volume Command Execution with Command Flexibility	Hierarchical Control with Command Dispatcher
	alternative	Low Volume Command Execution with Command Flexibility	Centralized Control with Command Dispatcher
	alternative	Time-Triggered Command Execution with Command Flexibility	Distributed Control with Command Dispatcher
<<exactly-one-of feature group>> Telemetry Storage and Retrieval	alternative	High Volume Telemetry Storage and Retrieval	Compound Commit
	alternative	Low Volume Telemetry Storage and Retrieval	Client Server
<<exactly-one-of feature group>> Telemetry Formation	alternative	High Volume Telemetry Formation	Master Slave with Pipes and Filters
	alternative	Flexible High Volume Telemetry Formation	Master Slave with Pipes and Filters & Strategy
	alternative	Low Volume Telemetry Formation	Pipes and Filters
	alternative	Flexible Low Volume Telemetry Formation	Pipes and Filters with Strategy
<<exactly-one-of feature group>> Format Telemetry Reliability	alternative	Extensive Checks	Protected Single Channel
	alternative	Quick Check	Sanity Check
<<at-least-one-of feature group>> Payload Data Collection	optional	Ground Driven Payload Data Collection	Multiple Client Multiple Server
	optional	Event Driven Payload Data Collection	Multicast
	optional	Event Driven Payload Data with Dynamic Collection	Publish Subscribe
<<at-least-one-of feature group>> Housekeeping Data Collection	optional	Ground Driven Housekeeping Data Collection	Multiple Client Multiple Server
	optional	Time Triggered Housekeeping Data Collection	Broadcast
	optional	Event Driven Housekeeping Data Collection	Multicast
	optional	Event Driven Housekeeping Data with Dynamic Collection	Publish Subscribe
N/A	optional	Housekeeping Data Checks	Multicast

N/A	optional	Spacecraft Clock	Multicast
N/A	kernel	Memory Storage Device Fault Detection	Watchdog

6.7.2 Software Product Line Architectural Design Pattern Modeling

After design patterns are mapped to features, the architectural design patterns can then be customized to the SPL. This is done to make the design patterns more directly applicable to FSW architectures and to save time when instantiating the design patterns for a specific SPL member. This process is feature driven, where the DRE architectural and executable design patterns are systematically updated to reflect the needs of the SPL features as described in section 5.4.1.

The process for updating the architectural design patterns is demonstrated using the FSW SPL. The required updates to the architectural views and executable version for the FSW Centralized Control of the design patterns is described in more detail below. Additional information on the FSW SPL design patterns is found in Appendix B.

The Centralized Control design pattern is mapped to the Low Volume Command Execution feature. This process for updating the DRE Centralized Control architectural design patterns is feature driven, where it is systematically updated to reflect the needs of the SPL pattern specific and pattern variability features. The required updates to the architectural views and executable version are described in more detail below.

First, the general purpose components in the DRE architectural design pattern are updated to be SPL specific based on the features, following the process described in Chapter 5. Figure 6-9 depicts the customized collaboration for the FSW Centralized Control design pattern. First, the Centralized_Controller is the only kernel component with a multiplicity of one must be used. Therefore its name is updated to be the CDH_Centralized_Controller to reflect the needs of the Low Volume Command Execution pattern specific feature.

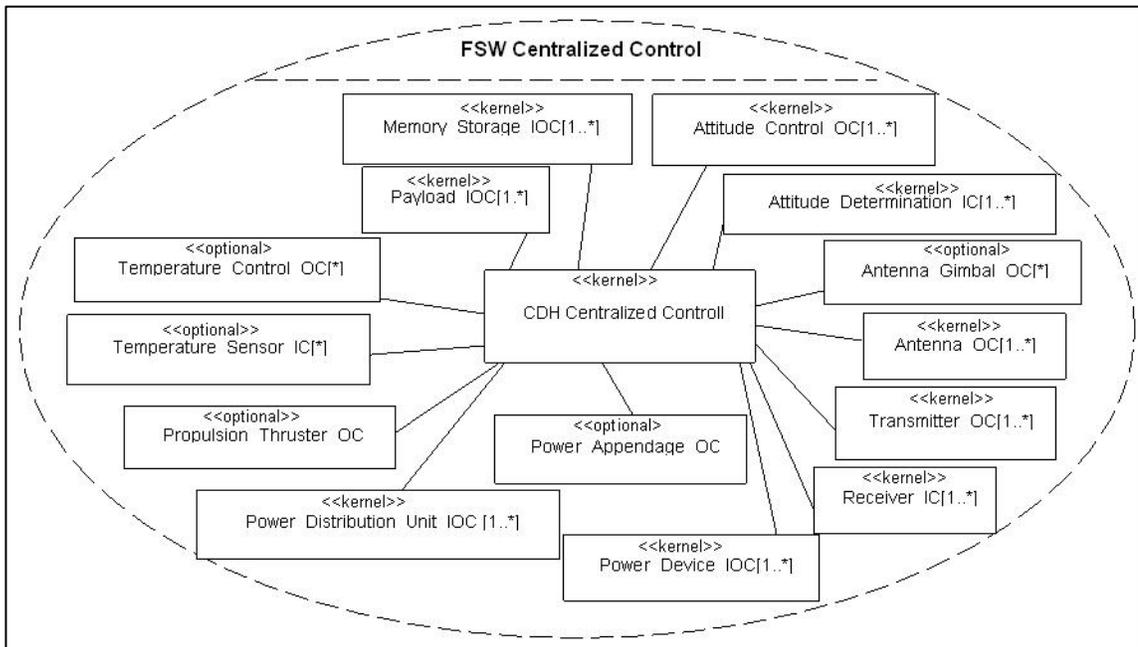


Figure 6-9 FSW level Centralized Control collaboration diagram

Second, the DRE level Centralized Control architectural design pattern contains an optional IO_Component with zero or many multiplicities. Based on the pattern specific and pattern variability features it is determined that this component is required because the FSW SPL has four features relating to IO devices. According to the Low Volume Command Execution pattern specific feature, the all members of the FSW SPL are required to interface with one or more memory storage devices, power distribution units, power devices, and payload devices. Therefore four versions of the IO_Component are created and given SPL names reflective of the SPL's devices. Additionally, the multiplicity is updated to one or many and the variability is updated to kernel to reflect the needs of the feature.

Additionally, the Memory Storage Device Type, Power Device Type, and Payload Device Type pattern variability features also point to variability in this design pattern. Each of these features contains a set of alternative device types that can be selected for an SPL member. To reflect this information, the different device types are modeled as variants components to the device superclass and use the variant stereotype. However, they are not graphically depicted in Figure 6-9 to keep the diagram readable.

This same process is followed for the other optional components with zero or many multiplicities in the DRE Centralized Control collaboration diagram, which include the input and IO components. The result is the collaboration diagram depicted in Figure 6-9.

Next, the object interactions for the DRE Centralized Control architectural design pattern are customized for the FSW SPL following the process defined in Chapter 5. This design pattern is mapped to the Low Volume Command Execution pattern specific feature and several other pattern variability features relate to variability in the design pattern. The Low Volume Command Execution pattern specific feature captures one scenario where the FSW processes and executes a set of ground commands, which involves invoking actions on the input, output, and IO components. The type and amount of input, output, and IO components is influenced by several other features, such as the Memory Storage Device pattern variability feature group that captures the alternative memory storage devices that can be used. Due to the amount of variability in this scenario, only a representative set of object interactions is created for this use case.

The set of representative object interactions for this architectural design pattern is depicted in Figure 6-10. Notice, the SPL components are used rather than the general purpose DRE level components. The variant components are not shown to keep the diagram readable. A set of alternative branches is used to represent the different actions that can be performed on different devices in response to a command.

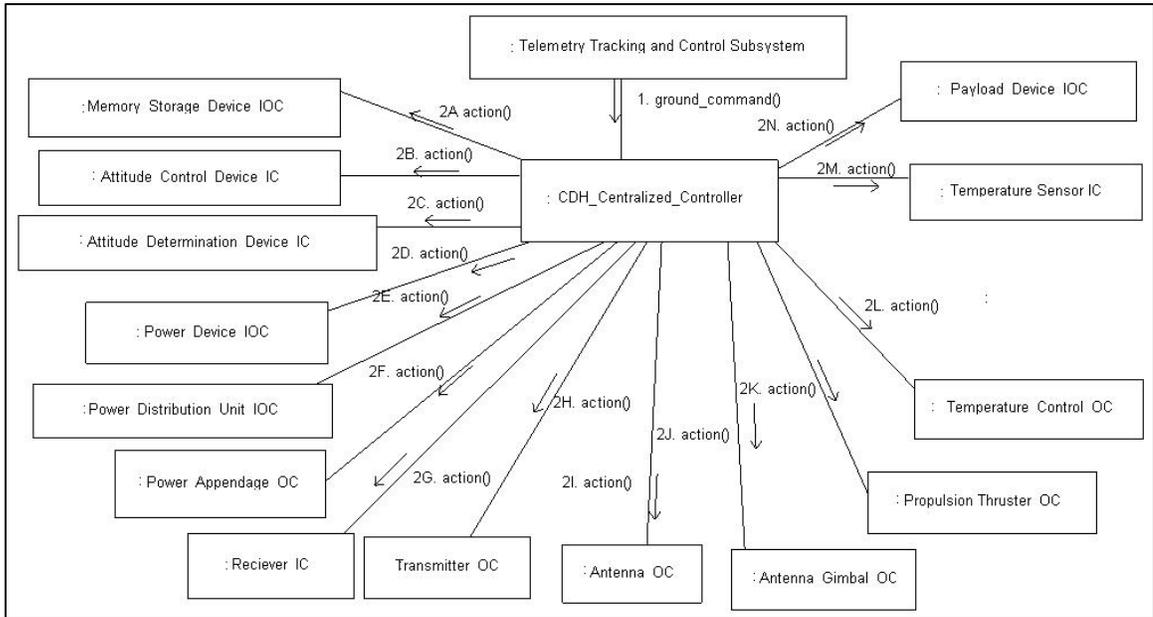


Figure 6-10 FSW Centralized Control execute commands interaction diagram

Finally, the last architectural view that needs to be updated is the communication between the components in the design pattern, which is captured in the component diagram. The component diagram for the FSW Centralized Control is depicted in Figure 6-11. Figure 6-11 shows that the general purpose Input Component is customized to the FSW SPL specific input components, which are the Receiver_IC, Temperature_Sensor_IC, and Attitude_Determination_IC. Additionally, the names of the required and provided ports are also updated to reflect the FSW SPL specific components. The same changes have also been applied to the output and IO components.

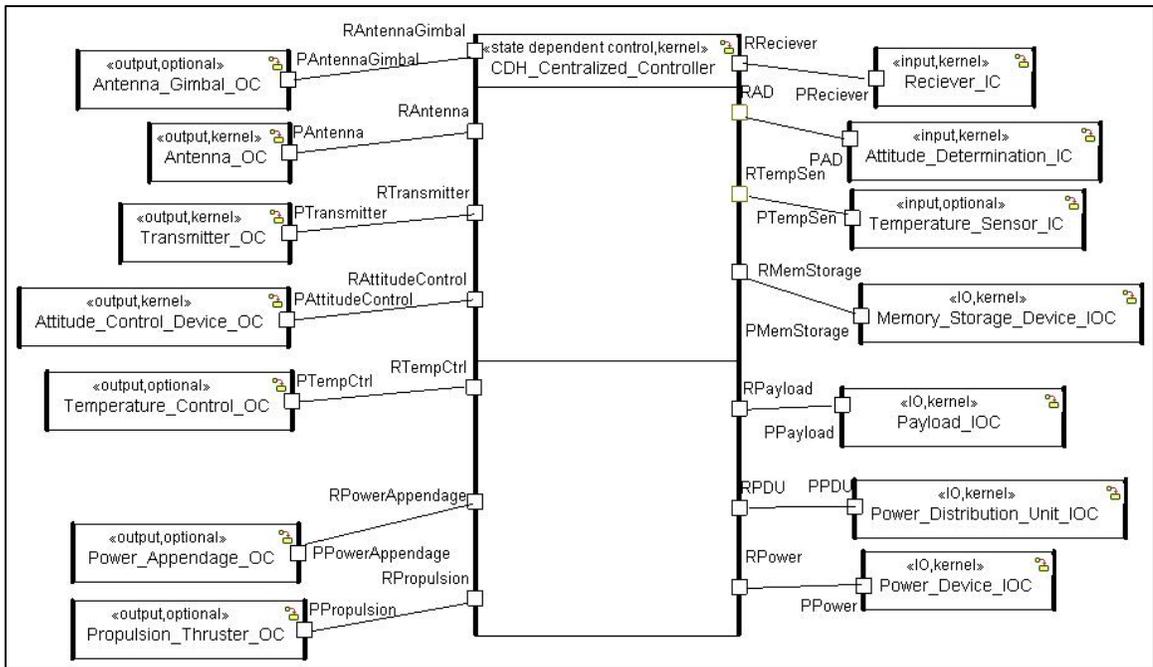


Figure 6-11 Component diagram for FSW Centralized Control

Finally, the required and provided interfaces between the ports must be customized to reflect the SPL customizations. These changes are illustrated on the port design component diagram. Figure 6-12 shows a subset of the port design for the FSW Centralized Control architectural design pattern. This shows that the connected ports have compatible required and provided interfaces that are reflective of the FSW SPL. For example, instead of listing a general output component from the Centralized Control architectural design pattern, the Heater_OC is shown, along with an updated port name. Its interface is an updated version of the IOutput interface from the Centralized Control

architectural design pattern. The general purpose action() method is replaced with the FSW SPL specific methods the Heater_OC must provide based on the pattern specific and pattern variability features. For instance, the Low Volume Command Execution pattern specific feature requires ground commands that turn the heater on and off. Therefore the FSW SPL methods which provide this functionality are added to the IHeater interface.

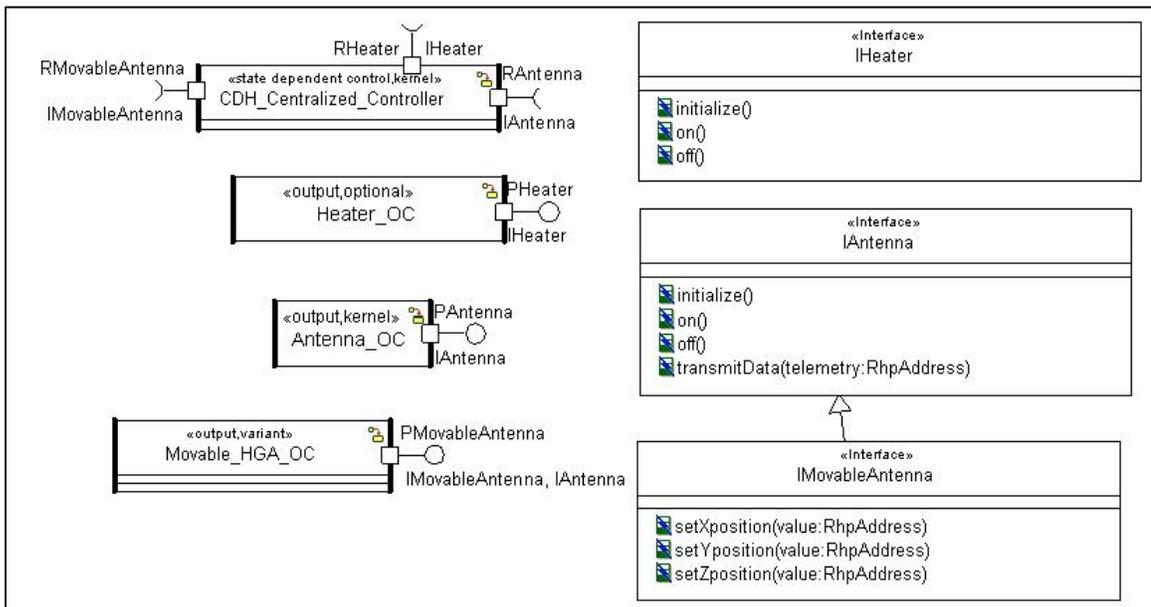


Figure 6-12 Port design for FSW SPL Centralized Control design pattern

6.7.3 Software Product Line Executable Design Pattern Modeling

After architectural design pattern is updated, the next step is to customize the executable design patterns. This is done to make the design patterns more directly applicable to FSW architectures and to save time when instantiating the design patterns for a specific SPL member. This process is feature driven, where the DRE architectural and executable design patterns are systematically updated to reflect the needs of the SPL features as described in section 5.4.2.

This process is demonstrated for the FSW SPL. According to the Low Volume Command Execution pattern specific feature, the CDH Centralized Controller must manage and take into account the spacecraft mode during command execution. The mode management function extends the behavior of the Centralized Controller. Therefore common modes, including launch mode, safe mode, and normal mode are added modeled as new states at the highest level in the state machine hierarchy, as seen in Figure 6-13. Within each of these states are the original Idle and Controlling states from the DRE level Centralized Controller. Also according to the Low Volume Command Execution pattern specific feature, the CDH Centralized Controller must validate all ground commands or responses to onboard events to ensure that it does not put the spacecraft in an unsafe state. This functionality refines the behavior of the Controlling state. Therefore this logic is modeled as substates within each of the Controlling state. These substates are modeled within each of the Controlling States, but to make the

diagram readable they are only depicted in the Normal Mode's Controlling state in Figure 6-13. The specific algorithms and logic used within the Validating Command, Executing Commands, Logging Commands and Rejecting commands substates is application specific. Therefore this information is not modeled as activities or actions in the FSW SPL state machine.

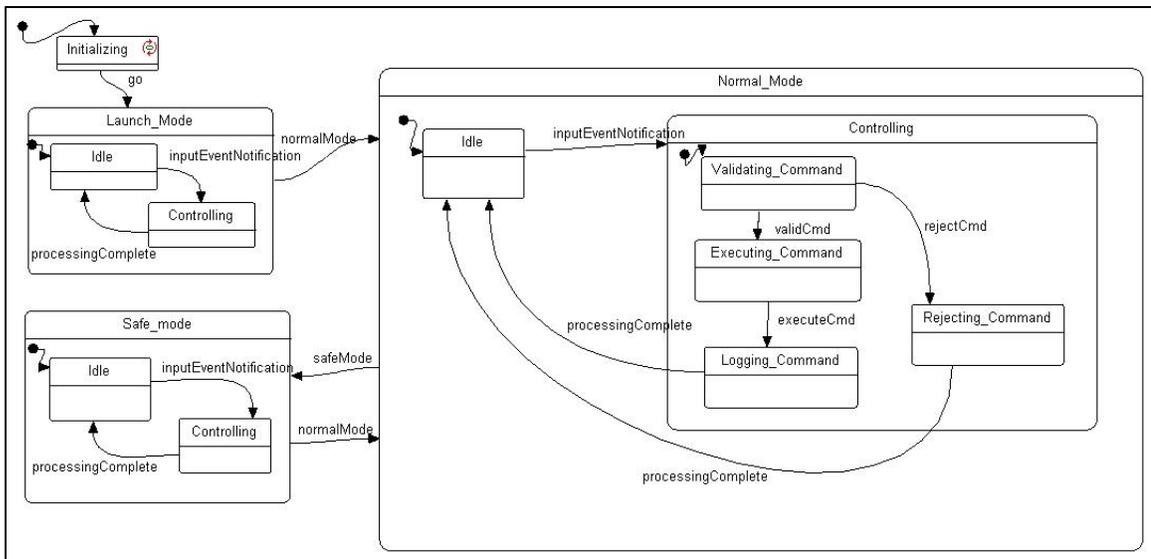


Figure 6-13 FSW CDH_Centralized_Controller state machine

The next component that needs to be updated in the command and data handling subsystem is the Heater_OC component, which are responsible for directly interfacing with the heater. The FSW Heater_OC is a specialized version of the DRE Output_Component. The state machine for the Output_Component is depicted in Figure

4-12. The customized version of the DRE Output_Component's state machine for the Heater_OC is shown in Figure 6-14. The Heater_OC extend the behavior of the Output_Component, thus additional states are added to the state machine. The Executing Command state is updated with the Turning Heater On and Turning Heater Off states.

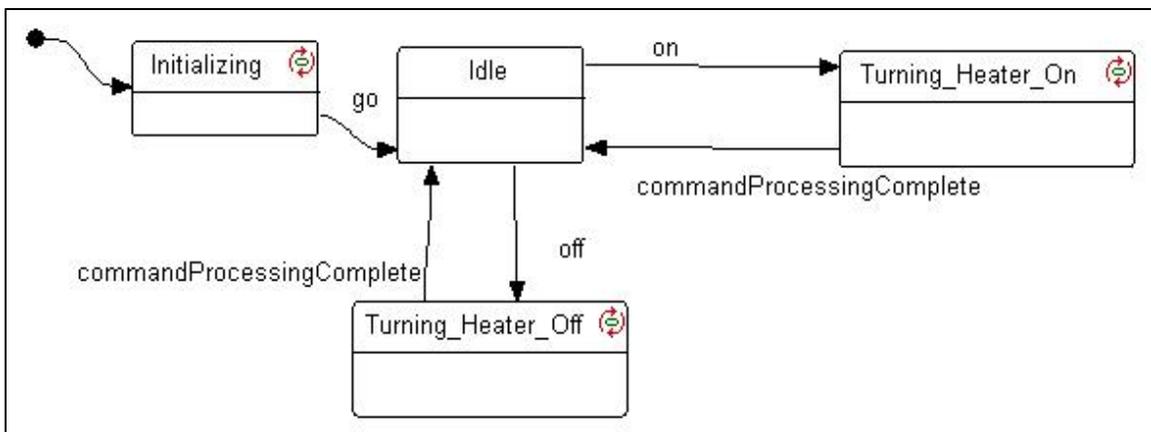


Figure 6-14 FSW Heater_OC state machine

FSW SPL customized state machines and interfaces for the Antenna_OC and Memory_Storage_Device_IOC were developed as part of this research following the process described in Chapter 5. However, FSW SPL specific interfaces and state machines were not created there remaining input, output, and IO components in this design pattern. This is because a significant amount of domain analysis and discussions

with hardware experts to properly develop the interfaces and state machines. Since the customization process was already applied and proven successful on multiple other components in the FSW SPL the other input, output, and IO components were not pursued. Instead they level the interfaces and state machines for their corresponding DRE base classes and were tested using those interfaces.

6.7.4 Software Product Line Design Pattern Interconnection

After the architectural and executable design patterns are customized for the FSW SPL, the next step is to address how the design patterns are integrated together to form software architectures. This is accomplished by created interaction overview diagrams. These diagrams are created using a use case scenario driven approach is used to interconnect design patterns to achieve the FSW SPL functionality. The process for deriving this artifact is described above in section 5.5.1.

The process for creating interaction overview diagram sis illusted using the FSW SPL. The use case scenarios that involve the FSW SPL Command and Data Handling subsystem include the execute commands scenario, collect and store housekeeping data scenario, and a portion of the perform fault management. The design pattern interconnection process is demonstrated using the execute commands use case. However, additional interaction overview diagrams for the FSW SPL are found in Appendix B.

The execute commands use case scenario captures how the spacecraft will manage the execution of commands. As seen in its activity diagram in Figure 6-5 for this use case there are several alternative ways this can be performed. To create an interaction overview diagram for this use case, the design pattern that supports each activity is determined and modeled in an interaction overview diagram. The interaction overview diagram for the Execute Commands use case is depicted in Figure 6-15. Each of the activity steps in the use case activity diagram is replaced with references to their supporting design pattern interaction diagrams. The Update Spacecraft Time steps are supported by the Spacecraft Clock feature. As seen in Table 6-3, the Spacecraft Clock feature is mapped to the FSW Spacecraft Time Multicast design pattern. All Update Spacecraft Time steps are replaced with a reference to the FSW Spacecraft Time Multicast's interaction diagram.

Next, each of the alternative execute command steps in the activity diagram in Figure 6-5 are updated to their design patterns according using the feature based condition on the execution path. For example, on the far left the path's feature condition is `CommandExecution = "LowVolume."` Using the feature to design pattern mapping in Table 6-3 the Low Volume Command Execution feature is mapped to the FSW Centralized Control design pattern. Therefore this step is replaced with a reference to the FSW Centralized Control's interaction diagram.

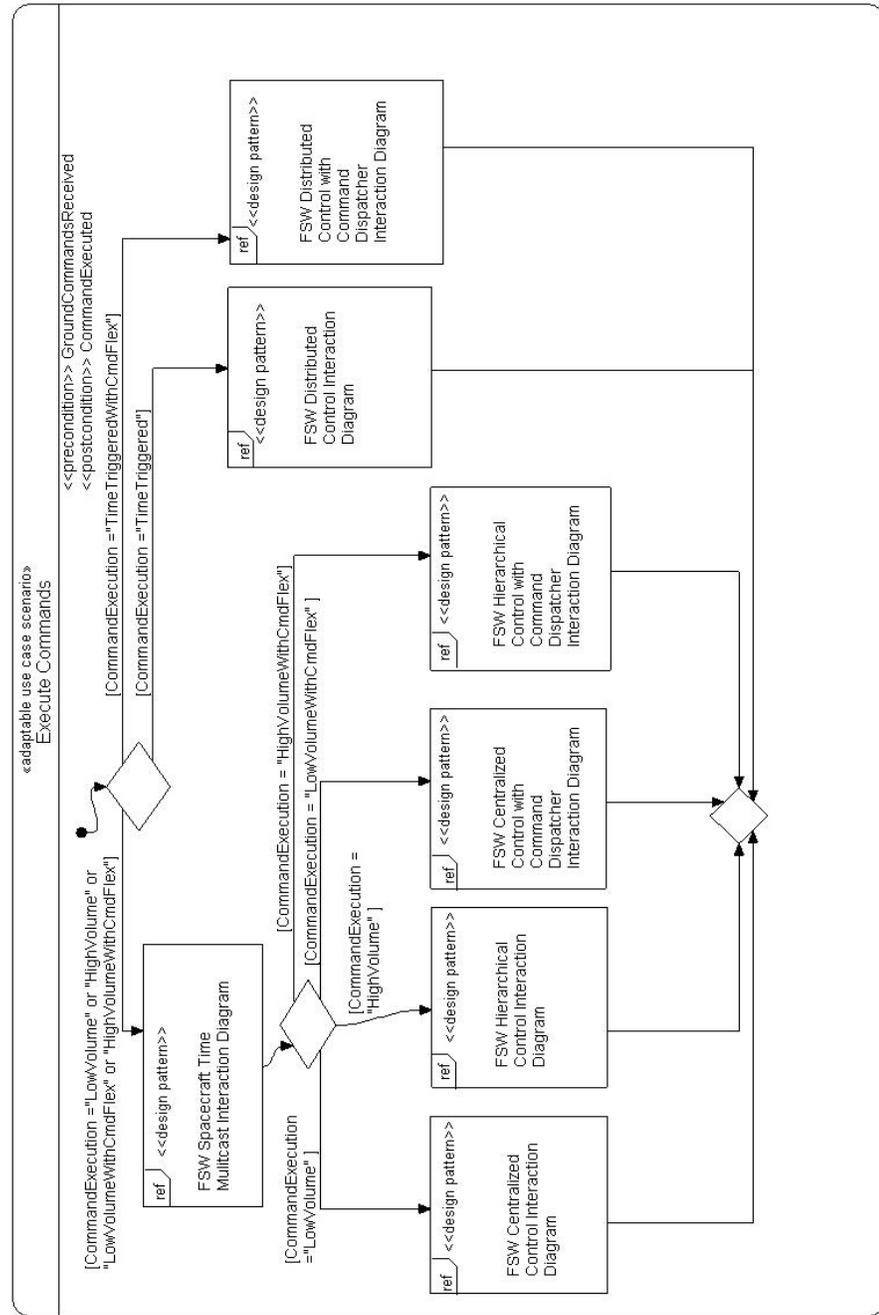


Figure 6-15 Interaction overview for Execute Commands use case

After the interaction overview diagram is created, it is then analyzed to identify where design patterns are interconnected. If there are two sequence design patterns along an execution path, then these design patterns must be interconnected. In the Command Execution use case, the FSW Spacecraft Time Multicast design pattern interconnects with the FSW Centralized Control, FSW Hierarchical Control, FSW Centralized Control with Command Dispatcher, and FSW Hierarchical Control with Command Dispatcher design patterns.

Once the design patterns that interconnect are identified, the specific components within these patterns that communicate are identified and interconnected with connectors. For example, in the case where the FSW Spacecraft Time Multicast design pattern interconnects with the FSW Centralized Control, the spacecraft clock multicast component sends time updates to the CDH_Centralized_Controller so that it has the current time when determining the command execution order. The Spacecraft IO devices also need a time update for time tagging their data. Therefore a connector must be created between all of these components. This interconnection is demonstrated using the CDH_Centralized_Controller and Spacecraft_Clock_Multicast component depicted in Figure 6-16. Since the Spacecraft_Clock_Multicast component is sending input to the CDH_Centralized_Controller, a required port called RController is added to the Spacecraft_Clock_Multicast component. It is connected to the CDH_Centralized_Controller's provided port called PController.



Figure 6-16 Interconnection between CDH_Centralized_Controller and Spacecraft_Clock_Multicast

6.8 Software Product Line Design Modeling

The purpose of the SPL design modeling phase is to map the analysis model to the operational environment. This involves structuring the SPL into components that could potentially execute on different nodes and identifying the communication between components (Gomaa 2005). One of the most important decisions in the design modeling phase is the architecture that will be used and the type of message communication design patterns that occurs between components. This information is captured in the SPL because it is not anticipated that applications will need to change this information.

6.8.1 Software Product Line Architecture

After the design patterns are interconnected, this information is used to help identify where the design patterns and their components fit into the overall software architecture. This is accomplished by analyzing the communication within a design pattern and the design pattern interconnections. For instance if a component from one design pattern utilizes a service from a component in another design pattern, then in a layered

architecture the component using the service is in a higher layer than the component providing the service.

This process is demonstrated using the FSW SPL. The software architecture for the FSW SPL is design as a layered architecture based on the Five Layers (Douglass 2003) and Layers of Abstraction (Gomaa 2005; Douglass 2003) architectural design patterns. The components are structured into layers where it may use the services provided by components in lower layers, but not the layers above. The layered architecture for the FSW SPL is depicted in Figure 6-17. First, the Five Layer Architecture organizes the software into the user interface layer, application layer, communication layer, abstract OS layer, and finally the abstract hardware layer. In FSW there is no user interface therefore the user interface layer is not used. The command and data handling subsystem falls in the Application Layer.

The Layers design pattern is used to provide structure to the Application Layer software.

The components from the design pattern are placed into the appropriate layers as follows:

- **Layer 1: Device Interface Layer.** This layer consists of the components that interface directly with the FSW SPL hardware.
- **Layer 2: Server Layer.** This layer contains the server components in the FSW SPL.
- **Layer 3: Fault Detection Layer.** This layer is composed of the components that assist in the fault detection of the FSW SPL's hardware.

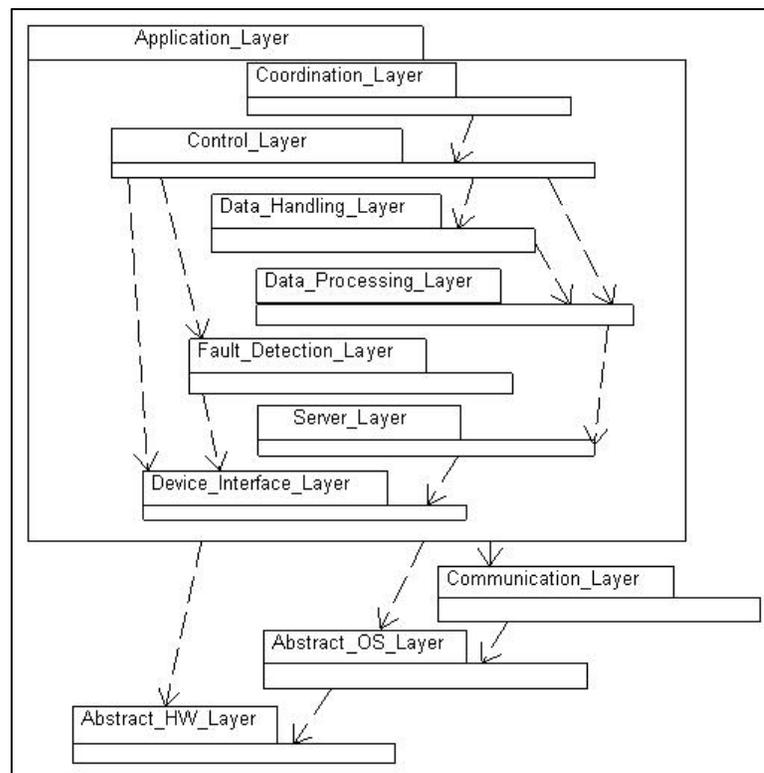


Figure 6-17 Layered Architecture for FSW SPL

- **Layer 4: Data Processing Layer.** This layer consists of components that are involved in manipulating and processing spacecraft data.
- **Layer 5: Data Handling Layer.** This layer is composed of the components that are involved in the collection and distribution of spacecraft data.
- **Layer 6: Control Layer.** This layer consists of the components that provide control in the FSW SPL.

- **Layer 7: Coordination Layer.** This layer contains the components that provide coordination of the FSW SPL.

The components from the design pattern are placed into the appropriate layers based on their interactions with other components. For instance from the FSW Centralized Control design pattern, the CDH_Centralized_Controller is placed in the Control Layer because it provides overall control for the system and relies on services provided by other components in the architecture. The various input, output, and IO components from the FSW Centralized Control design pattern interface with the hardware and provide services related to the hardware to the other components in the architecture. Therefore these components are placed in the Device Interface Layer, which is the lowest layer in the architecture. Another example is the Hierarchical Control design pattern. In this design pattern, there is a CDH_Coordinator, which is placed at the highest layer because it provides overall coordination. The FSW subsystem controllers from this design pattern are placed in the Control Layer because they provide control for the subsystems and rely on services provided by other components in the architecture. Again, the various input, output, and IO components from this design pattern interface with the hardware and provide services related to the hardware to the other components in the architecture.

6.8.2 Software Product Line Message Communication

Another important decision made in the design modeling phase is the specific type of message communication will be used in the SPL. This information is captured in the SPL because it is not anticipated that applications will need to change this information.

This process is demonstrated using the FSW SPL. The specific type of message communication used in the FSW SPL is as follows:

- **Asynchronous Message Communication and Bidirectional Asynchronous Message Communication.** The FSW SPL was primarily designed to level asynchronous communication for all one-way communication and bi-directional asynchronous communication for all two-way communication. This decision was made because it provided the most flexibility in the system and efficient use of concurrent components.
- **Synchronous Message Communication with Reply.** The FSW SPL did use limited synchronous messages communication in the data processing functionality of the command and data handling system. The raw data is transformed into telemetry packets using a serial process where the producers must wait on the consumers to finish. Therefore synchronous messages communication was used in this instance.
- **Publish/Subscribe Message Communication.** The Publish/Subscribe message communication pattern was used in the FSW SPL to ensure the consistency with the Layered architecture. For example, some components in the Control layer need to be notified when a fault is detected by a component in the Fault Detection layer. Therefore to ensure the Control Layer is dependent on the Fault Detection layer the publish/subscribe communication is used. Where the components in the

Control Layer subscribe to fault detection alerts from components in the Fault Detection Layer. Then when a fault is detected the component in the Control layer is notified.

6.9 Comparison of Approaches

The pattern based SPLE approach has several benefits that make it more scalable when compared to component/connector based SPLE. The pattern based SPLE is more scalable because it defers some variability modeling until the application engineering phase. Thus it trades SPLE development with application development. A comparison of the two approaches is detailed in Table 6-4. This table also summarizes what parts of the SPLE development are deferred until application development.

Table 6-4 Comparison pattern based SPLE with versus component/connector based SPLE

	Pattern based SPLE	Component/connector based SPLE
Variability Level	Subsystem/design pattern	Subsystem/component/connector
Component Internal Behavior Variability	Design patterns with customizable components <ul style="list-style-type: none"> Capture representative behaviors and interactions 	Parameterized and variant classes <ul style="list-style-type: none"> Fully specify all possible behaviors and interactions
Component Interaction Variability	Representative object interactions	All possible object interactions
Component Variability Mechanisms	Representative object behavior <ul style="list-style-type: none"> Plug compatible interfaces can be used to further reduce variability when variants use the same interface 	All possible internal behaviors <ul style="list-style-type: none"> Plug compatible interfaces can be used to reduce variability when variants use the same interface Additional modeling when variants do not share the same interfaces

Product Derivation Engineering Effort	Moderate <ul style="list-style-type: none"> • Design pattern customizations • Corrections to deltas from SPL (i.e. changes required to the SPL to meet application requirements) if they are detected 	Simple to Moderate <ul style="list-style-type: none"> • Corrections to deltas from SPL (i.e. changes required to the SPL to meet application requirements) if they are detected
Likelihood of Deltas from SPL (i.e. changes required to the SPL to meet application requirements)	Small <ul style="list-style-type: none"> • SPL is more flexible • SPL deltas mostly likely to be omissions and non-functional issues (e.g. performance requirements can't be met with current SPL artifacts). 	Moderate <ul style="list-style-type: none"> • SPL more detailed & fixed • SPL deltas most likely to be omissions, modifications & non-functional issues (e.g. performance requirements can't be met with current SPL artifacts)
Automation Potential for Product Derivation	Moderate <ul style="list-style-type: none"> • Automate the selection & instantiation of SPL artifacts and design patterns based on feature selection • Cannot automate customization process because additional work needs to be done at application derivation 	High <ul style="list-style-type: none"> • Automate the selection of SPL artifacts and classes based on feature selection

To help quantify the scalability benefits of the pattern based SPLE consider the Command Execution use case from the FSW SPL. Using a component/connector based SPLE approach the FSW SPL would require significantly more modeling during the SPL engineering phase when compared to the pattern based SPLE approach. This is because pattern based SPLE trades SPL development with application development. For example using the component/connector based SPLE approach, the interactions the FSW SPL has with all possible IO devices must be individually modeled within the control components' state machine and on interaction diagrams. A small spacecraft typically requires just the minimal amount of kernel and optional IO devices and can utilize a

centralized control approach. The FSW SPL has five optional IO components and 25 types of IO device variants. When a single parameterized centralized control component is used it would require 21 parameters, where one parameter is required each group of alternative variant devices, one parameter for each optional component, and one parameter for each variant component. The centralized control state machine must capture how it interacts with all possible IO devices and their variants for all possible ground station commands. Plug compatible interfaces can be used to help reduce the modeling effort for variant devices that use the same interface, such as the low gain antenna and medium gain antenna. However, additionally modeling is required to capture how the controller component interfaces with variants that do not have the same interface, such as torque rods and reaction control systems that are variant attitude control devices. This results in additional modeling when all possible ground commands must be captured. This causes extra modeling because ground command contain both simple commands, which require interactions on one device, and complex commands, which require interactions on multiple devices. The responses to these commands must take into account the appropriate variant device, which requires additional modeling when variant devices do not have plug compatible interfaces. Additionally, these interactions should also be modeled in interaction diagrams to show the component interactions. Furthermore, a large spacecraft typically requires all of the kernel and optional IO devices and utilizes a hierarchical or distributed control approach. When parameterized control components are used, the state machine must capture how each controller

interacts with the IO devices and variants it controls, along with all possible ground station commands it is responsible for processing. Additionally, these interactions should also be modeled on interaction diagrams to capture the component interactions.

However, when the pattern based SPLE approach is used on the FSW SPL, only a representative set of interactions and ground commands with the IO devices. Therefore the control components' state machines only need to model a subset of the interactions with the IO devices. Additionally, a smaller amount of interaction diagrams is needed to capture a representative set of object interactions. This results in significantly less modeling during the SPL phase.

During the application engineering phase, SPLE with component/connector approach the FSW SPL would require less modeling than the SPLE with design patterns approach. This is because the SPLE with component/connector approach already captures how the FSW SPL interactions will all possible IO devices and ground commands. Therefore the during the application engineering phase, the control components would just need to be configured to interact with the selected IO devices. Only the interactions with the application unique payload devices would need to be modeled.

Conversely, using the SPLE with design patterns approach the control components must be customized to interact with just the selected IO devices, payload unique devices, and

selected ground commands. While the SPLE with design patterns approach does create additional work for the application engineer, it does save a significant amount of time and effort during the SPL engineering phase since all possible IO device configurations and ground commands do not need to be individual modeled.

A summary comparison of the FSW SPL variable components you need to model at software product line engineering and application engineering for the two approaches is depicted in Table 6-5. This table illustrated that, during the SPL Engineering phase, the pattern based SPLE approach requires significantly less component modeling than the component/connector based SPLE approach. In the FSW SPL, during the SPL Engineering phase, the pattern based SPLE approach required modeling only 29 components containing representative SPL behavior, while the component/connector based SPLE approach required 53 components containing parameterized behavior for all the different SPL variants. As previously discussed, the tradeoff in a pattern based SPLE approach is additional modeling during the application engineering phases. Table 6-5 shows that during the application engineering phase of SNOE, 10 FSW SPL components are customized to the application and in STEREO 22 FSW SPL components are customized. This is compared to the component/connector based SPLE approach which does not require any customizations to the FSW SPL components.

Table 6-5 FSW SPL Variable Component Comparison for the Command Execution use case

	Pattern based FSW SPL	Component/connector based FSW SPL
Software Product Line Engineering (SPLE)	<ul style="list-style-type: none"> • CDH Centralized Controller • Attitude Control Device OC • Attitude Determination Device IC • Propulsion Thruster OC • Temperature Sensor IC • Louver OC • Heater OC • Memory Storage Device IOC • Payload Device IOC • Power Distribution Unit IOC • Power Device IOC • Power Appendage OC • Transmitter OC • Receiver IC • Antenna OC • Antenna Gimbal OC • CDH Coordinator • CDH Controller • Power Controller • Telemetry Tracking Controller • Attitude Control Controller • Thermal Controller • Payload Controller • CDH Distributed Controller • Power Distributed Controller • Telemetry Tracking Distributed Controller • Attitude Distributed Control Controller • Thermal Distributed Controller • Payload Distributed Controller 	<ul style="list-style-type: none"> • CDH Centralized Controller • Attitude Control Device OC • Reaction Wheel Assembly OC • Torque Rod OC • Reaction Control System OC • Small Thruster OC • Attitude Determination Device IC • Inertial Reference Unit IC • Star Tracker IC • Magnetometer IC • Horizon Crossing Indicator IC • Solar Tracker IC • Sun Sensor IC • Star Scanner IC • Planetary Limb Sensor IC • Propulsion Thruster OC • Temperature Sensor IC • Thermometer IC • Thermistor IC • Louver OC • Heater OC • Memory Storage Device IOC • RAM Device IOC • EEPROM IOC • Taper Recorder IOC • Payload Device IOC • Power Distribution Unit IOC • Power Device IOC • Photovoltaic IOC • Radioisotope Thermoelectric Generator IOC • Power Appendage OC • Transmitter OC • Receiver IC

		<ul style="list-style-type: none"> • Antenna OC • Low Gain Antenna OC • High Gain Antenna OC • Medium Gain Antenna OC • Movable Low Gain Antenna OC • Movable High Gain Antenna OC • Movable Medium Gain Antenna OC • CDH Coordinator • CDH Controller • Power Controller • Telemetry Tracking Controller • Attitude Control Controller • Thermal Controller • Payload Controller • CDH Distributed Controller • Power Distributed Controller • Telemetry Tracking Distributed Controller • Attitude Distributed Control Controller • Thermal Distributed Controller • Payload Distributed Controller
SPLE Component Count	<ul style="list-style-type: none"> • 29 components 	<ul style="list-style-type: none"> • 53 components
Application Engineering: SNOE	<ul style="list-style-type: none"> • CDH Centralized Controller* • Torque Rod OC • Magnetometer IC • Horizon Crossing Indicator IC • EEPROM IOC • Power Distribution Unit IOC* • Photovoltaic IOC • Transmitter OC* • Receiver IC* • Low Gain Antenna OC 	
Application Engineering: SNOE Component	<ul style="list-style-type: none"> • 10 components** 	<ul style="list-style-type: none"> • 0 components

Count		
Application Engineering: STEREO	<ul style="list-style-type: none"> • Reaction Wheel Assembly OC • Inertial Reference Unit IC • Star Tracker IC • Sun Sensor IC • Propulsion Thruster OC • Thermistor IC • Heater OC • EEPROM IOC • Power Distribution Unit IOC* • Photovoltaic IOC • Power Appendage OC* • Transmitter OC* • Receiver IC* • Low Gain Antenna OC • Medium Gain Antenna OC • Movable High Gain Antenna OC • CDH Coordinator* • CDH Controller* • Power Controller* • Telemetry Tracking Controller* • Attitude Control Controller* • Thermal Controller* 	
Application Engineering: STEREO Component Count	<ul style="list-style-type: none"> • 22 components** 	<ul style="list-style-type: none"> • 0 components

* SPLE component that is customized to the application

** Component count includes SPLE components that are customized to the application

7 APPLICATION ENGINEERING

7.1 Introduction

This chapter describes the application engineering process for deriving applications from SPL with design pattern level variability. Application engineering was performed on two case studies. The case studies included the Student Nitric Oxide Explorer (SNOE) and Solar TERrestrial RELations Observatory (STEREO). However, this chapter only describes process is illustrated using the SNOE. The details on the STERO case study are found in Appendix D.

7.2 Problem Description

SNOE, which was a real-world, small satellite program funded by the National Aeronautics and Space Administration (NASA) and managed by the Universities Space Research Association (USRA). The information used in this case study is found in (Scott Bailey et al. 1996; Mark A. Salada & Randal Davis 1995; Mark A. Salada et al. 1998; Laboratory For Atmospheric and Space Physics at the University of Colorado at Boulder n.d.; Stanley Solomon et al. 1998; Stanley Solomon et al. 1996). Whenever possible,

requirements and information from the real-world case study were used. However, when information was not available, assumptions were made based on personal experience in the flight software domain.

SNOE's mission involves using a spin stabilized spacecraft in a low earth orbit to measure thermospheric nitric oxide (NO) and its variability. The SNOE spacecraft is spin stabilized, meaning it maintains its orientation similar to that of a top. SNOE is required to maintain a spin rate of 5 rotations per minute (RPM). The spin rate can be adjusted having the FSW send a command to commutate the electromagnet transverse torque rod. The spin axis direction is controlled in a similar fashion by having the FSW send a command to commutate the electromagnet spin axis torque rod. SNOE's FSW does not perform the attitude determination and control calculations. Rather, the FSW collects the attitude measurements and downlinks them to the ground for processing. Then the ground uplinks attitude control commands back to the spacecraft for the SNOE FSW to execute. The attitude measurements are taken from two horizon crossing indicators (HCI) and three magnetometers.

SNOE's spacecraft body is surrounded on all sides by stationary solar panels which are used to generate power, as seen in Figure 7-1. Since the solar panels are stationary, the FSW does not interact with them. The power generated from the solar panels is stored in

batteries and the charge is controlled by the FSW. The thermal control onboard the spacecraft is completely passive and therefore it does not interact with the FSW.

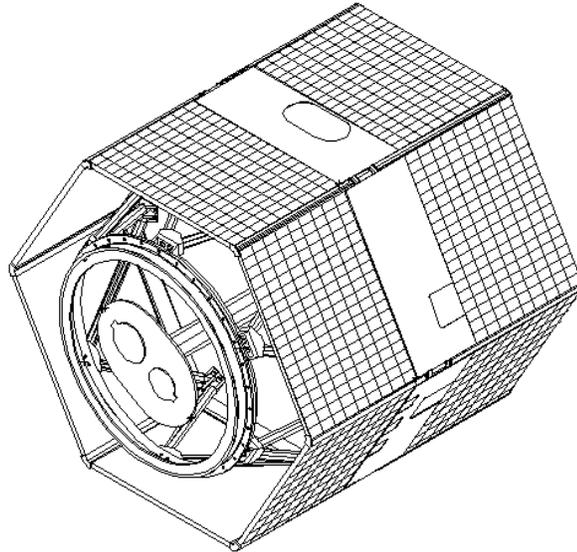


Figure 7-1 SNOE Spacecraft Structure (Laboratory For Atmospheric and Space Physics at the University of Colorado at Boulder n.d.)

The spacecraft contains three payload instruments to accomplish its scientific mission. These three instruments are an ultraviolet spectrometer (UVS) that measures NO density, an auroral photometer (AP) that measures the flux of energetic electrons entering the Earth's upper atmosphere, and a solar soft X-ray photometer (SXP) that measures the solar irradiance. The payload instruments are mounted on a shelf inside the spacecraft and therefore spin in correlation with spacecraft. The spacecraft and instruments spin

continuously and collect data constantly thus requiring minimal commanding from the FSW. The payload instruments collect critical and necessary data during the segment of their spin when the spacecraft and instruments are facing Earth. The data collected when the payload instruments are pointed away from Earth is discarded. During the spacecraft's rotation, the FSW must command the payload instruments to store the desired data in their instrument data buffers. The FSW also must empty each of the instrument data buffers at least once per spin and format and store the payload data in telemetry packets. However, the collection frequency is controlled by the ground station and is subject to change.

Additionally, SNOE also contains a microGPS Bit-Grabber Space Receiver (microGPS BGSR) instrument as a technology experiment. The microGPS BGSR gathers position information based on the Global Positioning System (GPS) constellation for experimental orbital determination. The microGPS BGSR utilizes a large amount of power, however its data is only required three times per orbit. Therefore to conserve spacecraft power, the microGPS BGSR is turned on by a hardware timer for a few seconds to take a measurement and then placed in idle mode. The FSW must empty the microGPS BGSR data buffer once per orbit as well as format and store the data in telemetry packets.

In addition to collecting science data and attitude control data, the SNOE FSW must also periodically collect health and status housekeeping data from the hardware. The FSW stores this data and sends it to the ground for processing and analysis.

The SNOE spacecraft orbits the Earth in a circular 550km orbit. SNOE communicates with the NASA Transportable Orbital Tracking Station (TOTS) ground station in Poker Flats, Alaska. SNOE's FSW must format and downlink all spacecraft data to the ground for processing. During normal operations only two contacts are made per day. The ground station is responsible for payload data processing, attitude determination, and monitoring of the spacecraft and instrument's long term health and safety. When necessary, flight controllers and engineers can uplink commands back to the SNOE to make orbital adjustments or changes to the instruments and hardware.

All data and commands exchanged between SNOE and the ground station are formatted using Consultative Committee for Space Data Systems (CCSDS) standards. The FSW must format all spacecraft data in this format. SNOE contains two telemetry channels; one low-rate channel (512 bps) for real-time data and one high-rate channel (128k bps) for playback and real-time data. To accommodate using the high-rate channel for playback and real-time data, SNOE uses the virtual channels feature of CCSDS packets. To avoid wasting bandwidth by allowing unused portions of the CCSDS frames to go empty, SNOE is also required to use source packets, which are programmable packets

that can overlap frames. The SNOE FSW is responsible for determining when and what data should be used in the source packets.

SNOE contains a single flight computer, which is the SC4A Single Board Spaceflight Computer developed by Southwest Research Institute (SwRI). It has a 16-bit 80C186 central processor with a clock frequency of 10 MHz. The throughput capacity of SNOE's flight computer is estimated at 0.77 million instructions per second (MIPS). The flight computer also comes equipped with 512K Bytes RAM, 256K Bytes EEPROM, and 64K Bytes UVPROM. Additionally, 8M Bytes of EEPROM is also provided for storing approximately 24hrs of telemetry. SNOE's flight computer also has 32 12-bit analog input channels, 4 12-bit analog output channels, a two channel RS-422 Serial I/O port, and 16 input/output Parallel I/O. The computer uses memory mapped I/O, providing 16 bit word reads or writes per instruction cycle. An ACTEL ACT1 FPGA is also used to handle in interfacing to all spacecraft's hardware. Additionally, each of the payload devices contains their own memory storage devices.

The SNOE spacecraft has limited hardware redundancy because it is a small, low cost satellite with a short lifespan. The only redundant hardware onboard is the transmitter, receiver, and battery. SNOE's FSW relies heavily on the ground to maintain spacecraft health. Therefore it has limited autonomous fault management functionality. The autonomous fault management functionality includes detecting if there is a problem with

the transmitter, receiver, or battery. If a problem is detected, the FSW should switch to the redundant hardware and notify the ground during its next contact.

7.3 Application Engineering

The application engineering process involves building a specific SPL member from the SPL assets. This process involves selecting the kernel and variable functionality that meet the requirements of the SPL member. The application engineer process involves three phases which are the requirements, analysis, and design modeling. Detailed descriptions of the SPL engineering phases are described in the subsequent subsections.

7.4 Application Requirements Modeling

The first step in application engineering process is to perform requirements modeling to scope the application. During the requirements phase the appropriate features, use cases, and use case activity paths are selected from the feature model, use case model and use case activity model, respectively. This selection process is driven by the SPL member's requirements. The application requirements engineering phase is illustrated using SNOE, which is a small spacecraft SPL member.

7.4.1 Application Feature Modeling

First, the appropriate features from the feature model are selected for the application. This process is demonstrated using SNOE and the FSW Command and Data Handling (C&DH) Feature Model described in Chapter 6. The final C&DH pattern specific feature model for SNOE is depicted in Figure 7-2, where the features selected are highlighted using a bold font and a gray background. By definition, SNOE must provide all of the

FSW SPL C&DH kernel pattern specific features. There is only one kernel feature which is the Memory Storage Device Fault detection feature.

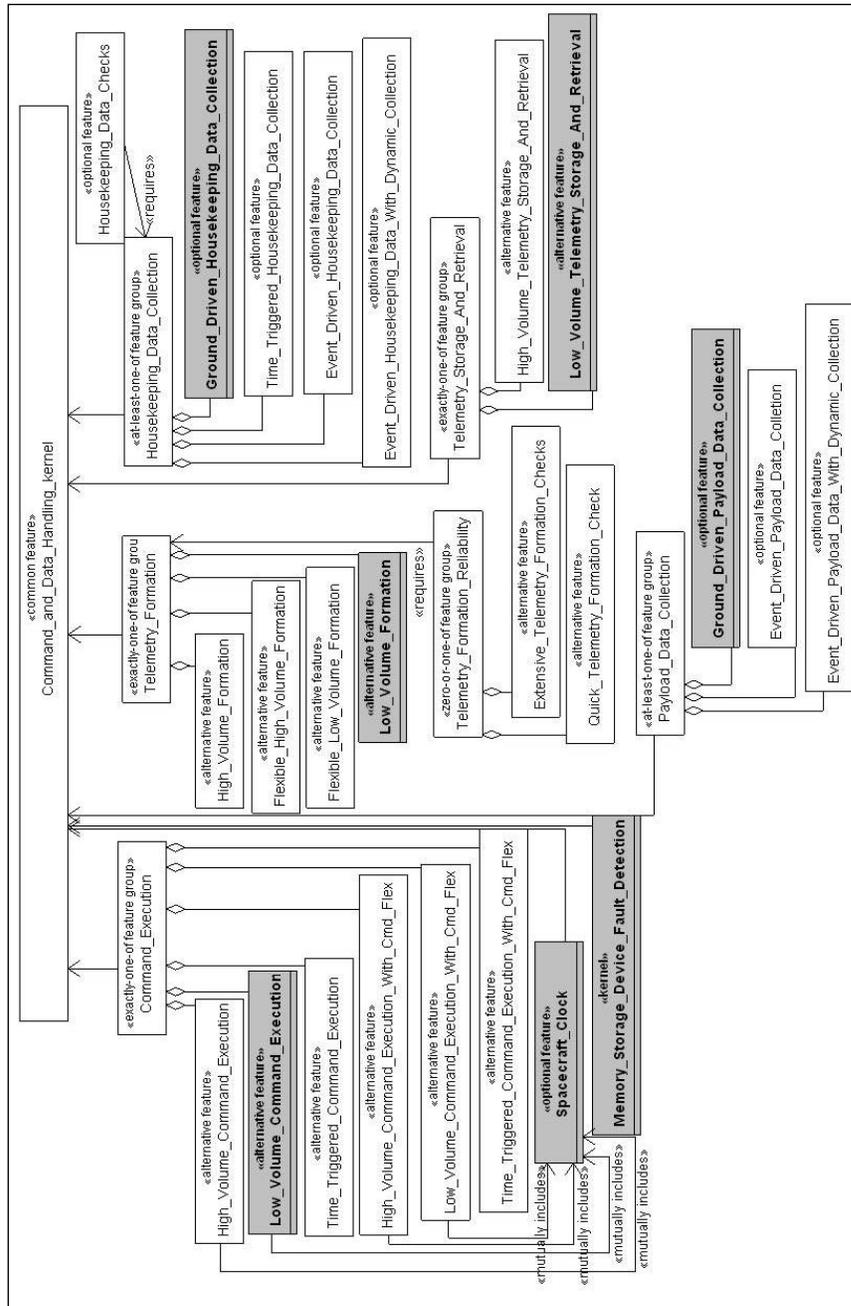


Figure 7-2 SNOE C&DH pattern specific feature selection

Additionally, SNOE must also provide one feature from the exactly-one-of pattern specific feature groups. The FSW SPL C&DH feature model contains three of these feature groups. The feature selection from each of these groups is described below.

- **Command Execution.** SNOE is a small spacecraft that only requires a small amount of commands to process. Additionally, it does not need the ability to support new commands or strict temporal predictability. Therefore Low Volume Command Execution alternative feature is the best choice for SNOE.
- **Telemetry Formation.** SNOE's payload instruments only produce a small amount of payload data and the amount of housekeeping data produced by SNOE's hardware is minimal. Thus SNOE is only required to format a low volume amount of data. As described in section 7.2, SNOE follows the CCSDS standard. Due to SNOE's short lifespan, changes to the CCSDS standard are not anticipated and flexible telemetry formatting is not required. Therefore the Low Volume Telemetry Formation feature is selected.
- **Telemetry Storage and Retrieval.** SNOE's payload instruments only produce a small amount of payload data and the amount of housekeeping data produced by SNOE's hardware is minimal. Thus SNOE is only required to store a low volume of data. Therefore the Low Volume Telemetry Storage and Retrieval alternative feature is selected.

Next, SNOE must also provide one or more features from the at-least-one-of pattern specific feature groups. The FSW C&DH feature model contains two of these feature groups. The feature selection from each of these groups is described below.

- **Payload Data Collection.** According to SNOE's description in section 7.2, the collection of data from all the payload devices is ground driven. Thus only the Ground Driven Payload Data Collection feature is selected from this group.
- **Housekeeping Data Collection.** According to SNOE's description in section 7.2, the collection of all housekeeping data is ground driven. Thus only the Ground Driven Housekeeping Data Collection feature is selected from this group.

Next, the optional pattern specific features that SNOE should provide are determined. In the FSW SPL C&DH feature model there are two optional features. The optional features are Spacecraft Clock and Housekeeping Data Checks. According to the SNOE description, all data monitoring is performed at the ground station thus the Housekeeping Data Checks feature is not selected. However, SNOE's Low Volume Command Execution feature mutually includes the Spacecraft Clock pattern specific feature. Therefore, SNOE must provide the optional Spacecraft Clock pattern specific feature.

Finally, the last step in SNOE's feature model is to determine the optional C&DH pattern specific feature groups that SNOE should provide. Optional pattern specific feature groups include the zero-or-one or zero-or-more feature groups. In the FSW C&DH feature model there is one optional feature groups, which is Format Telemetry Reliability.

According to the SNOE description in section 7.2, the SNOE does not have the responsibility to detect faults in the telemetry processing. Thus no features are selected from this group.

Next, the pattern variability features that SNOE realizes are selected from the FSW SPL. The final C&DH pattern variability feature model for SNOE is depicted in Figure 7-3, where the features selected are highlighted using a bold font and a gray background. By definition, SNOE must provide one feature from the FSW SPL C&DH exactly-one-of pattern variability feature groups. The FSW SPL C&DH feature model contains three of these feature groups. The feature selection from each of these groups is described below.

- **Memory Storage Device Type.** According to SNOE's description, all telemetry data is stored in 8M of EEPROM. Therefore the EEPROM Device feature is selected
- **Power Device Type.** SNOE's spacecraft body is surrounded on all sides by stationary solar panels which are used to generate power. Therefore the photovoltaic feature is selected.
- **Attitude Control Device Type.** SNOE is a spin stabilized spacecraft. According to its description the spin rate is adjusted by commutating an electromagnet transverse torque rod. Thus the Torque Rod feature is selected.

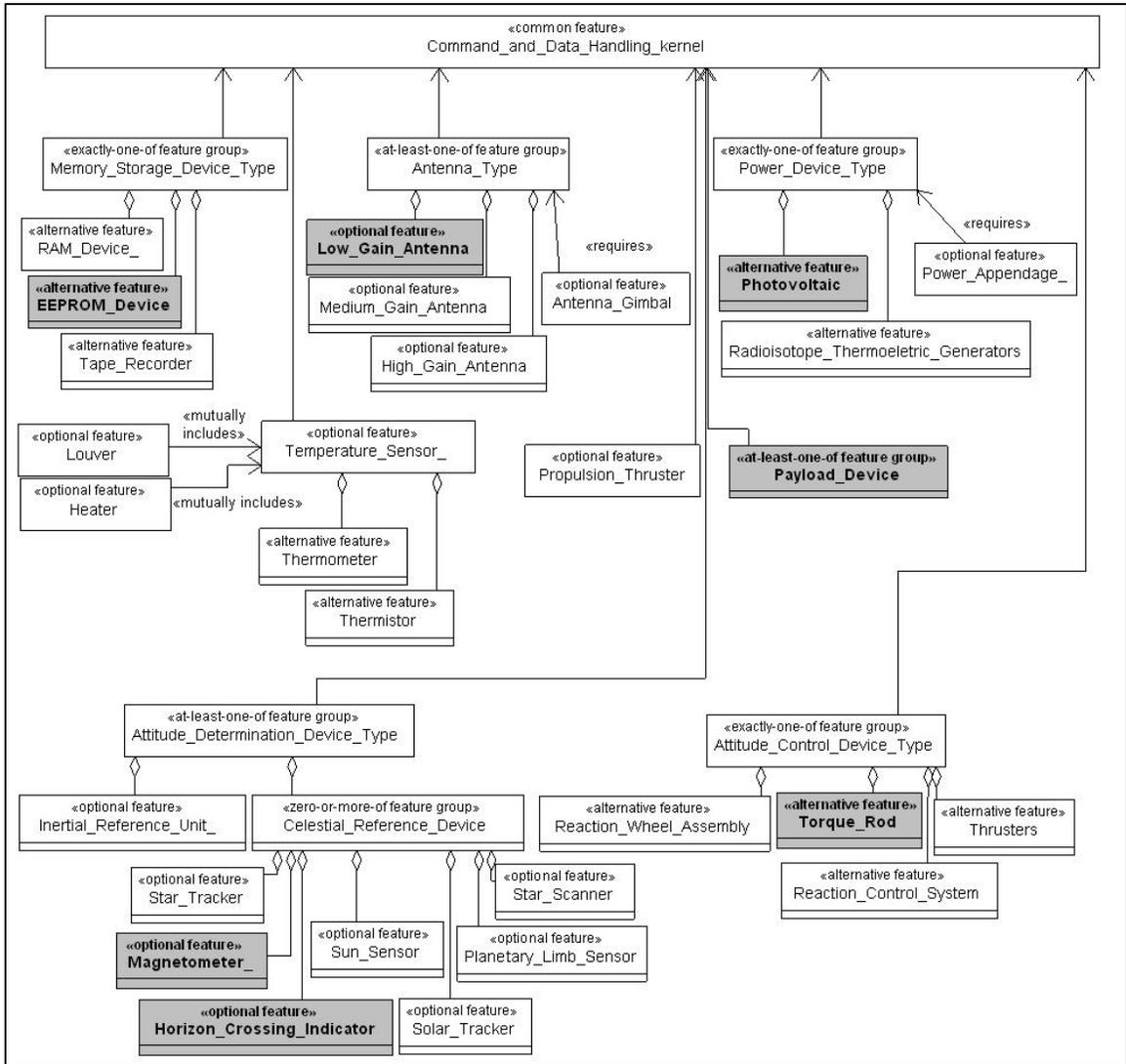


Figure 7-3 SNOE pattern variability feature selection

Next, SNOE must also provide one or more features from the at-least-one-of pattern specific feature groups. The FSW C&DH feature model contains two of these feature groups. The feature selection from each of these groups is described below.

- **Antenna Type.** SNOE contains a single low gain antenna to communicate with the ground station. Therefore only the Low Gain Antenna feature is selected from this group.
- **Attitude Determination Device Type.** According to SNOE's description, attitude measurements are taken from two horizon crossing indicators (HCI) and three magnetometers. Therefore the both the Magnetometer and Horizon Crossing Indicator features are selected.

Next, the optional pattern variability features that SNOE should provide are determined.

In the FSW SPL C&DH pattern variability feature model there are five optional features.

- **Propulsion Thruster.** According to the SNOE description, the spacecraft is placed in orbit and does not require maneuvering. Therefore this feature is not selected.
- **Antenna Gimbal.** SNOE utilizes an immobile low gain antenna (LGA). Therefore the antenna gimbal is not required or selected.
- **Power Appendage.** According to SNOE's description, its solar panels are stationary, thus this feature is not selected.
- **Heater and Louver.** As described in SNOE's description in section 7.2, active thermal control of the spacecraft is not required, thus these features are not selected.

Finally, the optional pattern variability feature groups that SNOE should provide are identified. Optional pattern specific feature groups include the zero-or-one or zero-or-more feature groups. In the FSW C&DH feature model there is one optional feature groups. This feature group relates to optional components that are needed if the FSW is

required to controls its temperature. As described in SNOE's description in section 7.2, active thermal control of the spacecraft is not required, thus no features are selected.

7.4.2 Application Use Case Modeling

Next, the appropriate use cases from the use case model are selected for the application.

This subsection describes the use case model for the SNOE SPL member. The use case model for SNOE is derived from the FSW SPL use case model defined in Chapter 6.

The final use case model for SNOE is depicted in Figure 7-4, where the use cases and actors selected are highlighted using a bold font and a gray background. By definition, SNOE must provide all of the kernel use cases. These include the Collect & Store Data, Execute Commands, Perform Fault Management, and Uplink and Downlink Telemetry.

SNOE must also interface will all the kernel actors. The optional use cases are not selected for SNOE based on SNOE's feature selection and description.

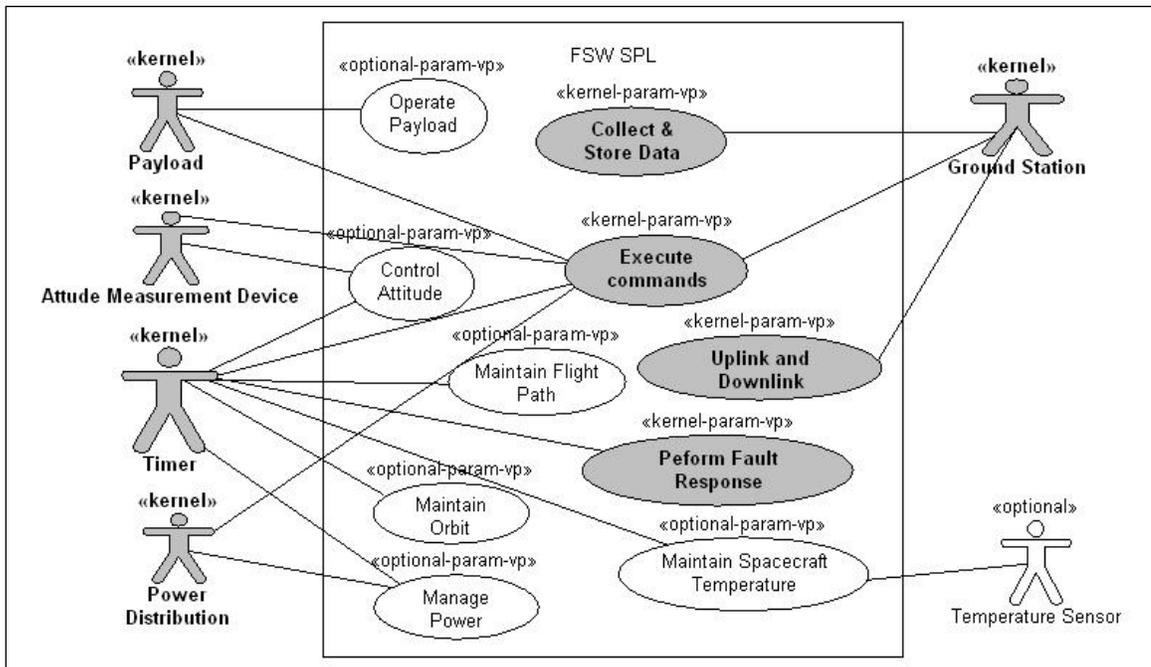


Figure 7-4 SNOE use case model

7.4.3 Application Use Case Activity Modeling

Next, the appropriate use case activity paths from the use case model are selected for the application. This is accomplished by only utilizing the feature based condition paths that corresponds to SNOE’s feature selection. This process is demonstrated using the SNOE’s Execute Commands use case, however, additional use cases are also demonstrated in Appendix C. This use case involves executing commands from the ground station to ensure the spacecraft is not put into an unsafe state and the actions are appropriate for the spacecraft’s mode. SNOE’s activity diagram for this use case is

depicted in Figure 7-5. SNOE selected the Low Volume Command Execution feature, thus the path that corresponds to this feature selection is taken as seen in Figure 7-5.

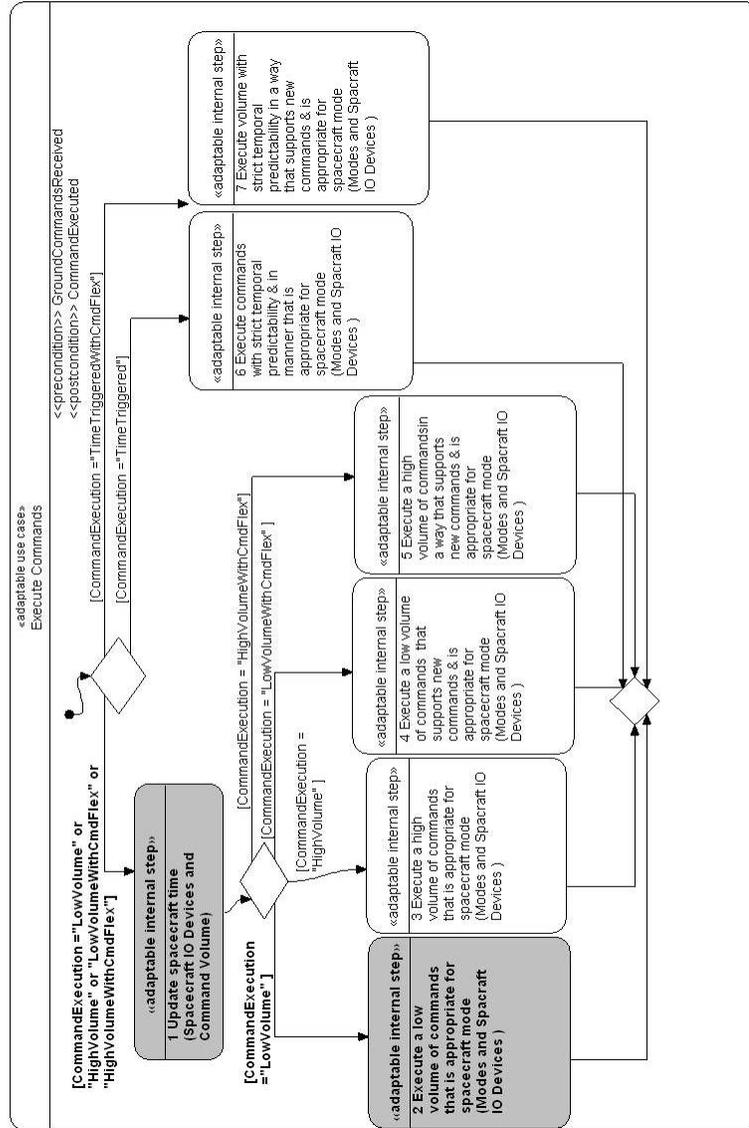


Figure 7-5 SNOE Execute Commands use case activity model

7.5 Application Analysis Modeling

After the requirements modeling phase is complete, the next step is to perform the analysis modeling. The goal of the analysis modeling phase is to develop the basic structure for the application's architecture. In this phase the conceptual static model, context model, subsystem dependency model, design pattern selection, and application specific architectural and executable design patterns are derived from the SPL assets. The process for deriving these assets is described below in more detail using the SNOE spacecraft as an example.

7.5.1 Application Conceptual Static Modeling

The application conceptual static modeling involves selecting just the appropriate devices from the SPL conceptual static model based on the applications features. This process is illustrated using SNOE. The conceptual static model for SNOE is depicted below in Figure 7-6, where the optional and variant devices selected are highlighted using a bold font and a gray background. By definition, SNOE must provide the kernel devices. Therefore the transmitter, receiver, payload device, and power distribution unit (PDU) is selected from the SPL conceptual static model, as seen in Figure 7-6.

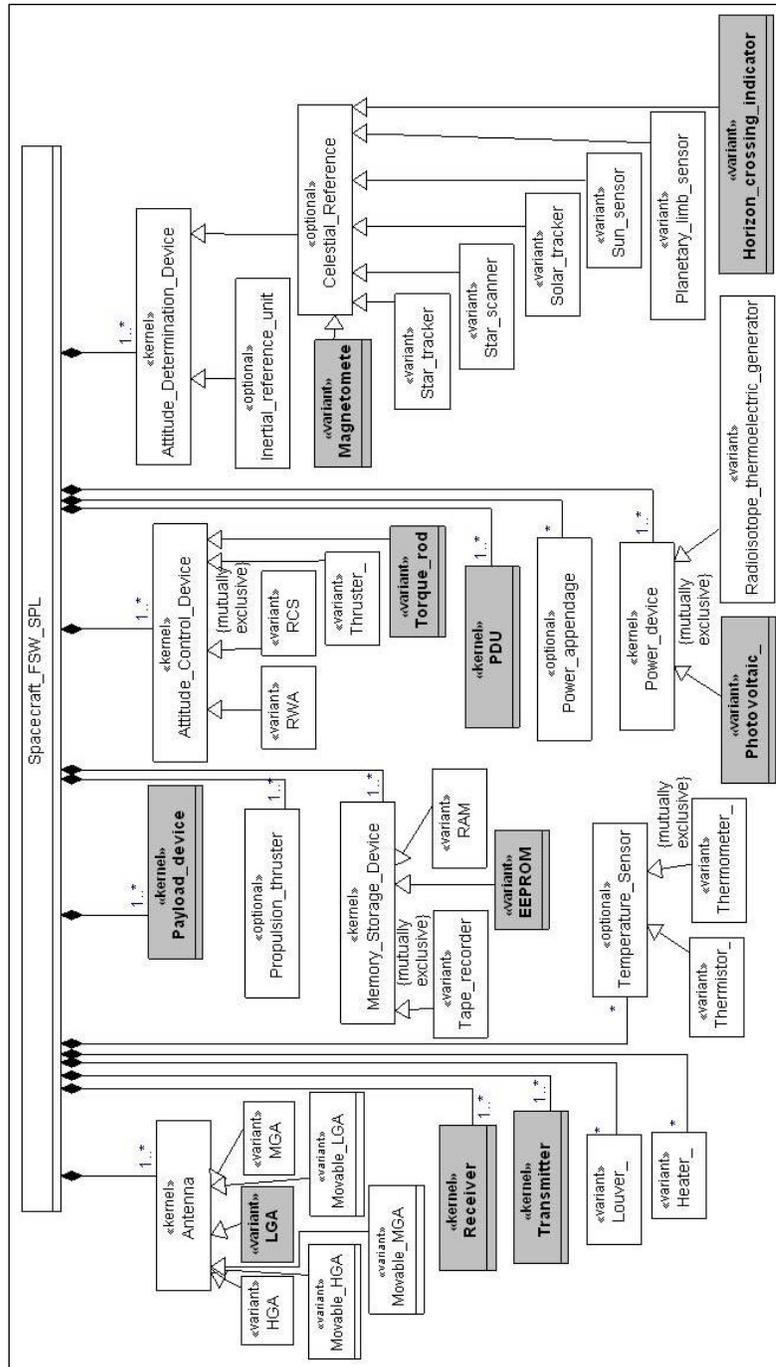


Figure 7-6 SNOE FSW conceptual static model

Next, the optional devices and variant devices are selected based on SNOE's feature selection. First, SNOE did not select any of the Temperature Control, Heater, Louver, Power Appendage, Antenna Gimbal, or Propulsion Thruster pattern variability features, thus the temperature control devices, heater, louver, power appendage, antenna gimbal and propulsion thruster physical devices associated with these features are not selected. Next, the variant kernel devices are selected based on SNOE's pattern variability feature selection. SNOE selected the Low Gain Antenna (LGA), EEPROM, Photovoltaic, Magnetometer, Horizon Crossing Indicator, and Torque Rod pattern variability features. Thus the physical devices associated with this feature selection are selected, as seen in Figure 7-6.

Finally, since the specific variants for the payload device were not defined in the FSW SPL, the SNOE specific variants must be defined. According to SNOE's description, SNOE utilizes four science instruments to achieve its mission. Therefore a payload variant for each of these devices must be created.

7.5.2 Application Context Modeling

Another major artifact produced from the application analysis modeling phase is the context diagram. The context diagram is used to define the system's boundary with the external environment. It is derived from the SPL context diagram, where only the optional and alternative devices applicable to the application are selected. This process is illustrated using SNOE's context diagram which is derived from the FSW SPL context

diagram described in Chapter 6. Figure 7-7 depicts the SPL context diagram with the variable devices selected for SNOE, based on its feature selection. The variable devices selected are highlighted using a bold font and a gray background. Figure 7-7 shows that SNOE does not interface with power appendage, antenna gimbal, or propulsion thrusters. This is because SNOE did not select the Antenna Gimbal, Propulsion Thruster or Power Appendage pattern variability features. Additionally, SNOE did not select any of the pattern specific or pattern variability features involving thermal control therefore it does not interface with the temperature control devices or temperature sensors.

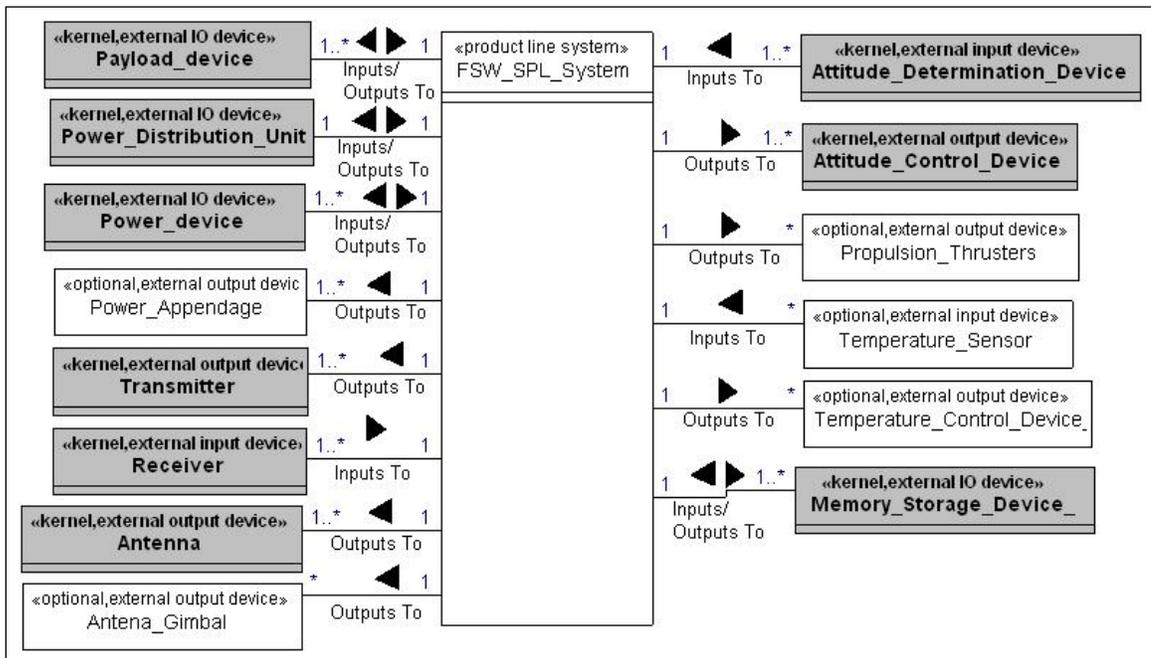


Figure 7-7 SNOE FSW system context diagram

Next, the SNOE context diagram can be refined to depict the actual variant devices that SNOE uses based on its pattern variability feature selection. Figure 7-8 illustrates the final system context diagram for SNOE. Finally, SNOE contains four payload devices, which are the auroral photometer, microGPS, solar x-ray photometer, and ultraviolet spectrometer. Therefore these SNOE specific variant payload devices added and depicted in Figure 7-8.

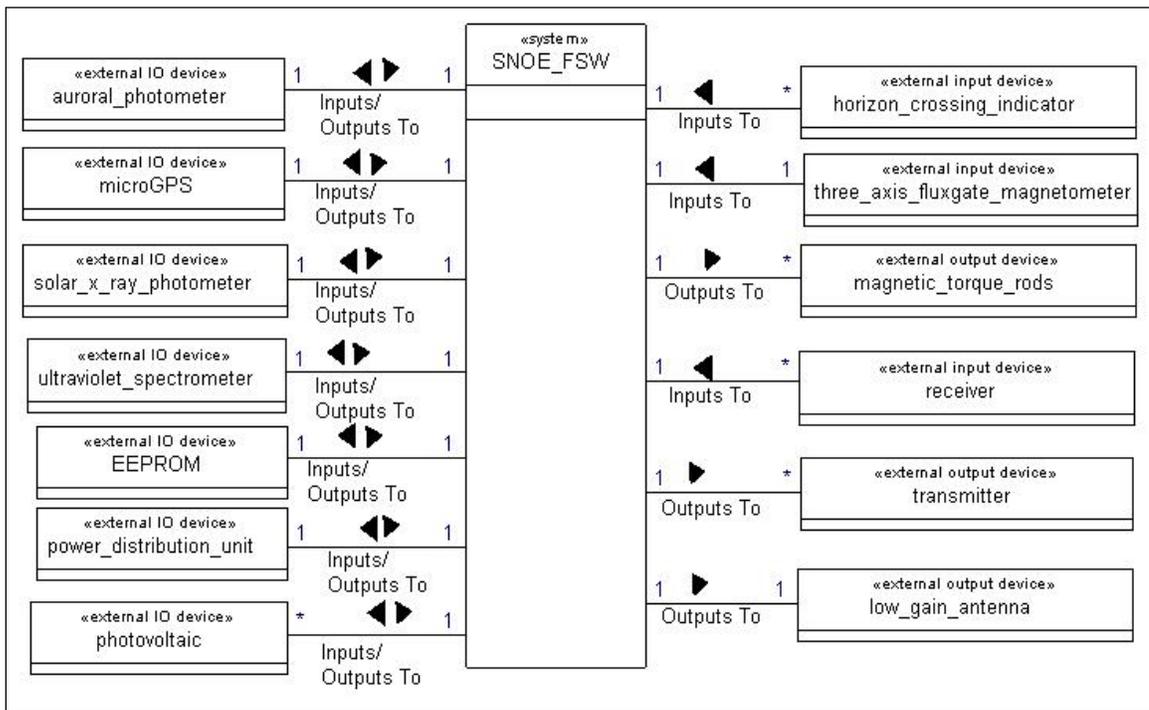


Figure 7-8 SNOE FSW context diagram

7.5.3 Application Subsystem Structuring

Another key artifact from the analysis modeling phase is the subsystem structuring model. It is derived from the SPL subsystem structuring, where only the optional and alternative subsystems applicable to the application are selected. This process is illustrated using SNOE's subsystem structuring model, which is identified from the FSW SPL subsystem structuring described in Chapter 6. The major relationships between SNOE's subsystems are shown in Figure 7-9, where the realized subsystems are highlighted using a bold font and a gray background.

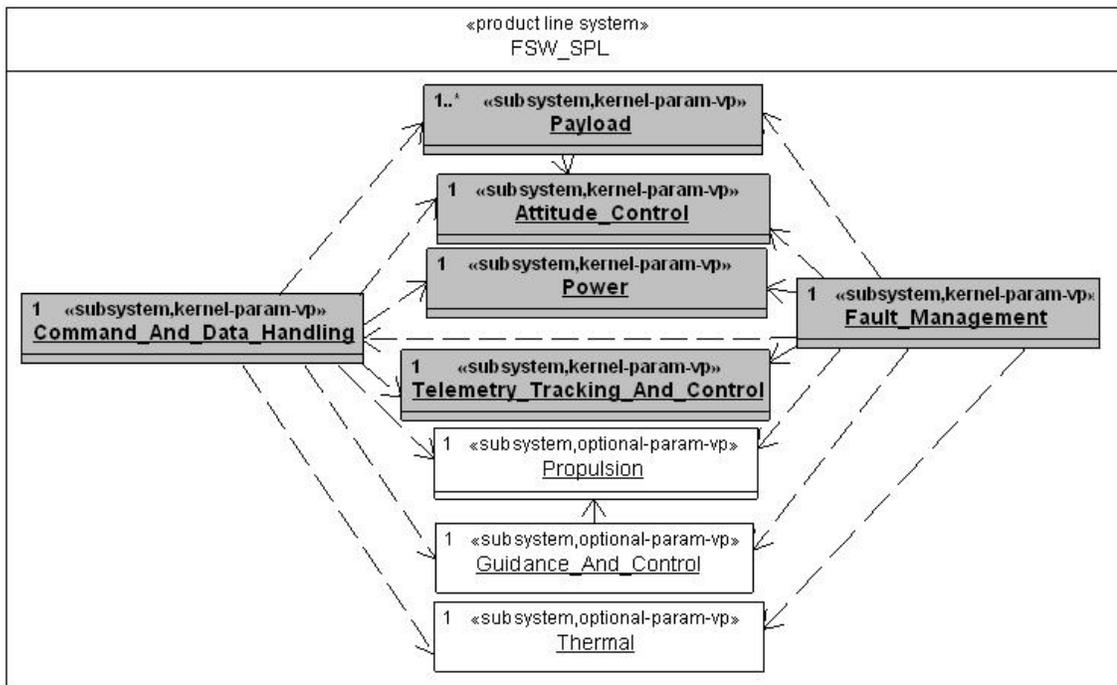


Figure 7-9 SNOE FSW subsystem structuring

SNOE must provide the five kernel subsystems; therefore it has Command and Data Handling, Telemetry Tracking and Control, Attitude Control, Payload, Power, and Fault Management subsystem. It is clear from SNOE's feature selection that it does not provide active thermal control, active propulsion, or guidance and control. Thus the Thermal, Propulsion and Guidance and Control subsystems are not selected for SNOE.

Another view of the subsystem structuring is a refinement of the context diagram, which is depicted in Figure 7-10. This diagram shows the subsystems along with the allocation to SNOE's actual physical devices.

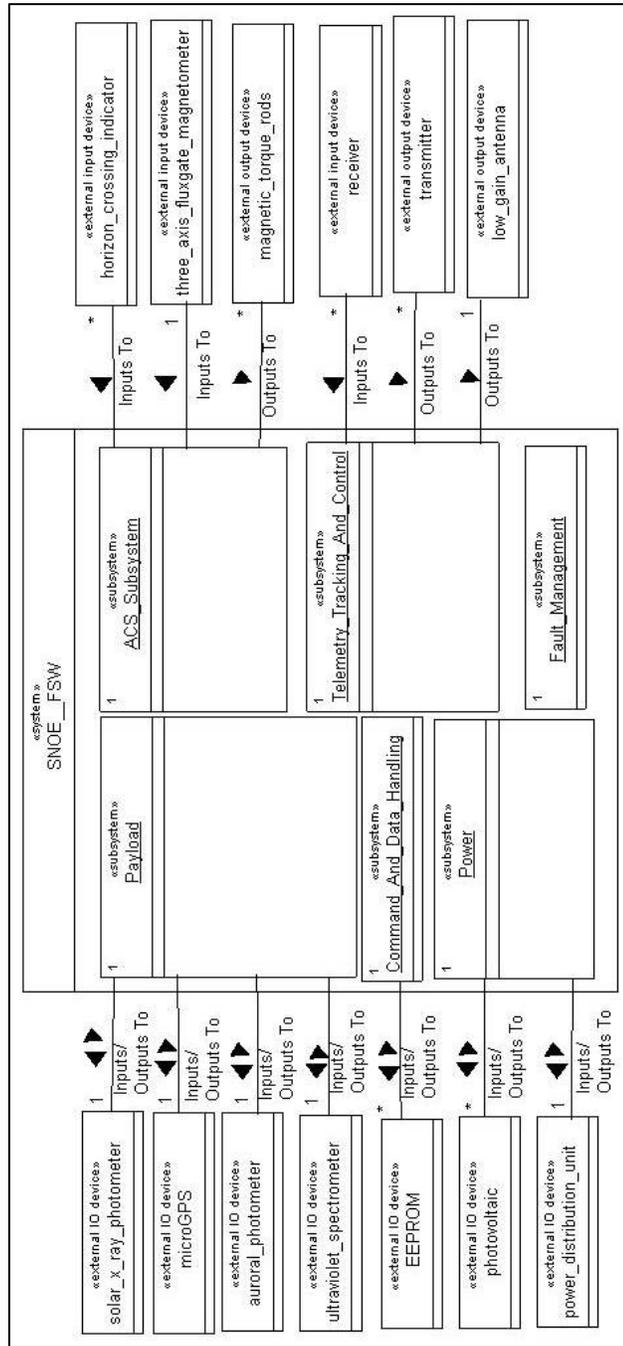


Figure 7-10 Subsystem allocation to physical devices

7.5.4 Application Design Pattern Selection

Next, the SPL design patterns that support the application's pattern specific features are determined from the SPL feature to design pattern mapping captured in Chapter 6. This process is illustrated using SNOE. Table 7-1 depicts SNOE's feature to design pattern mapping, which only lists the pattern specific features that SNOE selected.

Table 7-1 SNOE Design Pattern Selection

Feature Group	Variability	Feature	Design Pattern
<<exactly-one-of feature group>> Command Execution	alternative	Low Volume Command Execution	FSW Centralized Control
<<exactly-one-of feature group>> Telemetry Storage and Retrieval	alternative	Low Volume Telemetry Storage and Retrieval	FSW Telemetry Client Server
<<exactly-one-of feature group>> Telemetry Formation	alternative	Low Volume Telemetry Formation	FSW Pipes and Filters
<<at-least-one-of feature group>> Payload Data Collection	optional	Ground Driven Payload Data Collection	FSW Payload Multiple Client Multiple Server
<<at-least-one-of feature group>> Housekeeping Data Collection	optional	Ground Driven Housekeeping Data Collection	FSW Housekeeping Multiple Client Multiple Server
N/A	optional	Spacecraft Clock	FSW Spacecraft Clock Multicast
N/A	kernel	Memory Storage Device Fault Detection	FSW Memory Storage Device Watchdog

7.1.1 Application Architectural and Executable Design Patterns

The next step in the analysis modeling phase is to customize the SPL level architectural and executable design patterns to become application level architectural and executable

design patterns. The process for creating the application specific level architectural and executable design patterns is described in section 5.4.3. This process is demonstrated using SNOE's Centralized Control design pattern. Additional patterns are also demonstrated in Appendix C.

7.5.4.1 SNOE Centralized Control Architectural Design Pattern

SNOE utilizes the FSW SPL Centralized Control design pattern to execute commands and control the overall operation of the spacecraft. First, the collaboration diagram is customized to reflect the application specific components. As discussed in Chapter 5, this involves removing optional components that are not realized, updating the component multiplicities and selecting the appropriate variants based on the application's features. In some cases, variants are unique to the application and therefore must be defined at the application level. The Centralized Control executable design pattern collaboration diagram for SNOE is shown in Figure 7-11. Based on SNOE's feature selection, it is determined that none of the optional components are utilized and therefore are removed. Additionally, the SNOE specific variants are selected based on SNOE's pattern variability feature selection. For example, instead of using `Attitude_Control_Device_OC`, the specific type of attitude control device variant is reflected with the `Torque_Rod_OC`. SNOE uses two torque rods, thus its multiplicity is one or many. Finally, SNOE's unique payloads variants are created and included.

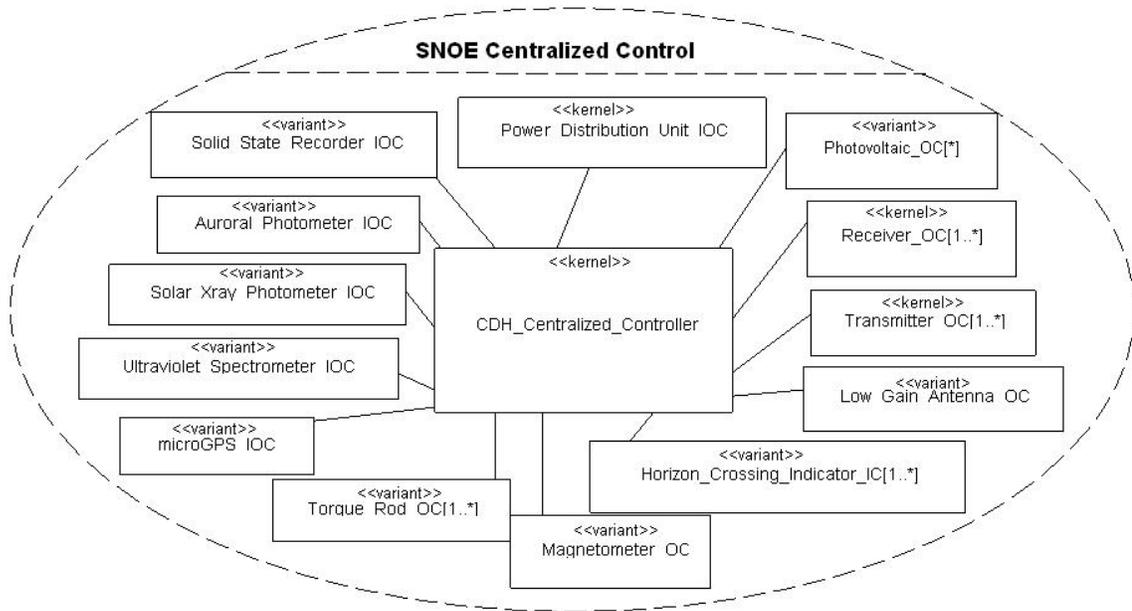


Figure 7-11 SNOE Centralized Control communication diagram

Next, since the object interactions from the FSW SPL Centralized Control executable design pattern only contained a representative set of interactions, it must also be customized for SNOE. As discussed in Chapter 5, this involves using the SNOE specific variants and object interactions. A sequence diagram for SNOE to reinitialize its low gain antenna based on a ground command is shown in Figure 7-12. It can be seen from Figure 7-12 that the interaction diagram is SNOE specific because it contains the SNOE specific input, output, and IO interface components.

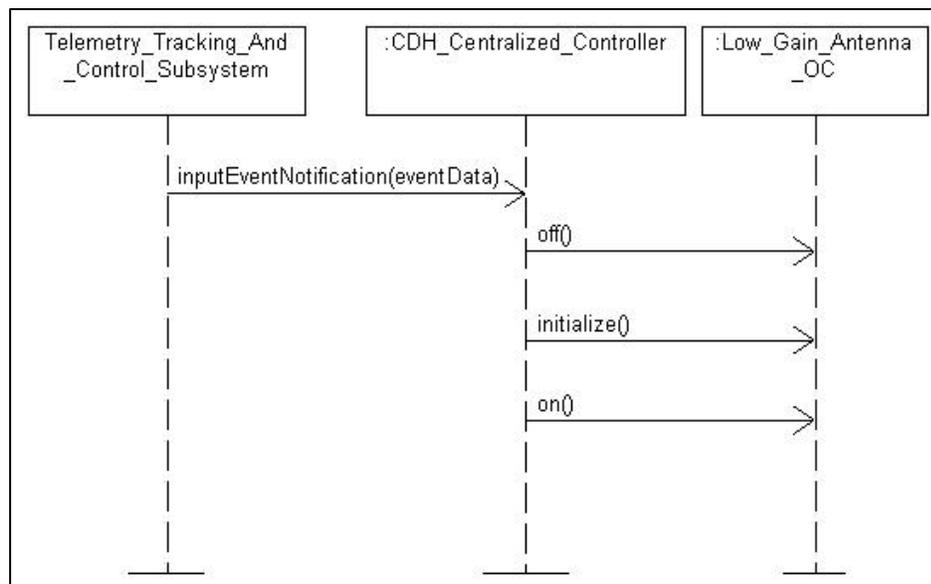


Figure 7-12 Execute Commands Scenario for SNOE

Finally, the last architectural view customized is the component diagram. As discussed in Chapter 5, this is accomplished by updating the diagram with the selected variants and option devices passed on SNOE’s feature selection. Additionally, the ports and interfaces for the payload variants that are unique to SNOE are modeled since they were not captured in the FSW SPL. The component diagram for SNOE’s Centralized Control component diagram is shown in Figure 7-13, which contains the SNOE specific variants based on SNOE’s feature selection. The ports, interfaces, and connectors for the common variants were captured in the FSW SPL. Customizations to this diagram include removing any FSW components that are not used by SNOE such as the

Temperature_Sensor_OC. Additionally, the payload variants are unique to SNOE and thus are created by the application engineer. SNOE contains four payload devices therefore four payload device variants are created from the Payload_IOC base class. For each payload variant, the port name is updated to reflect the specific payload, such as the microGPS_IOC. The port's interface is updated to reflect the specific actions that can be invoked on that payload. However, since these devices are part of the Payload subsystem and not the Command and Data Handling (C&DH) subsystem, their interfaces were not updated as part of this research.

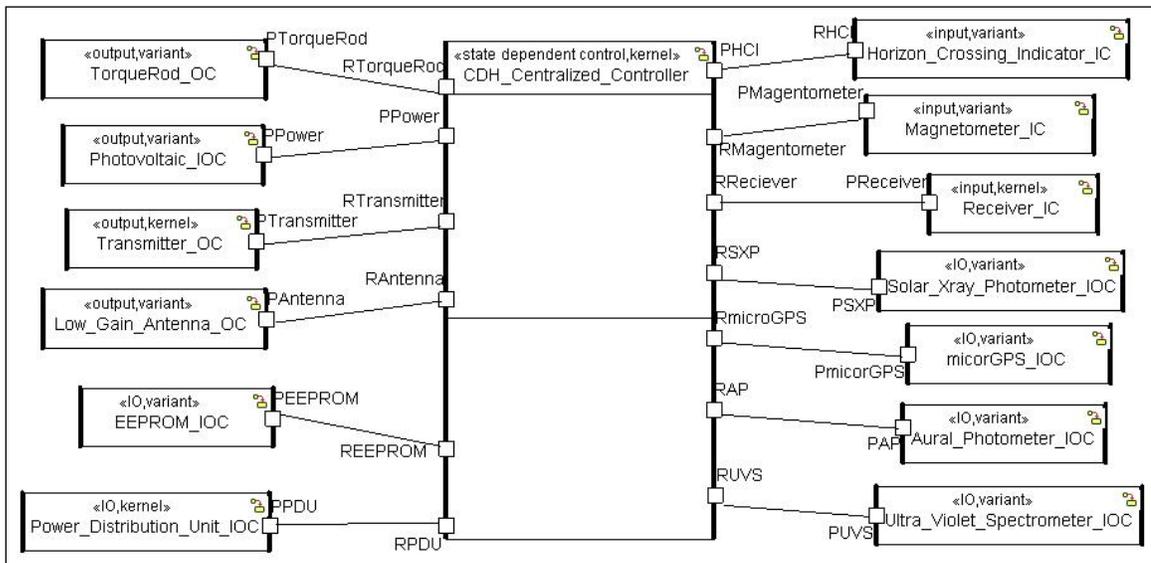


Figure 7-13 Component diagram for SNOE Centralized Control executable design pattern

7.5.4.2 SNOE Centralized Control Executable Design Pattern

Next, the executable version of the design pattern must also be updated for the SNOE. As described in Chapter 5, this involves potentially adding application specific states, actions, and activities to the SPL level state machines based on the application's features. For example, if the application features refine some behavior, then this can be modeled as substates. Also, if the component must send a message to an application specific variant or if application specific logic is required then this is modeled as an action or activity within a state or transition. First, the state machine for the SNOE's CDH_Centralized_Controller, which is a customized version of the FSW SPL's CDH_Centralized_Controller, is updated. SNOE's features do not refine the overall behavior of the component, thus the states do not require any customizations. Therefore the state machine is structurally the same as the FSW SPL state machine in Chapter 6. However, SNOE's features do require application specific logic to validate commands and to determine the appropriate response to commands. Additionally, the CDH_Centralized_Controller must interface with SNOE unique variants. Therefore these actions are updated in the appropriate states in the state machine. For example, the On Entry actions of the Executing_Command state should be modified to include SNOE specific responses to commands. Therefore when SNOE receives command to open the solar x-ray photometer door, it knows the precise operations to invoke on the Solar_Xray_Photometer_IOC. SNOE specific logic is added as actions in the states of

the CDH_Centralized_Controller. Due to the large amount of logic that is added to the CDH_Centralized_Controller state machine, this information is not depicted graphically.

Next the state machine for the SNOE's Low_Gain_Antenna_OC variant is examined.

Based on SNOE's feature selection, it does not require application-specific changes to this variant. Thus, the state machine defined for this component in the SPL is used without any modifications.

Finally, the state machines for the other variant input, output, and IO components, including the newly added payload variants, should be examined. However, since they fall into other subsystems, they are updated as part of this research.

7.5.5 Application Design Pattern Interconnection Modeling

The step in the analysis modeling phase is to customize the design pattern interconnections to the application using interaction overview diagrams. This involves customizing the interaction overview diagrams based on the application's selected features. The process for deriving this artifact is described in section 5.5.2. This process is demonstrated using SNOE's Execute Commands interaction overview. However, additional interaction diagrams are included in Appendix C. The Execute Commands interaction overview begins with a feature-based condition on the Command Execution pattern-specific feature group. SNOE selected to use the Low Volume Command Execution feature, thus only the path involving this feature is selected. Therefore the

other two paths are removed for SNOE. The final interaction overview diagram SNOE's Execute Commands is depicted in Figure 7-14. Additionally, the ports and connectors between the components in the FSW Spacecraft Clock executable design pattern and the FSW Hierarchical Control executable design pattern are removed.

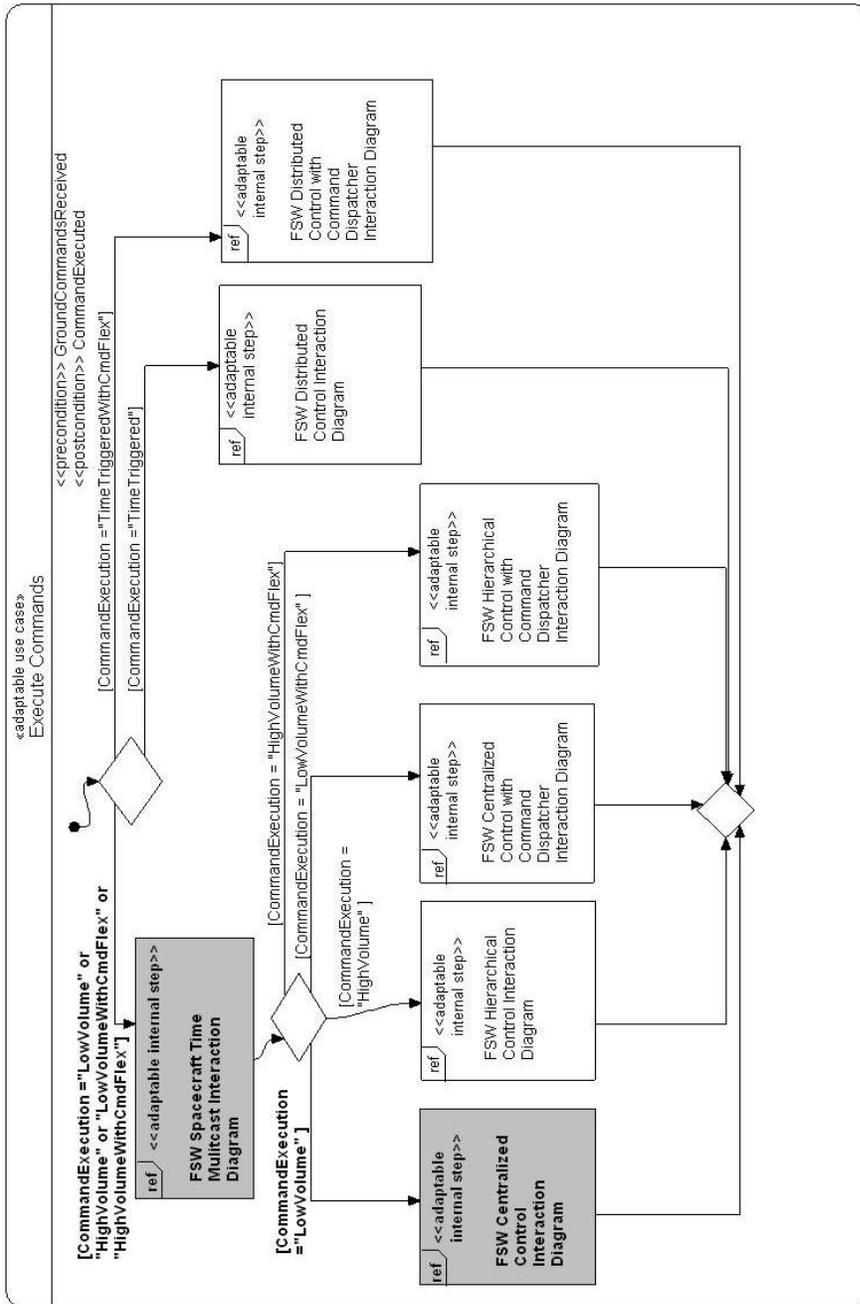


Figure 7-14 SNOE's command execution interaction overview diagram

7.5.6 SNOE Design Pattern within Layered Architecture

The last step in the analysis modeling phase is to update the Layered Architecture to include just the application specific components and variants. This view is illustrated using SNOE Centralized Control executable design pattern is depicted in Figure 7-15. The Device Interface Layer now only contains the SNOE specific variant components.

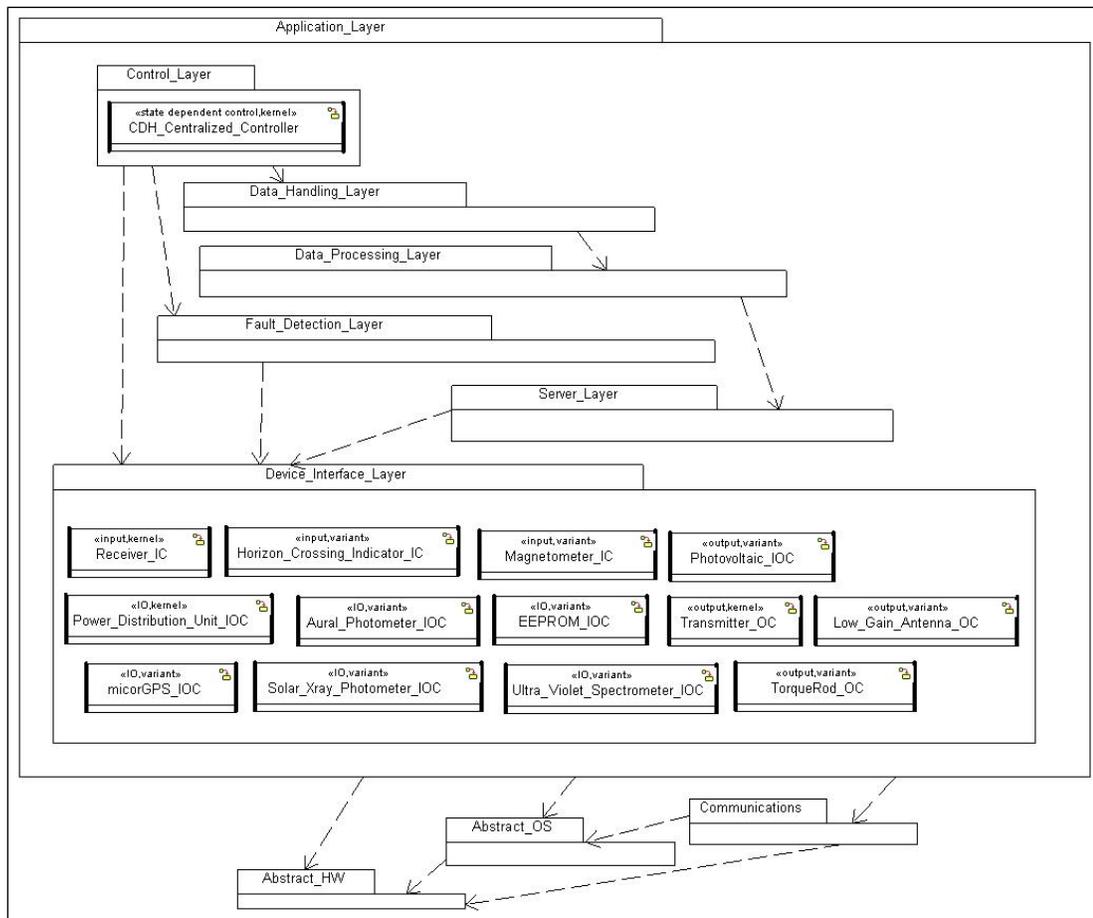


Figure 7-15 SNOE Architecture Layered View with Centralized Control Design Pattern Components

8 VALIDATION

This chapter describes the validation of the approach presented in this research. The approach is validated by ensuring the functional correctness of the individual DRE, SPL, and application executable design patterns. Additionally the FSW applications built following the approach are also validated. The validation of each piece is described in the following subsections.

8.1 DRE Executable Design Pattern Validation

The individual DRE executable design patterns are validated using executable object modeling with statecharts (Harel 1997). Test cases are created to test all states, transitions, and actions for the state machines of all the components in the DRE executable design pattern. The test cases capture the expected states, transitions, and actions that a component is expected to perform in response to an event or input data. Then the state machines are made executable using Harel's executable object modeling with statecharts and the test cases are then executed. If the components performed as expected for all steps then the test case is passed. If the test case fails, then the problem

is resolved and the test cases are re-run. Table 8-1 lists all the DRE design patterns that were validated as part of this research.

Table 8-1 Summary of DRE executable design patterns validated

DRE Design Pattern Name	
Broadcast	Monitor Actuator
Brokered Communication	Multicast
Centralized Control	Pipes and Filters
Client Server	Publish Subscribe
Command Dispatcher	Watch Dog
Compound Transaction	Sanity Check
Distributed Control	Single Protected Channel
Heterogeneous Redundancy Channel	Strategy
Hierarchical Control	Triple Modular Redundancy
Homogeneous Redundancy Channel	Two Phase Commit
Master Slave	

This validation process is illustrated using the Client Server executable design pattern, described in Appendix A. In this design pattern, the Client collects data from the Server. This design pattern only requires one test case to validate the communication sequence between client and server. This test case will cover all states, transitions and activities in the client and server components. The test case for this design pattern is depicted below in Table 8-2.

Table 8-2 Test case for the Client Server design pattern

Step	Expected Response	Pass/Fail
1. Client needs data from Server	<ul style="list-style-type: none"> • Client receives a requestNeededEvent • Client transitions into Preparing Request state • Client takes action to create request message • When client finishes message, it generates request event and transitions to Idle state • During transition, Client sends request message to Server 	
2. Server receives & processes request message	<ul style="list-style-type: none"> • Server transitions into Processing_Client_Request state • Server performs action to process the request and then generates processingComplete event • Server transitions into Preparing_Response state • Server performs actions to generate a response message • Server generates response event and transitions back to Idle • Along transition Server sends response message to Client 	
3. Client receives and processes the response message	<ul style="list-style-type: none"> • Client transitions into the Processing_Response state • Client performs actions in response to message • After actions are performed, client generates processingCompleteEvent • Client transitions into Idle state 	

After the test case is created, the design pattern is executed. The flow of states, transitions and actions is traced through the executing state machines. As seen in Figure 8-1 and Figure 8-2, the state machines perform as expected for each step in the test case. Since all test cases were passed, the design pattern is validated.

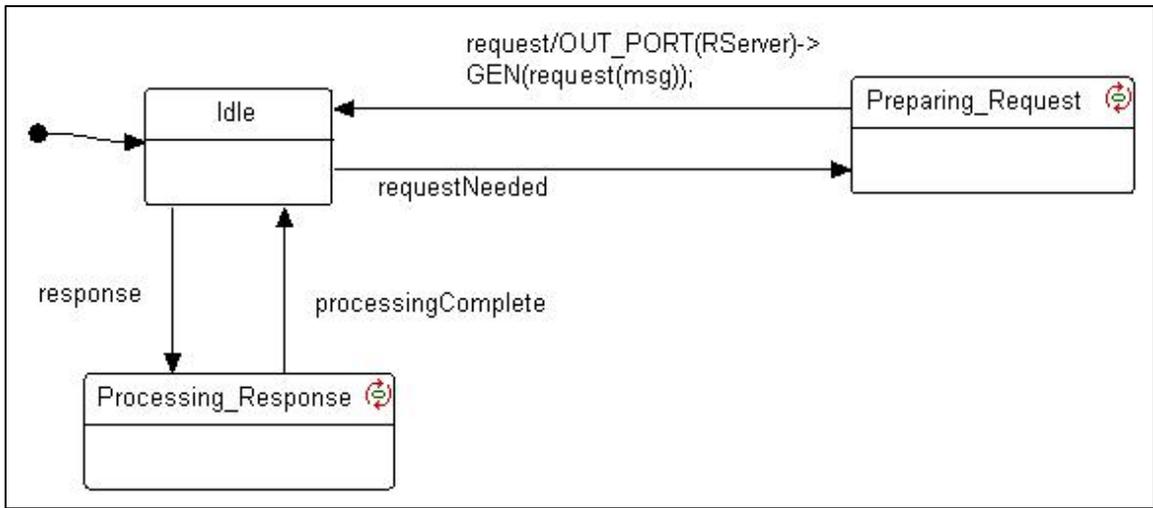


Figure 8-1 State machine for Client

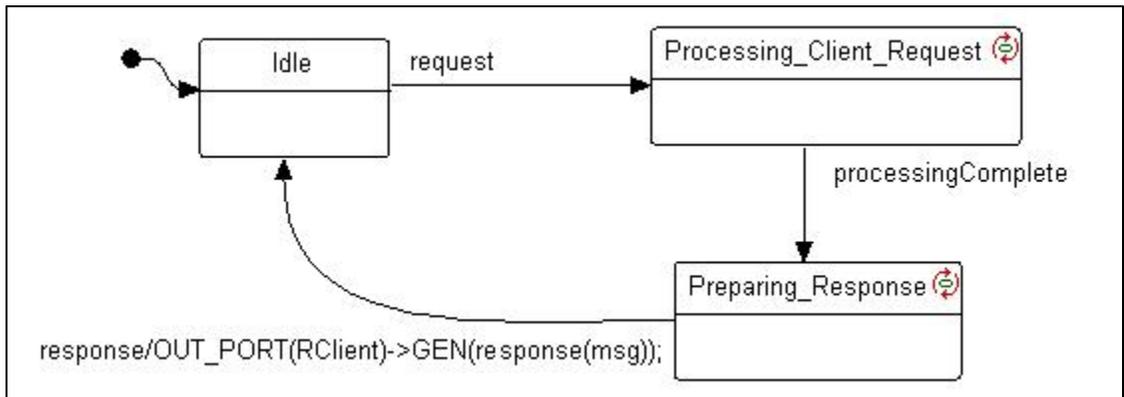


Figure 8-2 State machine for Server

8.2 FSW SPL Executable Design Pattern Validation

Second, the FSW SPL executable design patterns are also individually validated. The validation process is similar to the validation process at the DRE level. However the design patterns must function as required by the SPL features. Table 8-3 summarizes all the FSW SPL design patterns that were validated as part of this research along with the DRE executable design pattern(s) they are derived from.

Table 8-3 Summary of FSW SPL executable design patterns validated

DRE Executable Design Pattern (s)	FSW SPL Executable Design Pattern
Hierarchical Control	FSW Hierarchical Control
Centralized Control	FSW Centralized Control
Distributed Control	FSW Distributed Control
Hierarchical Control Command Dispatcher	FSW Hierarchical Control with Command Dispatcher
Centralized Control Command Dispatcher	FSW Centralized Control with Command Dispatcher
Distributed Control Command Dispatcher	FSW Distributed Control with Command Dispatcher
Compound Commit	FSW Telemetry Storage and Retrieval Compound Commit
Client Server	FSW Telemetry Storage and Retrieval Client Server
Master Slave Pipes and Filters	FSW Telemetry Formation Master Slave with Pipes and Filters
Master Slave Pipes and Filters Strategy	FSW Telemetry Formation Master Slave with Pipes and Filters & Strategy
Pipes and Filters	FSW Telemetry Formation Pipes and Filters
Pipes and Filters Strategy	FSW Telemetry Formation Pipes and Filters with Strategy
Protected Single Channel	FSW Telemetry Formation Reliability Protected Single Channel

Sanity Check	FSW Telemetry Formation Reliability Sanity Check
Client Server	FSW Payload Data Collection Multiple Client Multiple Server
Multicast	FSW Payload Data Collection Multicast
Publish Subscribe	FSW Payload Data Collection Publish Subscribe
Client Server	FSW Housekeeping Data Collection Multiple Client Multiple Server
Broadcast	FSW Housekeeping Data Collection Broadcast
Multicast	FSW Housekeeping Data Collection Multicast
Publish Subscribe	FSW Housekeeping Data Collection Publish Subscribe
Multicast	FSW Housekeeping Data Checks Multicast
Multicast	FSW Spacecraft Clock Multicast
Watchdog	FSW Memory Storage Device Fault Detection Watchdog

This validation process is illustrated using the FSW Housekeeping Client Server executable design pattern, described in Appendix B. This design pattern is based on the Ground Driven Housekeeping Data Collection feature which requires the Housekeeping_DClient collects the housekeeping data from the Housekeeping_DServer in response to a command from the ground station. This design pattern only requires one test case for each type of data that is requested to validate the communication sequence between the Housekeeping_DClient and the Housekeeping_DServer. These test cases will cover all types of data exchanged, states, transitions and activities in the Housekeeping_DClient and the Housekeeping_DServer. In addition to the behavior, the test cases also make sure the correct data is returned. One test case for this design pattern is depicted below in Table 8-4.

Table 8-4 Test Case for the Houskeeping_DClient Housekeeping_DServer design pattern

Step	Expected Response	Pass/Fail
4. Housekeeping_DClient needs data from Housekeeping_DServer	<ul style="list-style-type: none"> • Housekeeping_DClient receives a requestNeededEvent from the CDH_Centralized Controller to collect general housekeeping data • Housekeeping_DClient transitions into Preparing_Housekeeping_Data_Request state • Housekeeping_DClient takes action to create request message • When Housekeeping_DClient finishes message, it generates request event and transitions to Idle state • During transition, Housekeeping_DClient sends request message to Housekeeping_DServer 	
5. Housekeeping_DServer receives & processes request message	<ul style="list-style-type: none"> • Housekeeping_DServer transitions into Processing_Housekeeping_Data_Client_Request state • Housekeeping_DServer performs action to process the request and then generates processingComplete event • Housekeeping_DServer transitions into Preparing_Response state • Housekeeping_DServer performs actions to generate a response message • Housekeeping_DServer generates response event and transitions back to Idle • Along transition Housekeeping_DServer sends response message to Housekeeping_DClient 	
6. Housekeeping_DClient receives and processes the response message	<ul style="list-style-type: none"> • Housekeeping_DClient transitions into the Processing_Response state • Housekeeping_DClient performs actions in response to the general housekeeping data message • After actions are performed, Housekeeping_DClient generates processingCompleteEvent • Housekeeping_DClient transitions into Idle state 	

After the test case is created, the design pattern is executed. The flow of states, transitions, and actions are traced through the executing state machines. As seen in

Figure 8-3 and Figure 8-4 the state machines perform as expected for each step in the test case. Since all test cases were passed, the SPL design pattern is individually validated.

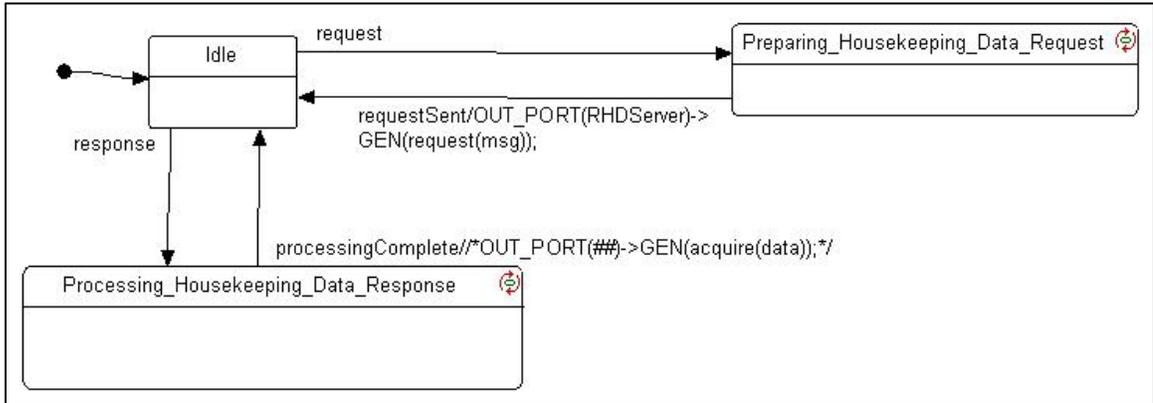


Figure 8-3 State machine for FSW SPL Housekeeping_DClient

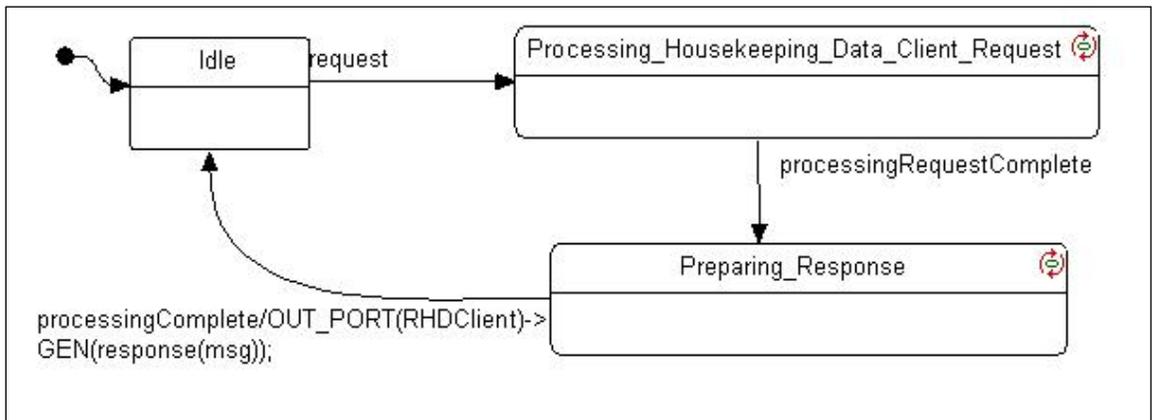


Figure 8-4 State machine for FSW SPL Housekeeping_DServer

8.3 Application Executable Design Pattern Validation

Next, the application executable design patterns are also individually validated. The validation process is the similar to the validation process at the DRE and FSE SPL levels. However the design patterns must function as required by the application features. Table 8-5 and Table 8-6 summarize all the FSW application design patterns that were validated as part of this research along with the FSW SPL executable design pattern(s) they are derived from.

Table 8-5 Summary of SNOE executable design patterns validated

FSW SPL Executable Design Pattern	SNOE Executable Design Pattern
FSW Centralized Control	SNOE Centralized Control
FSW Telemetry Storage and Retrieval Client Server	SNOE Telemetry Storage and Retrieval Client Server
FSW Telemetry Formation Pipes and Filters	SNOE Telemetry Formation Pipes and Filters
FSW Payload Data Collection Multiple Client Multiple Server	SNOE Payload Data Collection Multiple Client Multiple Server
FSW Housekeeping Data Collection Multiple Client Multiple Server	SNOE Housekeeping Data Collection Multiple Client Multiple Server
FSW Spacecraft Clock Multicast	SNOE Spacecraft Clock Multicast
FSW Memory Storage Device Fault Detection Watchdog	SNOE EEPROM Fault Detection Watchdog

Table 8-6 Summary of STEREO executable design patterns validated

FSW SPL Executable Design Pattern	STEREO Executable Design Pattern
FSW Hierarchical Control	STEREO Hierarchical Control
FSW Telemetry Storage and Retrieval Compound Commit	STEREO Telemetry Storage and Retrieval Compound Commit

FSW Telemetry Formation Pipes and Filters with Strategy	STEREO Telemetry Formation Pipes and Filters with Strategy
FSW Telemetry Formation Reliability Sanity Check	STEREO Telemetry Formation Reliability Sanity Check
FSW Payload Data Collection Multicast	STEREO Payload Data Collection Multicast
FSW Housekeeping Data Collection Multiple Client Multiple Server	STEREO Housekeeping Data Collection Multiple Client Multiple Server
FSW Housekeeping Data Collection Multicast	STEREO Housekeeping Data Collection Multicast
FSW Housekeeping Data Checks Multicast	STEREO Housekeeping Data Checks Multicast
FSW Spacecraft Clock Multicast	STEREO Spacecraft Clock Multicast
FSW Memory Storage Device Fault Detection Watchdog	STEREO EEPROM Fault Detection Watchdog

8.4 FSW SPL Members' Software Architectures Validation

To validate the SPL members' software architectures a reusable model-based testing approach for SPL is applied. This approach helps to reduce the overall validation effort by created reusable model based test assets that can be customized and used for SPL applications. The approach applied is based on the Customizable Activity Diagrams, Decision Tables, and Test Specifications (CADET) method (Olimpiew 2008; Olimpiew & Gomaa 2009). CADeT is extended so the test assets are related to the SPL with design pattern level variability rather than component level variability.

CADET was selected because it is use case and feature based. It needs to identify the use case scenarios for each use case, which were identified as part of the FSW SPL. It also develops reusable FSW SPL test specifications based on use case scenarios and the features that impact the use case scenarios. Thus CADeT ensures that all FSW SPL

features are covered in the reusable FSW SPL test specifications. This is particularly important for this research because by ensuring that all features are covered, it consequently ensures that all design patterns are covered. This is because each pattern specific feature is mapped to a design pattern or pre-integrated design pattern combination. Thus if all pattern specific features are cover, all design patterns are also covered.

CADeT involves performing activities in both the SPL and application engineering processes. The validation process for the FSW SPL is described below in more detail.

8.4.1 FSW SPL Validation Activities

8.4.1.1 FSW SPL Decision Tables and Test Specifications

The first step in the functional validation is to create a decision table for each FSW SPL activity diagram, which are described in section 0. The purpose of the decision table is to organize the associations between the test specifications and features to cover all use case scenarios in the SPL. The table contains a row for activity node, feature, and execution condition in the activity diagram. A column is created for each simple path that can be created through the activity diagram. The features, execution conditions, and activity nodes that it exercises are marked in the appropriate rows. The simple paths then become reusable test specifications for the SPL (Olimpiew 2008; Olimpiew & Gomaa 2009).

The FSW SPL command and data handling is associated with three use cases. Therefore, decision tables for each of these use cases must be created. This step is demonstrated using the Execute Commands' activity diagram. The decision table for this use case is populated and illustrated in Table 8-7. This use case has six unique paths, which are captured in the decision table's test specification columns.

Table 8-7 Execute Commands Decision Table

<<adaptable use case>> Execute Commands		1	2	3	4	5	6
Test Specifications		<<adaptable>> LV Cmd Exe.	<<adaptable>> HV Cmd Exe.	<<adaptable>> TT Cmd Exe.	<<adaptable>> LV Flex. Cmd Exe.	<<adaptable>> HV Flex. Cmd Exe.	<<adaptable>> TT Flex. Cmd Exe.
Feature Conditions							
CommandExecution = LowVolume		T					
CommandExecution = HighVolume			T				
CommandExecution = TimeTriggered				T			
CommandExecution = LowVolumeFlexCmd					T		
CommandExecution = HighVolumeFlexCmd						T	
CommandExecution = TimeTriggeredFlexCmd							T
SpacecraftClock		T	T	F	T	T	F
Preconditions :Ground Commands Received							
Actions							
1	<<adaptable internal step>> Send spacecraft time update (Command Volume)	X	X		X	X	
2	<<adaptable internal step>> Execute a low volume of commands that is appropriate for spacecraft mode (Modes and Spacecraft IO Devices)	X					
3	<<adaptable internal step>> Execute a high volume of commands that is appropriate for spacecraft mode (Modes and Spacecraft IO Devices)		X				

4	<<adaptable internal step>> Execute a low volume of commands that supports new commands and is appropriate for spacecraft mode (Modes and Spacecraft IO Devices)				X		
5	<<adaptable internal step>> Execute a high volume of commands that supports new commands and is appropriate for spacecraft mode (Modes and Spacecraft IO Devices)					X	
6	<<adaptable internal step>> Execute commands with strict temporal predictability & in manner that is appropriate for spacecraft mode (Modes and Spacecraft IO Devices)			X			
7	<<adaptable internal step>> Execute commands with strict temporal predictability & in manner that supports new commands and is appropriate for spacecraft mode (Modes and Spacecraft IO Devices)						X
Post Conditions: Commands Executed							

8.4.1.2 FSW SPL Feature and Design Pattern Test Coverage

The next step in the validation process is to create a test specification / feature / design pattern table, which captures the relationship between features and design patterns to test specifications. This table is used to quickly summarize and analyze the test specifications to ensure that all features and design patterns in the FSW SPL are covered in the FSW SPL test specifications. For each test specification in the use case activity diagram, the features that affect the test specification, the adaptable test steps in a test specification, and the nature of the feature interactions, are depicted in a feature combination function (Olimpiew 2008; Olimpiew & Gomaa 2009). The test specification / feature / design pattern table for Execute Commands use case is depicted in Table 8-8.

This table summarized the features and design patterns covered in the test specifications for this use case. In the FSW SPL test specifications created for the command and data handling subsystem, all features and design patterns were successfully covered.

However, the test specifications are not complete at the SPL level. This is because all the variability the SPL architecture is not fully specified because architectural variability is captured in the design patterns. Therefore the steps in the test specifications that are not fully specified must be customized to the actual application behavior during application engineering. This will typically result in the need additional test specifications at the application level.

Table 8-8 Execute Commands test specification / feature / design pattern table

Test Spec.	Feature to Test Spec.	Adaptable Steps	Feature to adaptable test step	Design Pattern
LV Cmd Exe.	Command Execution = LowVolume Spacecraft Clock = T	<ul style="list-style-type: none"> Send spacecraft time update (Command Volume and Spacecraft IO Devices) Execute a low volume of commands that is appropriate for spacecraft mode (Modes and Spacecraft IO Devices) 	Modes + Antennas + Antenna Gimbals + Memory Storage + Devices+ Power Appendages + Power Devices + Attitude Control Devices + Attitude Determination Devices + Payload Devices + Thrusters + Heaters + Louvers + Temp. Sensors	FSW Centralized Control FSW Spacecraft Clock

HV Cmd Exe.	Command Execution = HighVolume Spacecraft Clock = T	<ul style="list-style-type: none"> Send spacecraft time update (Command Volume and Spacecraft IO Devices) Execute a high volume of commands that is appropriate for spacecraft mode (Modes and Spacecraft IO Devices) 	Modes + Antennas + Antenna Gimbals + Memory Storage + Devices+ Power Appendages + Power Devices + Attitude Control Devices + Attitude Determination Devices + Payload Devices + Thrusters + Heaters + Louvers + Temp. Sensors	FSW Hierarchical Control FSW Spacecraft Clock
TT Cmd Exe.	Command Execution = TimeTriggered	<ul style="list-style-type: none"> Execute commands with strict temporal predictability and is appropriate for spacecraft mode (Modes and Spacecraft IO Devices) 	Modes + Antennas + Antenna Gimbals + Memory Storage + Devices+ Power Appendages + Power Devices + Attitude Control Devices + Attitude Determination Devices + Payload Devices + Thrusters + Heaters + Louvers + Temp. Sensors	FSW Time Triggered Control
LV Flex. Cmd Exe.	Command Execution = LowVolumeFl exCmd Spacecraft Clock = T	<ul style="list-style-type: none"> Send spacecraft time update (Command Volume and Spacecraft IO Devices) Execute a low volume of commands that supports new commands and is appropriate for spacecraft mode (Modes and Spacecraft IO Devices) 	Modes + Antennas + Antenna Gimbals + Memory Storage + Devices+ Power Appendages + Power Devices + Attitude Control Devices + Attitude Determination Devices + Payload Devices + Thrusters + Heaters + Louvers + Temp. Sensors	FSW Centralized Control with Command Dispatcher FSW Spacecraft Clock
HV Flex. Cmd Exe.	Command Execution = HighVolumeFl exCmd Spacecraft Clock = T	<ul style="list-style-type: none"> Send spacecraft time update (Command Volume and Spacecraft IO Devices) Execute a high volume of commands that supports new commands and is appropriate for spacecraft mode (Modes and Spacecraft IO Devices) 	Modes + Antennas + Antenna Gimbals + Memory Storage + Devices+ Power Appendages + Power Devices + Attitude Control Devices + Attitude Determination Devices + Payload Devices + Thrusters + Heaters + Louvers + Temp. Sensors	FSW Hierarchical Control with Command Dispatcher FSW Spacecraft Clock

TT Flex. Cmd Exe.	Command Execution = TimeTriggered FlexCmd Spacecraft Clock = T	<ul style="list-style-type: none"> Execute commands with strict temporal predictability & in manner that supports new commands and is appropriate for spacecraft mode (Modes and Spacecraft IO Devices) 	Modes + Antennas + Antenna Gimbals + Memory Storage + Devices+ Power Appendages + Power Devices + Attitude Control Devices + Attitude Determination Devices + Payload Devices + Thrusters + Heaters + Louvers + Temp. Sensors	FSW Time Triggered Control with Command Dispatcher
----------------------------	---	--	--	--

8.4.2 SNOE Functional Validation

8.4.2.1 SNOE Test Specifications

The first step in the functional validation process for SNOE is to customize the test specifications to the application. This is accomplished by first setting feature selection values in the feature list to the feature selections of the application. Then the feature list, decision tables and test specifications are updated to reflect the features selected for the application (Olimpiew 2008; Olimpiew & Gomaa 2009).

For example, in FSW SPL the Execute Command decision table in Table 8-7 contains six test specifications. SNOE only uses the Low Volume Command Execution feature, thus adaptable test specification LV Exe. Cmd. is the only applicable test specification.

The next step in the validation process is to refine the adaptable steps in the selected test specifications. This step must be performed because the FSW SPL test specifications were captured at a higher degree of abstraction. Therefore it is not until the application

engineering phase that these adaptable steps should be refined to non-adaptable. This involves refining each adaptable test specification into non-adaption steps based on the application's feature selection. In many cases refining the test specification will also result in additional test specifications. This is because additional paths and steps are often needed when an adaptable step is refined. For example, the Command Execution use case test specification should be refined to cover all application commands. This results in a new test case for each application command.

This process is illustrated using SNOE's adaptable test specification LV Exe. Cmd. seen in Table 8-7, which contains two adaptable steps. The first adaptable step involves updating the spacecraft time and notifying the proper components. Based on SNOE's feature selection, only the CDH_Centralized_Controller and SNOE's IO devices need a time update. SNOE's IO devices include the Low Gain Antenna (LGA), EEPROM, torque rod, magnetometer, horizon crossing indicators, microGPS, auroral photometer, solar x-ray photometer, and ultraviolet spectrometer. Therefore the step is updated to reflect this specific change, as seen in

Table 8-9.

Table 8-9 Subset of SNOE’s Execute Commands Decision Table

<< use case>> Execute Commands		1	2
Test Specifications		EEPROM Restart Cmd	LGA Reinitiali ze Cmd
Feature Conditions			
CommandExecution = LowVolume		T	T
Execution Conditions			
CommandType		EEPROM restart	LGA Reinitiali ze
Preconditions :Ground Commands Received			
Actions			
1	<<internal step>> Send spacecraft time update to CDH Centralized Controller and SNOE’s IO devices	X	X
2	<<internal step>> Validate and determine the command sequence for a low volume of commands that is appropriate for spacecraft mode	X	X
3	<<internal step>> Restart EEPROM	X	
4	<<internal step>> Re-initialize Low Gain Antenna (LGA)		X
Post Conditions: Commands Executed			

The second step involves executing a low volume of commands that is appropriate for spacecraft mode, which is impacted by several variation points. The variation points can be set based on the SNOE’s modes and IO devices. For example, SNOE does not utilize the optional Heater pattern specific feature, thus non-adaptable steps for the Heater variation point do not need to be created for SNOE. Next, SNOE utilizes the EEPROM

for its Memory Storage Device variation point and a LGA for its Antenna variation point. Therefore non-adaptable steps related to the Memory Storage Device and Antenna variation points are modeled using the EEPROM and LGA, respectively. The same process is followed for the other variation points. A subset of the updated decision table for SNOE is shown in

Table 8-9.

The refining of the adaptable steps also resulted in the additional execution condition called `CommandType`, which controls the path taken based on the type of command that must be executed. This results in several additional test specifications based on the execution condition. A subset is depicted in

Table 8-9.

8.4.2.2 SNOE Test Data

The next step in the SNOE's validation process is to select the input data for the test specifications. The test data selection approach described in CADeT is focused on transaction-based software (Olimpiew 2008; Olimpiew & Gomaa 2009). In the case of real-time and embedded software, the actions performed by the software are not always transaction-based and written to a database. Thus a different strategy is needed.

To address the challenge of non-transactional test specifications, the test specifications will be populated with design pattern based test data based on the state machines of the design pattern's components. A component's actions are state dependent and different input data and events will result in different actions being performed by the components. The test specifications will be populated with the state, transitions, and actions the design pattern components are expected to perform in response to a particular input data or event. For example, if a step involves a client component making a request, then the value listed in this step is 'the client component transitions from Idle to Preparing Request state'.

To determine which design pattern is applicable in the step, the interaction overview diagrams are used as a guide, where the design pattern that corresponds to the step is used. Then the state machines within that design pattern can then be referenced to determine the expected behavior. Then the design pattern component's state machines are analyzed to determine states, transitions, and actions that components should perform for the test specification. These are then recorded in the steps for the test specification.

This process is illustrated using SNOE's LGA Reinitialize test specifications depicted in

Table 8-9. The first step in this test specification is to send spacecraft time updates to CDH_Centralized_Controller and SNOE's IO devices. According to SNOE's Execute Commands interaction overview diagram in section 7.5.5, this step is supported by the FSW Spacecraft Clock Multicast executable design pattern. Within this design pattern, the Spacecraft_Clock_Multicast component performs this step by transitioning into the Notifying_Consumer state and performing a series of actions to send updates to SNOE's IO devices and CDH_Centralized_Controller. Therefore this behavior is listed as the test data for this step, as seen in Table 8-10.

Table 8-10 SNOE's LGA Reinitialize Test Specification

<< use case>> Execute Commands	
Test Specification: LGA Reinitialize Cmd	
Feature Conditions	
CommandExecution = LowVolume	T
Execution Conditions	
CommandType	LGA Reinitialize
Preconditions :Ground Commands Received	
Actions	
<<internal step>> Send spacecraft time update to CDH_Centralized_Controller and SNOE's IO devices	<ul style="list-style-type: none"> Spacecraft_Clock_Multicast component transitions into Notifying_Consumer state Spacecraft_Clock_Multicast sends updates to SNOE's IO devices and CDH_Centralized_Controller

	<<internal step>> Validate and determine the command sequence for a low volume of commands that is appropriate for spacecraft mode	<ul style="list-style-type: none"> • CDH_Centralized_Controller transitions into Validating_Command state & validates the LGA Reinitialize command
	<<internal step>> Reinitialize Low Gain Antenna (LGA)	<ul style="list-style-type: none"> • CDH_Centralized_Controller transitions into Executing_Command state • CDH_Centralized_Controller calls the off() method on the Low_Gain_Antenna_OC • Low_Gain_Antenna_OC transitions into Final State • CDH_Centralized_Controller calls the initialize() method on the Low_Gain_Antenna_OC • Low_Gain_Antenna_OC transitions into Initializing state • CDH_Centralized_Controller calls the on() method on the Low_Gain_Antenna_OC • Low_Gain_Antenna_OC transitions into Working State • CDH_Centralized_Controller transitions into Logging_Command state and logs command
Post Conditions: Command Executed		

The second step in Table 8-10 states “Validate and determine the command sequence for a low volume of commands that is appropriate for spacecraft mode”. Using SNOE’s Execute Commands interaction overview diagram in Figure 7-14 as a guide this step is a refinement of the Execute Low Volume of Commands step, which is supported by the FSW Centralized Control executable design pattern. Using this design pattern, it is determined that step two is performed by the CDH_Centralized_Controller component. Based on the SNOE’s CDH_Centralized_Controller’s state machine in described section 7.5.4.2, it is expected that upon receiving ground commands as an inputEventNotification, it will transition into the Validating_Command state and perform

the actions in this state. Therefore this behavior is listed as the test data for this step, as seen in Table 8-10.

Finally, the last step in this test specification is to reinitialize Low Gain Antenna (LGA).

This step is also a refinement of the Execute Low Volume of Commands step, therefore it is supported by the FSW Centralized Control executable design pattern. Using this design pattern, it is determined that this step is performed by the CDH_Centralized_Controller and Low_Gain_Antenna_OC component. To reinitialize the low gain antenna, the CDH_Centralized_Controller performs a series of actions involving the Low_Gain_Antenna_OC . The Low_Gain_Antenna_OC is expected to respond properly to these commands. Thus, this behavior is listed as the test data for this step, as seen in Table 8-10.

8.4.2.3 Test application

The final step in the functional validation is to execute the tests on the SPL member's model. The software architecture is made executable using executable state chart semantics (Harel 1997). When the tests are executing the test engineer will follow the flow of events and messages through the system and design patterns to ensure they execute as expected. A "Pass" result is given to a test if the observed state machine behavior matched the expected behavior for all the steps in the test specification. A "Fail" result is given to a test where the observed state machine behavior did not matched

the expected behavior for all the steps in the test specification. If any of the tests fail, then the software architecture is modified and regression tested to ensure the software performs as expected. After all tests are successfully passed the architecture is considered to be functionally validated.

On SNOE, all the test specifications related to the command and data handling subsystem were executed and passed. A summary of the test specifications executed along with the features and design patterns they tested are summarized in Table 8-11. In total 22 test specifications were executed and passed. These test specifications covered all of SNOE seven pattern specific features and ten pattern variability features. Since all the pattern specific features were covered, consequently all of SNOE’s seven design patterns were covered.

Table 8-11 Summary of SNOE's Test Specifications

Test Specification	Pass/ Fail	Features Covered	Design Patterns Covered
LGA Reinitialize	Pass	Low Gain Antenna Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock
EEPROM Restart	Pass	EEPROM Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
Photovoltaic Command	Pass	Photovoltaic Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast

PDU Command	Pass	Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
HCI Command	Pass	Horizon Crossing Indicator Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
Magnetometer Command	Pass	Magnetometer Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
Torque Rod Command	Pass	Torque Rod Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
Receiver Command	Pass	Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
Transmitter Command	Pass	Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
Solar X-Ray Photometer Command	Pass	Solar X-Ray Photometer Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
microGPS Command	Pass	microGPS Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
Auroral Photometer Command	Pass	Auroral Photometer Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
Ultraviolet Spectrometer Command	Pass	Ultraviolet Spectrometer Low Volume Command Execution	SNOE Centralized Control SNOE Spacecraft Clock Multicast
Collect & Store Critical Housekeeping Data	Pass	Low Volume Command Execution Low Volume Telemetry Formation Ground Driven Housekeeping Data Collection IO Devices	SNOE Centralized Control SNOE Spacecraft Clock Multicast SNOE Housekeeping Client Server SNOE Pipes and Filters SNOE Telemetry Client Server
Collect & Store General Housekeeping Data	Pass	Low Volume Command Execution Low Volume Telemetry Formation Ground Driven Housekeeping Data Collection IO Devices	SNOE Centralized Control SNOE Spacecraft Clock Multicast SNOE Housekeeping Client Server SNOE Pipes and Filters SNOE Telemetry Client Server

Collect & Store Subsystem Status Housekeeping Data	Pass	Low Volume Command Execution Low Volume Telemetry Formation Ground Driven Housekeeping Data Collection IO Devices	SNOE Centralized Control SNOE Spacecraft Clock Multicast SNOE Housekeeping Client Server SNOE Pipes and Filters SNOE Telemetry Client Server
Collect & Store Comm. Status Housekeeping Data	Pass	Low Volume Command Execution Low Volume Telemetry Formation Ground Driven Housekeeping Data Collection IO Devices	SNOE Centralized Control SNOE Spacecraft Clock Multicast SNOE Housekeeping Client Server SNOE Pipes and Filters SNOE Telemetry Client Server
Collect & Store Solar X-Ray Photometer Data	Pass	Low Volume Command Execution Low Volume Telemetry Formation Ground Driven Payload Data Collection Solar X-Ray Photometer	SNOE Centralized Control SNOE Spacecraft Clock Multicast SNOE Payload Client Server SNOE Pipes and Filters SNOE Telemetry Client Server
Collect & Store microGPS Data	Pass	Low Volume Command Execution Low Volume Telemetry Formation Ground Driven Payload Data Collection microGPS	SNOE Centralized Control SNOE Spacecraft Clock Multicast SNOE Payload Client Server SNOE Pipes and Filters SNOE Telemetry Client Server
Collect & Store Auroral Photometer Data	Pass	Low Volume Command Execution Low Volume Telemetry Formation Ground Driven Payload Data Collection Auroral Photometer	SNOE Centralized Control SNOE Spacecraft Clock Multicast SNOE Payload Client Server SNOE Pipes and Filters SNOE Telemetry Client Server
Collect & Store Ultraviolet Spectrometer Data	Pass	Low Volume Command Execution Low Volume Telemetry Formation Ground Driven Payload Data Collection Ultraviolet Spectrometer	SNOE Centralized Control SNOE Spacecraft Clock Multicast SNOE Payload Client Server SNOE Pipes and Filters SNOE Telemetry Client Server
EEPROM Watchdog Detect Failure	Pass	EEPROM Memory Storage Device Fault Detection	SNOE Memory Storage Device Watchdog

8.4.3 STEREO Functional Validation

The STEREO application was validated following the same process that was applied on SNOE. For example, the FSW SPL Execute Command decision table in Table 8-7 contains six test specifications. STEREO only uses the High Volume Command Execution feature, thus adaptable test specification HV Exe. Cmd. is the only applicable test specification selected for STEREO from the Execute Commands use case test specifications.

Next, the adaptable steps in the selected test specifications are refined for STEREO. This process is illustrated using STEREO’s adaptable test specification HV Exe. Cmd. seen in Table 8-7, which contains two adaptable steps. Each step is refined based on STEREO’s feature selection described in Appendix D. Table 8-12 shows a subset of STEREO’s Execute Commands Decision Table.

Table 8-12 Subset of STEREO’s Execute Commands Decision Table

<< use case >> Execute Commands	1	2
Test Specifications	EEPROM Restart Cmd	HGA Reinitiali ze Cmd
Feature Conditions		
CommandExecution = HighVolume	T	T
Execution Conditions		

CommandType		EEPROM restart	HGA Reinitiali ze
Preconditions :Ground Commands Received			
Actions			
1	<<internal step>> Send spacecraft time update to CDH Centralized Controller and SNOE's IO devices	X	X
2	<<internal step>> Validate and determine the command sequence for a low volume of commands that is appropriate for spacecraft mode	X	X
3	<<internal step>> Restart EEPROM	X	
4	<<internal step>> Re-initialize Low Gain Antenna (LGA)		X
Post Conditions: Commands Executed			

Next, STEREO's test specifications were populated with design pattern based test data. This process is illustrated using STEREO's reinitialize HGA antenna test specification. The test data for this test specification determined from STEREO's Execute Commands interaction overview diagram and executable design patterns described in Appendix D.

Table 8-13 STEREO's HGA Reinitialize Test Specification

<< use case>> Execute Commands	
Test Specification: HGA Reinitialize Cmd	
Feature Conditions	
CommandExecution = HighVolume	T
Execution Conditions	
CommandType	HGA Reinitialize
Preconditions :Ground Commands Received	

Actions		
	<<internal step>> Send spacecraft time update to CDH_Centralized_Controller and SNOE's IO devices	<ul style="list-style-type: none"> • Spacecraft_Clock_Multicast component transitions into Notifying_Consumer state • Spacecraft_Clock_Multicast sends updates to SNOE's IO devices and CDH_Coordinator
	<<internal step>> Validate and determine the command sequence for a low volume of commands that is appropriate for spacecraft mode	<ul style="list-style-type: none"> • CDH_Coordinator transitions into Validating_Command state & validates the HGA Reinitialize command • CDH_Coordinator determines the command must be executed and transitions into Dispatching Commands state and sends the command to the CDH_Centralized_Controller • CDH_Coordinator transitions back to the Idle state
	<<internal step>> Reinitialize Low Gain Antenna (HGA)	<ul style="list-style-type: none"> • CDH_Controller transitions into Executing_Command state • CDH_Controller calls the off() method on the High_Gain_Antenna_OC • High_Gain_Antenna_OC transitions into Final State • CDH_Controller calls the initialize() method on the High_Gain_Antenna_OC • High_Gain_Antenna_OC transitions into Initializing state • CDH_Controller calls the on() method on the High_Gain_Antenna_OC • High_Gain_Antenna_OC transitions into Idle State • CDH_Controller calls the position() method on the High_Gain_Antenna_OC • High_Gain_Antenna_OC transitions into Positioning State • High_Gain_Antenna_OC finishes positioning and transitions into Idle State • CDH_Controller transitions into Logging_Command state and logs command
Post Conditions: Command Executed		

A summary of the test specifications that were executed along with the features and design patterns they tested are summarized in Table 8-14. In total 32 test specifications were executed and passed. These test specifications covered all of STEREO's 10 pattern specific features and 19 pattern variability features. Since all the pattern specific features were covered, consequently all design patterns were covered.

Table 8-14 Summary of STEREO's Test Specifications

Test Specification	Pass/ Fail	Features Covered	Design Patterns Covered
HGA Reinitialize	Pass	High Gain Antenna High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
MGA Reinitialize	Pass	Medium Gain Antenna High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
LGA Reinitialize	Pass	Low Gain Antenna High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
EEPROM Restart	Pass	EEPROM High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
Photovoltaic Command	Pass	Photovoltaic High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
Photovoltaic Position	Pass	Photovoltaic Power Appendage High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
PDU Command	Pass	High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
Star Tracker Command	Pass	Star Tracker High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
Sun Sensor Command	Pass	Sun Sensor High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
IRU Command	Pass	IRU High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
RWA Command	Pass	RWA High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock Multicast
Receiver Command	Pass	High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock
Transmitter Command	Pass	High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock
Propulsion Command	Pass	Propulsion Thruster High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock

Thermistor Command	Pass	Thermistor High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock
IMPACT Command	Pass	IMPACT High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock
PLASTIC Command	Pass	PLASTIC High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock
SWAVES Command	Pass	SWAVES Photometer High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock
SECCHI Command	Pass	SECCHI Command High Volume Command Execution	STEREO Hierarchical Control STEREO Spacecraft Clock
Collect & Store Critical Housekeeping Data	Pass	High Volume Command Execution Flexible High Volume Telemetry Formation Quick Check Ground Driven Housekeeping Data Collection High Volume Telemetry Storage & Retrieval IO Devices	STEREO Hierarchical Control STEREO Spacecraft Clock STEREO Housekeeping Client Server STEREO Pipes and Filters with Strategy STEREO Sanity Check STEREO Telemetry Compound Commit
Collect & Store General Housekeeping Data	Pass	High Volume Command Execution Low Volume Telemetry Formation Quick Check Ground Driven Housekeeping Data Collection High Volume Telemetry Storage & Retrieval IO Devices	STEREO Hierarchical Control STEREO Spacecraft Clock STEREO Housekeeping Client Server STEREO Pipes and Filters with Strategy STEREO Sanity Check STEREO Telemetry Compound Commit
Collect & Store General IMPACT Housekeeping Data	Pass	Event Driven Housekeeping Data Collection High Volume Telemetry Storage & Retrieval IMPACT	STEREO IMPACT Multicast STEREO Telemetry Client Server
Collect & Store General PLASTIC Housekeeping Data	Pass	Event Driven Housekeeping Data Collection High Volume Telemetry Storage & Retrieval PLASTIC	STEREO PLASTIC Multicast STEREO Telemetry Client Server

Collect & Store General WAVES Housekeeping Data	Pass	Event Driven Housekeeping Data Collection High Volume Telemetry Storage & Retrieval WAVES	STEREO WAVES Multicast STEREO Telemetry Client Server
Collect & Store General SECCHI Housekeeping Data	Pass	Event Driven Housekeeping Data Collection High Volume Telemetry Storage & Retrieval SECCHI	STEREO SECCHI Multicast STEREO Telemetry Client Server
Collect & Store General IMPACT Data	Pass	Event Driven Payload Data Collection High Volume Telemetry Storage & Retrieval IMPACT	STEREO IMPACT Multicast STEREO Telemetry Client Server
Collect & Store General PLASTIC Data	Pass	Event Driven Payload Data Collection High Volume Telemetry Storage & Retrieval PLASTIC	STEREO PLASTIC Multicast STEREO Telemetry Client Server
Collect & Store General WAVES Data	Pass	Event Driven Payload Data Collection High Volume Telemetry Storage & Retrieval WAVES	STEREO WAVES Multicast STEREO Telemetry Client Server
EEPROM Watchdog Detect Failure	Pass	EEPROM Memory Storage Device Fault Detection	STEREO Memory Storage Device Watchdog
Housekeeping Data Fault	Pass	Housekeeping Data Checks	STEREO Housekeeping Data Check Multicast
EEPROM Watchdog Stroke	Pass	EEPROM Memory Storage Device Fault Detection	STEREO Memory Storage Device Watchdog
Housekeeping Data without fault	Pass	Housekeeping Data Checks	STEREO Housekeeping Data Check Multicast

8.5 Validation Summary

In summary, validation was performed on the individual DRE, SPL, and application executable design patterns. Additionally the two FSW applications, SNOE and STEREO, which were built from the FSW SPL following the approach, were also

validated for functional correctness. SNOE and STEREO were validated in such a way that all feature and design patterns were tested. These validation activities were performed to achieve validation of the proposed approach. The approach is validated because the validation effort determined that the approach successfully produced an FSW SPL and two FSW SPL applications.

9 CONCLUSIONS AND FUTURE WORK

This research has presented an approach for building domain specific software architectures from software architectural design patterns. Along with the approach, a set of DRE architectural and executable software architectural design patterns is provided. An unmanned space flight software product line (FSW SPL) and two FSW applications were developed to illustrate and validate the approach. The SPL approach used in this research has several benefits. First, it provided a systematically and repeatable method for constructing software architectures from software architectural design patterns. The resulting executable software architectures can be leveraged for design time validation. The use of three levels of executable design patterns facilitates the systematic customization of design patterns and helps it reduce the amount of work required by the application engineer. This chapter summarizes the contributions of this research and highlights some potential areas for future work.

9.1 Contributions

The main contribution of this research is an approach to build domain specific software architectures from software architectural design patterns. The other contributions are described next:

a) Executable design patterns

Previous research (Gomaa 2005; Buschmann et al. 2007, Buschmann et al. 1996; Douglass 2003; Gamma et al. 1995; Kircher & Jain 2004; D. Schmidt et al. 2000; Selic 1996a, Selic 1996b, Selic 2004) has only provided design pattern specifications to help in the application of design patterns. This research takes an additional step and provides a set of DRE executable design patterns using state machines. The purpose of providing executable design patterns is so that they can be systematically incorporated into executable software architectures. This enables not only the design patterns to be individually validated for functional correctness during design time, but it also facilitates the validation of the executable software architectures produced.

b) Software product lines with design pattern level variability

Existing SPL approaches such as (Gomaa 2005; Clements & Northrop 2002; Pohl et al. 2005) capture variability at the subsystem, component, and connector level. While this approach works well in many domains, it has the potential not to scale when SPLs require a significant amount of architectural variability. This is because to capture all the variability using existing approaches large amount of components and connectors must be individually modeled, which can be time consuming to create and cumbersome to

maintain. Thus this research addressed the problem of scalability by reducing the amount of variability that must be modeled by mapping features to design patterns. Design patterns contain customizable components, connectors, and interactions rather than specific components, connectors, and interactions. Therefore several different combinations of specific components, connectors, and interactions are abstracted into one design pattern and do not need to be individually modeled. Thus less modeling is required at the SPL level since features have dependencies on design patterns rather than multiple individual components and connectors. Design patterns also give the application developer the capability of customizing the patterns to the needs of the application. The tradeoff is that the application engineering process does require additional modeling since the application specific components, connectors, and interactions must be derived from the design patterns. However, guidance is provided to help assist the application engineer and ensure the SPL architecture is maintained.

c) Three levels of executable design pattern customization

Previous research (Gomaa 2005; Buschmann et al. 2007; Buschmann et al. 1996; Douglass 2003; Gamma et al. 1995; Kircher & Jain 2004; D. Schmidt et al. 2000; Selic 1996a; Selic 1996b; Selic 2004) has only provided platform independent design pattern specifications to help in the application of design patterns. This research goes beyond these works by modeling three levels of architectural design patterns to progressively address variability within the patterns themselves and variability in the patterns selected

for a member application. The purpose of providing three levels of executable design patterns is to facilitate reuse and customization during SPL and SPL application development. Additionally, this also saves the software engineer time when building SPL and SPL applications because they provide a good starting point, as opposed to starting completely from scratch. First, the DRE architectural and executable design patterns are modeled once and capture variability within the individual design patterns. They serve as the foundation for the other two levels of executable design patterns. Next are the SPL level architectural and executable design patterns. This level of design pattern captures the customizations of DRE design patterns to reflect the needs and variability of the SPL features. Finally, there are the application's architectural and executable design patterns, which are derived from the SPL executable design patterns based on the application's features.

d) Design pattern integration modeling

Existing SPL approaches such as (Gomaa 2005; Clements & Northrop 2002; Pohl et al. 2005) capture variability at the subsystem, component, and connector level. They do not provide modeling views to support interconnecting design patterns together to form software architectures. This research created a new architectural view for modeling how design patterns are integrated together to form software architectures using UML interaction overview diagrams. This view also models variability in the design pattern

interconnections and captures the different ways the design patterns can and cannot be integrated together for a SPL member.

e) Feature and design pattern based validation

This research also presented a feature and design pattern, model based testing approach to ensure all features and design patterns are tested. As discussed in Chapter 9, this approach leverages reusable SPL test specifications following the CADeT process (Olimpiew 2008; Olimpiew & Gomaa 2009) to ensure that all features and subsequently all design patterns are covered in the test specifications. However, CADeT was extended to specifically address design pattern level variability and design time validation of executable software architectures. This is achieved by capturing SPL test specifications at higher level of abstraction and then refining them during the application engineering phase. Furthermore, the test data associated with specifications is mapped to expected state machine behaviors in the executable design patterns. Thus the flow of events and messages can be followed through the executable software architectures to ensure the actual behavior matches the expected behavior for test step in the test specifications. Any issues can then be identified and remedied before the architecture is implemented.

9.2 Future Work

This research has revealed several areas of further study, which are described in the following subsections.

9.2.1 Application to other subsystems in FSW SPL

First, this research can be extended to further demonstrate applicability in the FSW domain by modeling the other subsystems, such as the attitude control and telemetry, tracking and control subsystems. Building a complete FSW SPL would further demonstrate the scalability of the approach in industrial domains.

9.2.2 Application to other industrial domains

This research was applied to create two applications from an FSW SPL. The case studies were based on industrial real world spacecraft. However, to further demonstrate a wide applicability across the DRE domain, this research should be applied to other industrial domains, such as unmanned aerial vehicles (UAVs) and automobiles.

9.2.3 Automation

Another logical step for future work is to automate some of the manual processes in the proposed approach in order to more effectively scale the approach for large industrial SPLs. For example, during the application engineering phase after the features are selected, the SPL assets that are applicable to the application could be automatically instantiated based on the application's feature selection, as described in (Tawhid & D. Petriu 2008; Tawhid & D. Petriu 2011).

9.2.4 Extending approach to implementation phase

Another area of future work is to extend this research into the implementation phase. Currently the approach relies on existing techniques and process to translate the software architecture into an implementation. However, this approach can be extended with state-based code generators to create an implementation. For example, NASA's Jet Propulsion Laboratory has developed a tool for auto-coding UML Statecharts for specifically for flight software (Ed Benowitz et al. 2006; Wagstaff et al. 2008).

9.2.5 Design pattern based performance analysis

This research can be extended to address software performance analysis, since the ability to meet performance requirements is important for DRE systems. This can be achieved by leveraging existing SPL model driven performance analysis approach, such as (Street & Gomaa 2006).

APPENDIX A. DISTRIBUTED REAL-TIME AND EMBEDDED ARCHITECTURAL AND EXECUTABLE DESIGN PATTERNS

A.1 Hierarchical Control

The Hierarchical Control Pattern (Gomaa 2005) involves several control components. Each of these controllers manages a part of the system by conceptually executing a state machine. Additionally, there is also a coordinator component that orchestrates high level control by determining the next job for each of the controllers. This design pattern is included because it provides a control structure for DRE systems where overall control is centralized, but the work is executed by multiple controllers. This enables the workload to be distributed among multiple controllers, while still maintaining centralized control. This pattern is suitable for larger control applications because it provides high level coordination, as well as, a distributed workload.

The executable design pattern contains five main components as depicted in the collaboration diagram in Figure A-1. The Coordinator is responsible for determining the next job for each of the Controllers. The Controller is responsible for executing jobs it receives from the Coordinator. The jobs involve actuating an action by commanding the

Output_Components and IO_Components. Additionally, the Controller receives event notifications from multiple Input_Components and IO_Components and passes this information to the Coordinator. There is variability in the amount of Input_Components, Output_Components, and IO_Components that can be used when this pattern is applied. Therefore this variability is modeled using the <<optional>> stereotype and as a zero-or-more multiplicity in the collaboration diagram. There is also variability in the amount of Controllers, but at least one controller must be used. Thus, this variability is modeled using the <<kernel>> stereotype and one-or-many multiplicity.

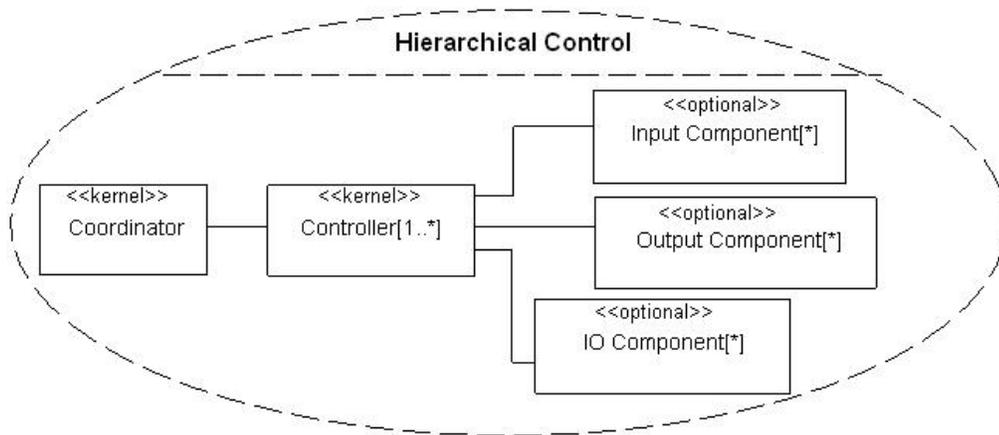


Figure A-1 Collaboration diagram for Hierarchical Control

A.2 Distributed Control

The Distributed Control Pattern (Gomaa 2005) involves several control components. Each of these controllers manages a part of the system by conceptually executing a state machine. However, in this design pattern there is no overall control. Instead the distributed controls have the option to send notification messages to peer controllers if there is information it needs to share with other controllers. This design pattern is included it provides a control structure for DRE systems where overall control is distributed among multiple independent controllers. This pattern is suitable for control applications where high-level coordination is not needed.

The collaboration diagram for the Distributed Control design pattern is shown in Figure A-2. This design pattern is composed of four main components. There are multiple Distributed_Controllers, each of which is responsible for controlling part of the system. These controllers manage their respective parts of the system interfacing with Input_Components, IO_Components, and Output_Components. The Distributed_Controllers are responsible for receiving event notifications from multiple Input_Components and IO_Components. The Distributed_Controllers are also responsible for executing the appropriate response, which involves commanding the Output_Components and IO_Components. Additionally, Distributed_Controllers can send messages to other controllers using peer-to-peer communication. There is variability in the amount of Input_Components, Output_Components, and

IO_Components that can be used when this pattern is applied. Therefore this variability is modeled using the <<optional>> stereotype and as a zero-or-more multiplicity in the collaboration diagram. There is also variability in the amount of Controllers, but at least one controller must be used. Thus, this variability is model using the <<kernel>> stereotype and one-or-many multiplicity.

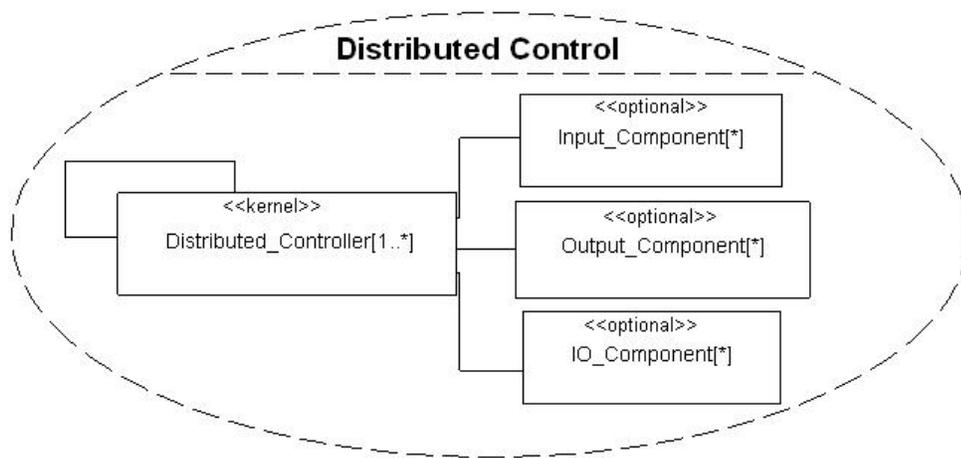


Figure A-2 Collaboration diagram for Distributed Control design pattern

A.3 Command Dispatcher

The Command Dispatcher design pattern (Dupire & Fernandez 2001) encapsulates requests as parameterized objects thereby decoupling the producer from the consumer. The parameterized command structure enables this design pattern to make efficient use out of memory. Additionally, this structure enables the easy addition of new commands

as well as undoable commands. This pattern is included because it provides a structure for commands and command handling. This pattern is suitable for DRE applications where decoupling the producer and consumer is desirable, or there is a need to support new commands, or there is a need to support undoable commands.

The collaboration diagram for the Command Dispatcher design pattern is depicted in Figure A-3. The Command Dispatcher design pattern consists of six components. In this design pattern there are multiple producers and consumers. When a Producer needs a command performed, it simply creates a `Concrete_Command` object with the action and action value specified. For example, an action could be “set_speed” and the value could be “5”. Then the Producer sends the `Concrete_Command` to the Invoker. The Invoker is responsible for determining when the `Concrete_Command` should be executed. When the Invoker is ready to execute a `Concrete_Command` it sends it to the `Command_Dispatcher`. The `Command_Dispatcher` then retrieves the action and value from the `Concrete_Command`. Using this information the `Command_Dispatcher` determines the appropriate `Command_Handler` to perform this action. The `Command_Dispatcher` then invokes the `Command_Handler` to perform the action. The `Command_Handler` in turn calls the required operations on the Consumer to perform the required action. All the components in this design pattern are required, thus they are all modeled with the <<kernel>> stereotype. However, there is some variability in the

amount of Producers, Consumers, and Command Handlers. This information is modeled with the one-or-many multiplicity.

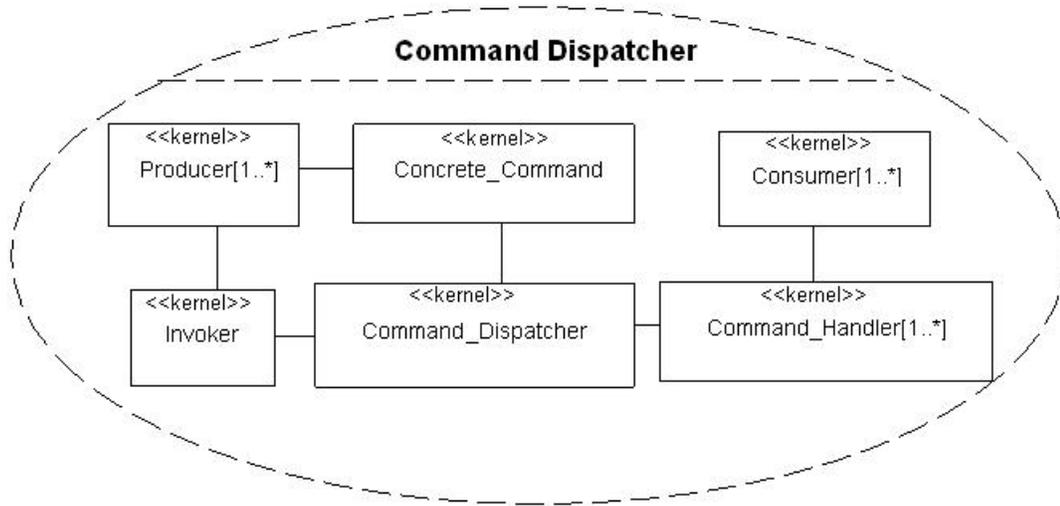


Figure A-3 Collaboration diagram for the Command Dispatcher design pattern

A.4 Pipes and Filters

The Pipes and Filters architectural design pattern, also referred to as Channel, (Buschmann et al. 1996; Douglass 2003) provides a structure for processing a stream of data in a serial fashion. This pattern is included because it provides a structure for DRE systems where the data processing algorithms and functions are encapsulated within a single structure called a pipe. The pipe is composed of data processing algorithm classes called filters. This enables all the data processing to be contained in a single component,

which makes it easy to understand and maintain. Additionally, since the algorithms are encapsulated in their own class, they can be independently modified which improves maintainability. This pattern is suitable for any type of DRE application that performs data processing.

The executable design pattern contains four main components as depicted in the collaboration diagram in Figure A-4. The Pipe component provided the overall structure for the data processing and interface to the external environment. It is composed of three main types of filters, an Input_Filter that reads the data from the data source, a Transformation_Filter that performs some processing on the data, and finally an Output_Filter that is responsible for sending out the final data. The Pipe contains only one Input_Filter and Output_Filter, but can have as many Transformation_Filters as necessary. All the components in this design pattern are required, thus they are all modeled with the <<kernel>> stereotype. However, there is some variability in the amount of Transformation Filters, thus it has a one-or-many multiplicity.

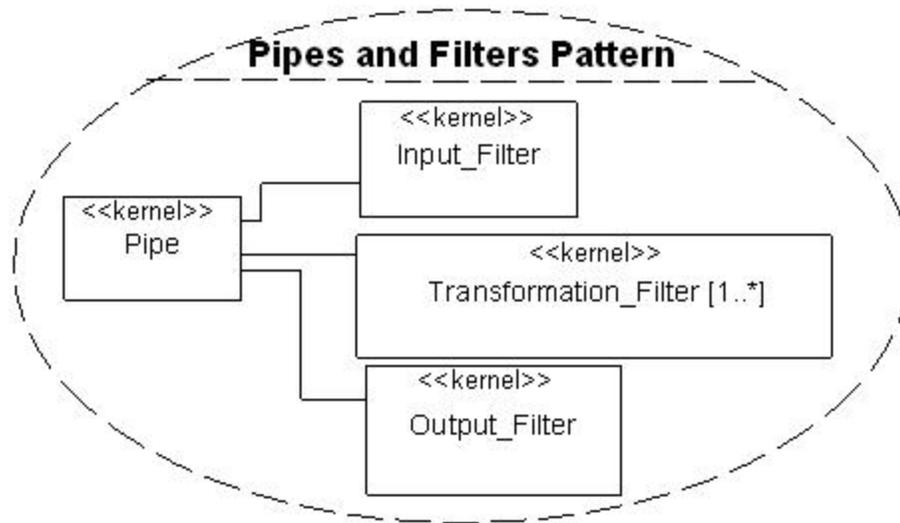


Figure A-4 Collaboration diagram for Pipes and Filters design pattern

A.5 Master Slave

The Master Slave design pattern provides a structure to perform distributed and parallel processing on data (Buschmann et al. 1996). The Master Slave design pattern is included because it provides a structure for DRE systems where the data processing can be performed in a parallel fashion. Processing is divided between master and slave components. This structure enables all the data processing to be performed in parallel, which helps to reduce the overall processing time. This pattern is suitable for any type of DRE application that performs data processing.

The executable design pattern contains two main components as depicted in the collaboration diagram in Figure A-5. The overall coordination of the data processing is handled by the Master component. The Master component distributes the work to identical Slave components for processing. Any final processing that is required after all the Slave components are finished is also handled by the Master. All the components in this design are required and thus are marked with the <<kernel>> stereotype. There is variability in the number of slaves, thus a one or many multiplicity is applied.

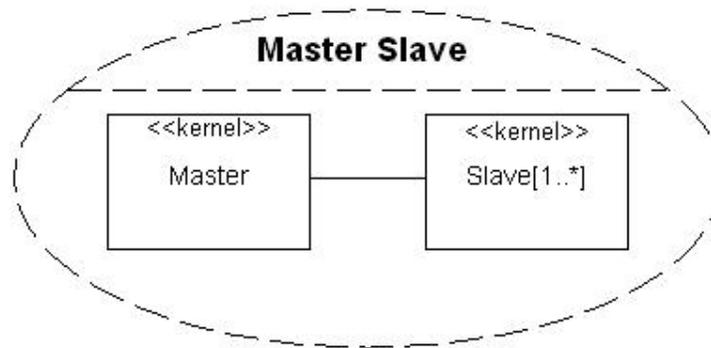


Figure A-5 Collaboration diagram for the Master Slave design pattern

A.6 Strategy

The Strategy design pattern (Gamma et al. 1995) is to create a set of interchangeable algorithms. This is accomplished by encapsulating each algorithm in a separate class and defining a common interface to all the algorithms. This pattern can help improve system

maintainability because algorithms can be easily interchanged and can be modified or added independently. This design pattern is included because it provides a structure for handling data processing algorithms. This design pattern is suitable for any DRE system that performs data processing.

The collaboration diagram for the Strategy design pattern is depicted in Figure A-6. The Strategy design pattern consists of two components. The IStrategy interface object defines the interface that Concrete_Strategy objects must follow. The Concrete_Strategy components are the classes that encapsulate the algorithms. A dependency is drawn between the Concrete_Strategy and IStrategy because the Concrete_Strategy realizes the IStrategy interface. All the components in this design are required and thus are marked with the <<kernel>> stereotype. There is variability in the number of Concrete_Strategy objects, thus a one or many multiplicity is applied.

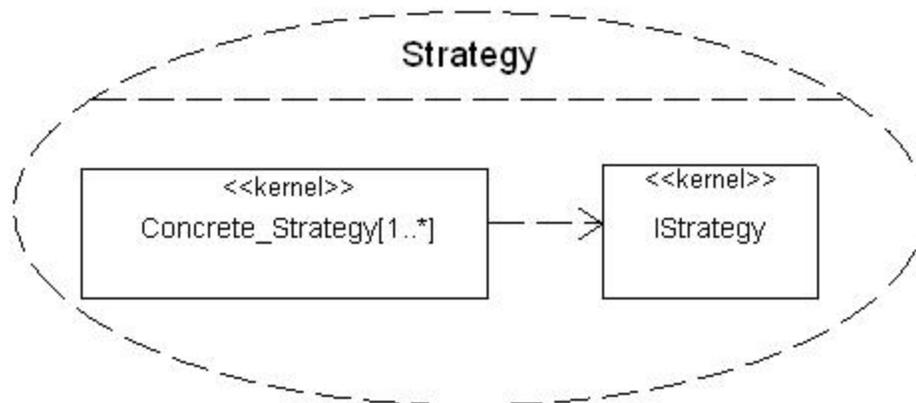


Figure A-6 Collaboration diagram for the Strategy design pattern

A.7 Two Phase Commit

The Two Phase Commit design pattern (Gomaa 2005) is used to synchronize data updates on distributed nodes when all the data must be updated at the same time. This design pattern is included because it provides the ability to update shared data on distributed nodes in an atomic fashion. This design pattern is suitable for DRE systems where atomic transactions are needed for distributed nodes.

The collaboration diagram for the Two Phase Commit design pattern is depicted in Figure A-7. It can be seen that the Two Phase Commit design pattern is composed of two components, which are the Commit_Server and the Participant. There are multiple Participants, each of which is responsible for committing a piece of data. The

Commit_Server is responsible for ensuring that either all the Participants commit the data or they all do not commit the data. All the components in this design are required and thus are marked with the <<kernel>> stereotype. There is variability in the number of Participants, thus a one or many multiplicity is applied.

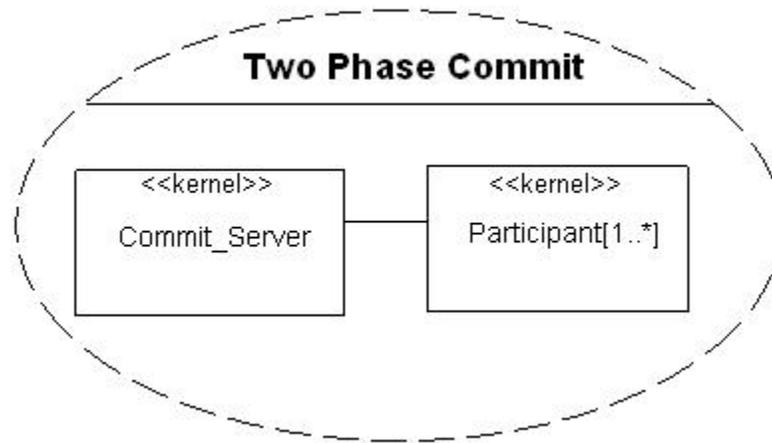


Figure A-7 Collaboration diagram for the Two Phase Commit design pattern

A.8 Compound Commit

The Compound Commit design pattern (Gomaa 2005) is used to synchronize data updates on distributed nodes when the data can be broken down into smaller pieces and updated independently. This design pattern is included because it provides the ability to update shared data on distributed nodes using several atomic transactions. This design

pattern is suitable for DRE systems where data can be broken down into smaller independent atomic transactions on distributed nodes.

The collaboration diagram for the Compound Commit pattern is shown in Figure A-8.

The design pattern is composed of Commit_Servers and Participants. There are multiple Participants each of which is responsible for committing a piece of data. The Commit_Server is responsible for ensuring that each of smaller transactions is completed. All the components in this design are required and thus are marked with the <<kernel>> stereotype. There is variability in the number of Participants, thus a one or many multiplicity is applied.

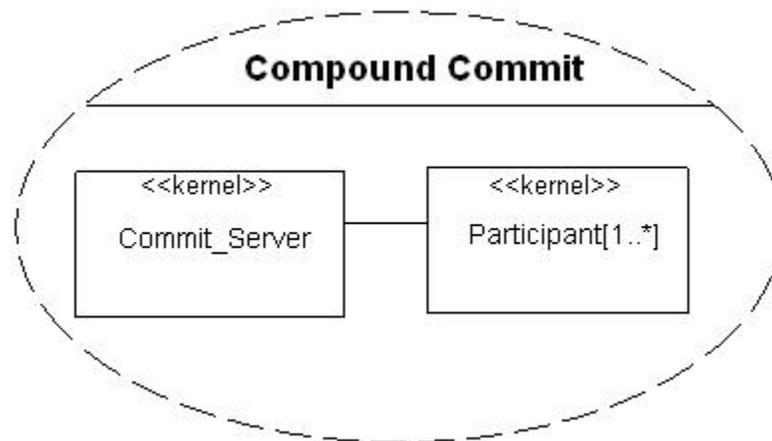


Figure A-8 Collaboration diagram for the Compound Commit design pattern

A.9 Client Server

The Client Server design pattern, also seen as the Multiple Client Multiple Server and Multitier Client Server, (Gomaa 2005) is used to help organize and distribute processing of data. This design pattern is composed of clients that request services from servers.

The Client Server design pattern is included because it can be used provide client driven data collection activities. This design pattern is suitable to use on DRE systems that require distributed processing.

The Client Server collaboration diagram is depicted in Figure A-9. This design pattern is composed of three components, which are the Client, Server, and Multitier Client Server component, which acts as both client and server. Any number of clients, servers, or multitier client server components can be used. Therefore its variability is modeled using the <<optional>> stereotype and zero-or-more multiplicity.

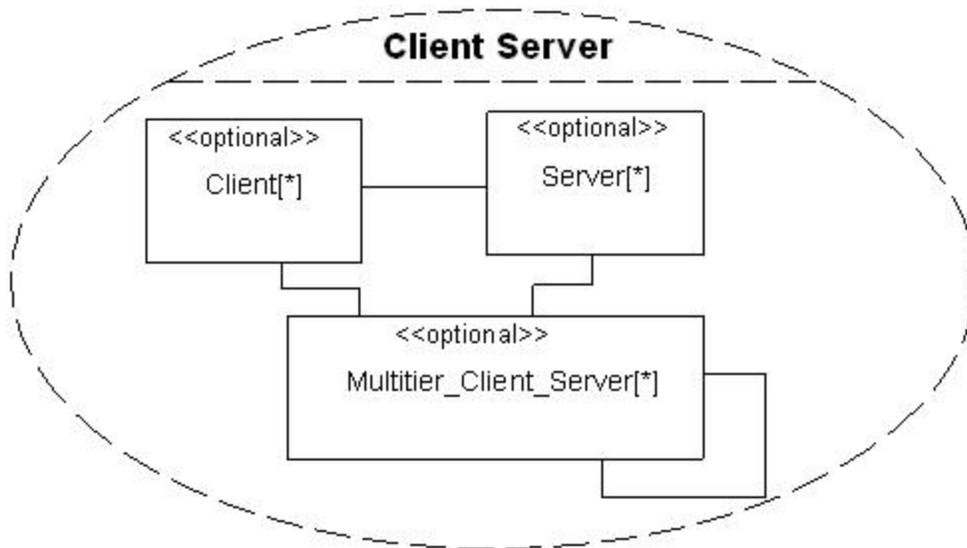


Figure A-9 Collaboration diagram for the Client Server design pattern

A.10 Publish Subscribe

The Publish Subscribe design pattern (Gomaa 2005; Buschmann et al. 1996; Gamma et al. 1995), also called Observer and Subscription/Notification, is an event driven, selective form of group communication. In this design pattern clients subscribe to receive a specific type of message and the subscription server notifies these clients with these messages as they occur. When a client is no longer interested in the data, it can simply unsubscribe to the data. This design pattern is included because it can be used to provide a dynamic way to push data to the appropriate clients. This design pattern is suitable to use on DRE applications where data is pushed from the source to the clients using

selective group communication in which the client's interest in data may change over time.

The collaboration diagram for the Publish Subscribe design pattern is depicted below in Figure A-10. This design pattern is composed of multiple Clients and one Subscription_Server. The Clients subscribe to messages they are interested in through the Subscription_Server. When the message occurs, the Subscription_Server forwards the information only to the interested Clients. All the components in this design are required and thus are marked with the <<kernel>> stereotype. There is variability in the number of Clients, thus a one or many multiplicity is applied.

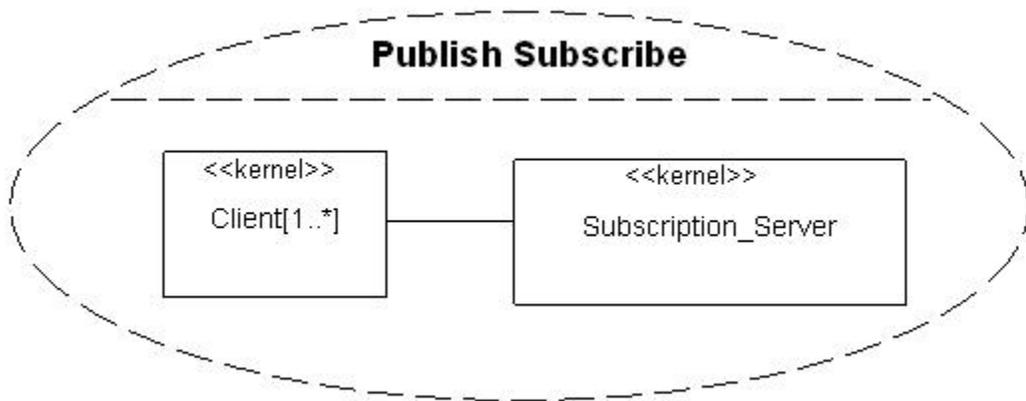


Figure A-10 Collaboration diagram for the Publish Subscribe design pattern

A.11 Broadcast

The Broadcast design pattern (Gomaa 2005) is the simplest form of group communication where a server sends its information to all clients. It is up to the client to use this information or to discard the information if it is not interested in the message.

This design pattern is included because it can be used provide a simple way to push data to all clients. This design pattern is suitable to use on DRE applications where data is pushed from the source to the clients and there is extra bandwidth to support the unnecessary messages.

The collaboration diagram for the Broadcast design pattern is depicted in Figure A-11.

This design pattern is composed a single Broadcast_Component. When the Broadcast_Component has new information it simply sends the information to all other components. When another component receives the data, it must either discard the information if it is not useful or act on the information if it is useful.

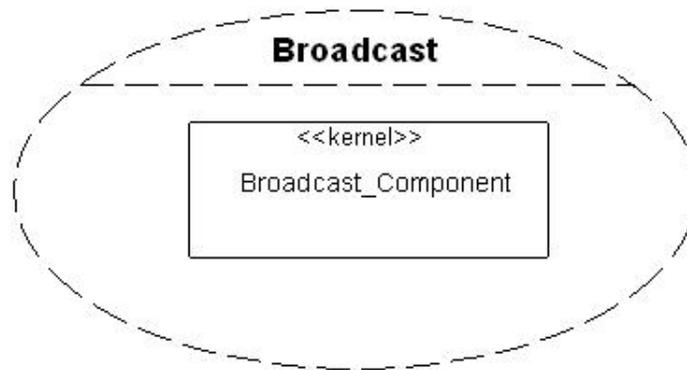


Figure A-11 Collaboration diagram for the Broadcast design pattern

A.12 Multicast

The Multicast design pattern (Gomaa 2005) is a type of group communication where a server sends information to a predetermined set of clients. This design pattern is included because it can be used to provide a simple way to push data to the just the appropriate clients. This design pattern is suitable to use on DRE applications where data is pushed from the source to the clients and the interested set of clients will not change.

The collaboration diagram for the Multicast design pattern is depicted in Figure A-12. This design pattern is composed of a single Multicast_Component, thus it is modeled with the <<kernel>> stereotype. When the Multicast_Component has new information it simply sends the information to a predetermined set of receivers.

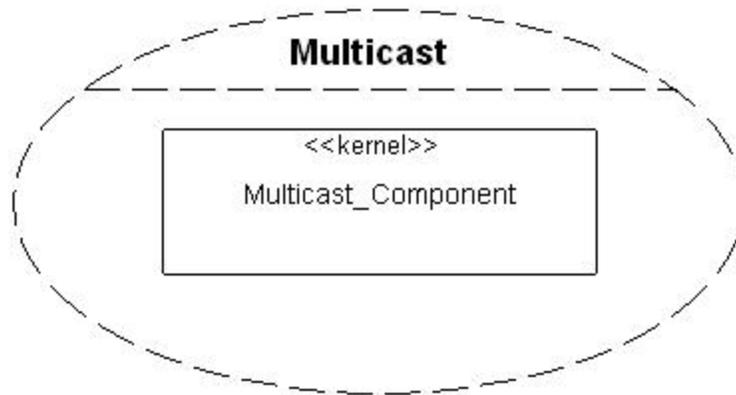


Figure A-12 Collaboration diagram for the Multicast design pattern

A.13 Protected Single Channel

The Protected Single Channel (Douglass 2003) is a simple and inexpensive pattern to achieve reliability through key point checks. The key point checks enable the detection of single event faults. This design pattern is included because it provides a level of reliability to the Pipes and Filter design pattern. This design pattern is suitable to use on DRE applications which do not have the need to function in the presences of a persistent fault.

The collaboration diagram for the Protect Single Channel design pattern is depicted in Figure A-13. It can be seen that this design pattern requires the Pipes and Filters design pattern therefore it is composed of the four components from the Pipes and Filters design pattern plus an additional Data_Validation component. The Data_Validation component

encapsulates the key point checks and is used by the Pipe during data processing. There is variability in the amount of Data_Validation and Transformation_Filters that can be used, but at least one of each type must be used. Therefore this variability is modeled using the <<kernel>> stereotype and one-or-many multiplicity.

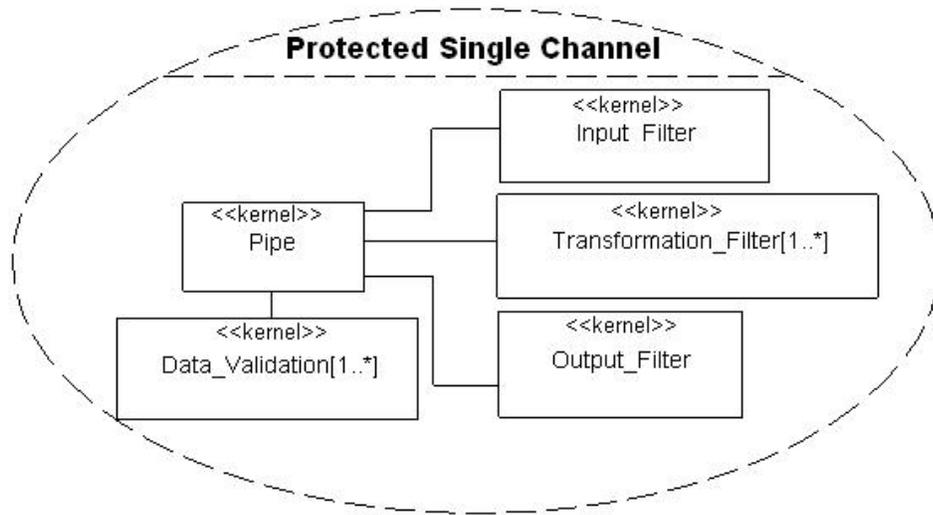


Figure A-13 Collaboration diagram for Protected Single Channel design pattern

A.14 Homogeneous Redundancy

The Homogeneous Redundancy design pattern (Douglass 2003) is a pattern to improve reliability using multiple identical processing pipes. If a problem is detected in the first pipe, then the back-up pipe can be switched on. This design pattern is included because it provides a level of reliability to the Pipes and Filter design pattern. This design pattern

is suitable to use on DRE applications that need to be able to function in the presences of a fault and can tolerate data loss in between deactivating the primary pipe and activating the backup pipe.

Figure A-14 depicts the collaboration diagram for the Homogeneous Redundancy design pattern. Since this pattern requires the Pipes and Filters design pattern, it contains the four components from the Pipes and Filters design pattern. In this collaboration diagram the multiplicity on the Pipe component changed to reflect the use of two identical Pipe components in this pattern. Additionally Figure A-14 contains Data_Validation and Actuation_Validation components. The Data_Validation components encapsulate key point checks to detect faults during data processing. The Actuation_Validation component provides an additional check to ensure the commanded output is the same as the actual output. This provides another layer of fault checking in this design pattern. If either the Data_Validation or the Actuation_Validation components detect a problem, then the primary pipe is deactivated and the secondary pipe is activated. There is variability in the amount of Data_Validation and Transformation_Filters that can be used, but at least one of each type must be used. Therefore this variability is modeled using the <<kernel>> stereotype and one-or-many multiplicity.

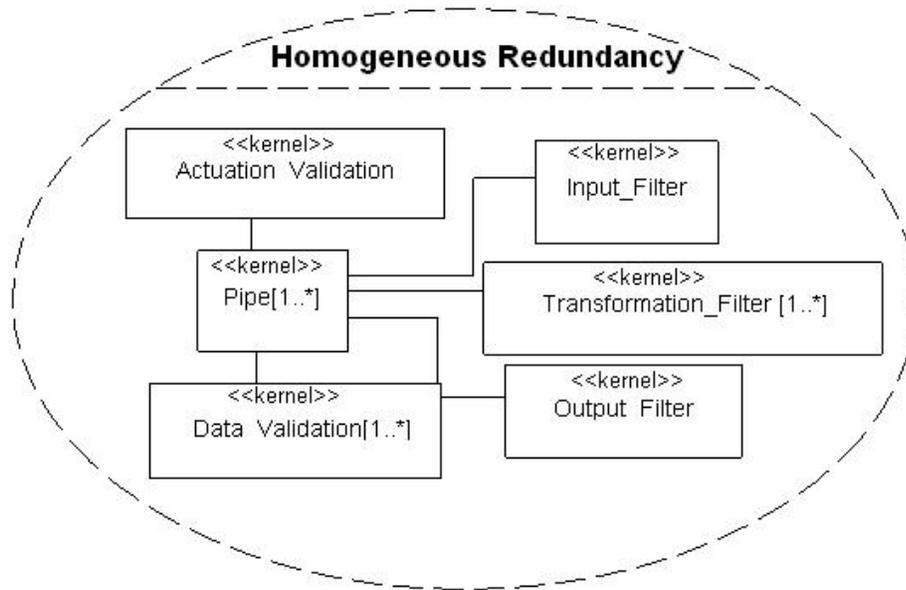


Figure A-14 Collaboration diagram for the Homogeneous Redundancy design pattern

A.15 Triple Modular Redundancy

The Triple Modular Redundancy (TMR) design pattern (Douglass 2003) is a pattern to improve reliability using multiple identical processing pipes. However, when using this pattern there is no data loss if a fault is detected. This design pattern is included because it provides a level of reliability to the Pipes and Filter design pattern. This design pattern is suitable to use on DRE applications that need to be able to function in the presence of a fault and cannot tolerate data loss if a fault is detected.

The collaboration diagram for the Protect Single Channel design pattern is depicted in Figure A-15. It can be seen that this design pattern is required the Pipes and Filters design pattern therefore it is composed of the four components from the Pipes and Filters design pattern. However, on this collaboration diagram the multiplicity on the Pipe component changed to reflect the use of three identical Pipe components in this pattern. Additionally, this design pattern contains the Comparator component. The Comparator component compares the output from the three Pipes. If a fault has occurred in one of the channels, then the output from one of the pipes will not agree with the others. In this case, the Comparator uses a majority take all policy and discards the output from the faulty pipe. This structure ensures that no data is lost when a fault is detected. All the components are required therefore they are modeled with the <<kernel>> stereotype. There is variability in the amount of Transformation_Filters that can be used, but at least one of each type must be used. Therefore this variability is modeled using the one-or-many multiplicity.

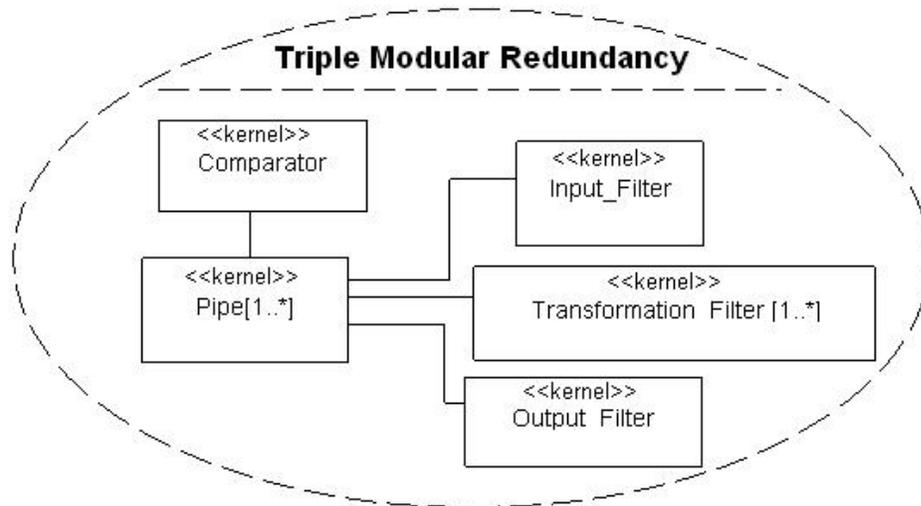


Figure A-15 Collaboration diagram for the Triple Modular Redundancy design pattern

A.16 Heterogeneous Redundancy

The Heterogeneous Redundancy design pattern (Douglass 2003) is a pattern to improve reliability using multiple different processing pipes. If a problem is detected in the first pipe, then the back-up pipe can be switched on. Using two different pipes enables this design pattern to not only detect random faults but is also prevents against systematic faults because the pipes are different. This design pattern is included because it provides a level of reliability to the Pipes and Filter design pattern. This design pattern is suitable to use on DRE applications that need to be able to function in the presences of random and systematic faults and can tolerate data loss in between deactivating the primary pipe and activating the backup pipe.

Figure A-16 depicts the collaboration diagram for the Heterogeneous Redundancy design pattern. Since this pattern requires the Pipes and Filters design pattern, it contains the four components from the Pipes and Filters design pattern. Since this design pattern requires two different patterns the components from the Pipes and Filters design pattern appear twice. One is the primary pipe and the other is the secondary pipe. Additionally Figure A-16 contains Data_Validation and Actuation_Validation components. The Data_Validation components encapsulate key point checks to detect faults during data processing. The Actuation_Validation component provides an additional check to ensure the commanded output is the same as the actual output. This provides another layer of fault checking in this design pattern. If either the Data_Validation or the Actuation_Validation components detect a problem, then the primary pipe is deactivated and the secondary pipe is activated. All the components are required therefore they are modeled with the <<kernel>> stereotype. There is variability in the amount of Transformation_Filters, Data_Validation, Secondary_Transformation_Filters, and Secondary_Data_Validation that can be used, but at least one of each type must be used. Therefore this variability is modeled using the one-or-many multiplicity.

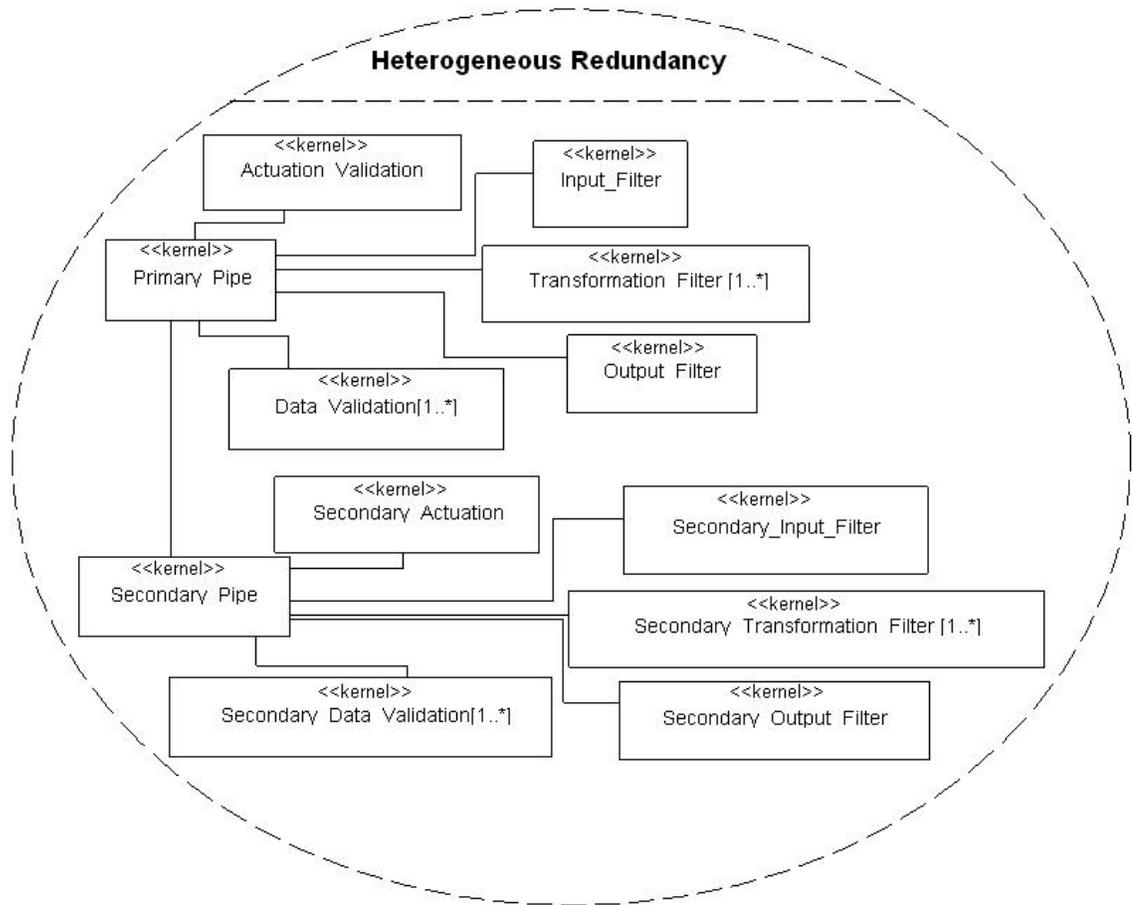


Figure A-16 Collaboration diagram for Heterogeneous Redundancy design pattern

A.17 Monitor-Actuator

The Monitor-Actuator design pattern (Douglass 2003) is a pattern to improve reliability using an independent sensor that watches the actuation. If a problem is detected, then the system is put into a failsafe state, which is a state that is always known to be safe. This design pattern is included because it provides a level of reliability to the Pipes and Filter

design pattern. This design pattern is suitable to use on DRE applications that have reliability requirements but do not have high availability requirements.

The collaboration diagram for the Monitor-Actuator design pattern is depicted in Figure A-17. The Monitor-Actuator design pattern contains eight components. The Actuation_Channel is composed of a Sensor_Input_Processing component that is the interface component for the actuation device source, the Data_Transformation component that performs processing on the data, the Data_Integrity_Check component that checks for faults, and the Output_Processing component that is the interface component to the actuator. In addition to the Actuation_Channel, there is also a Monitor_Channel. The Monitor_Channel is composed of a Monitor that compares the commanded output with output received from an actuator monitor sensor. If a problem is detected, then the Monitor_Channel sends a message to the Actuation_Channel to go into a failsafe state. All components are required, thus the <<kernel>> stereotype added to all components. There is variability in the amount of Data_Integrity_Checks and Data_Transformation components that can be modeled, which is captured with the one-or-many multiplicity.

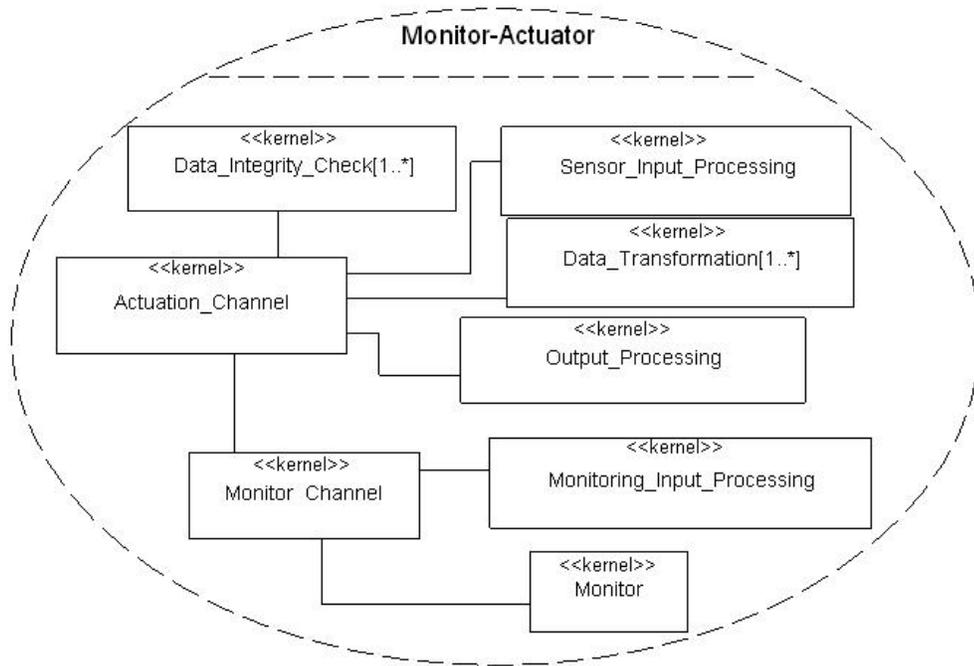


Figure A-17 Collaboration diagram for the Monitor-Actuator design pattern

A.18 Sanity Check

The Sanity Check design pattern (Douglass 2003) is a pattern to improve reliability that ensures the system is performing more or less as expected. If a problem is detected, then the system is put into a failsafe state, which is a state that is always known to be safe.

This design pattern is very similar to the Monitor-Actuator design pattern described in Section A.17. The only difference between this pattern and the Monitor-Actuator is that the Sanity Checks ensures the actuation is in an acceptable range, while the Monitor-Actuator ensures the values are exact. This design pattern is included because it provides

a level of reliability to the Pipes and Filter design pattern. This design pattern is suitable to use on DRE applications that have reliability requirements but do not have high availability requirements.

The collaboration diagram for the Sanity Check design pattern is depicted in Figure A-18. The Sanity Check design pattern contains eight components. The Actuation_Channel is composed of a Sensor_Input_Processing component that is the interface component for the actuation device source, the Data_Transformation component that performs processing on the data, the Data_Integrity_Check component that checks for faults, and the Output_Processing component that is the interface component to the actuator. In addition to the Actuation_Channel, there is also a Monitor_Channel. The Monitor_Channel is composed of a Monitor that compares the commanded output with output received from an actuator monitor sensor. If the values are outside the acceptable range, then the Monitor_Channel sends a message to the Actuation_Channel to go into a failsafe state. All components are required, thus the <<kernel>> stereotype is added to all components. There is variability in the amount of Data_Integrity_Checks and Data_Transformation components that can be modeled, which is captured with the one-or-many multiplicity.

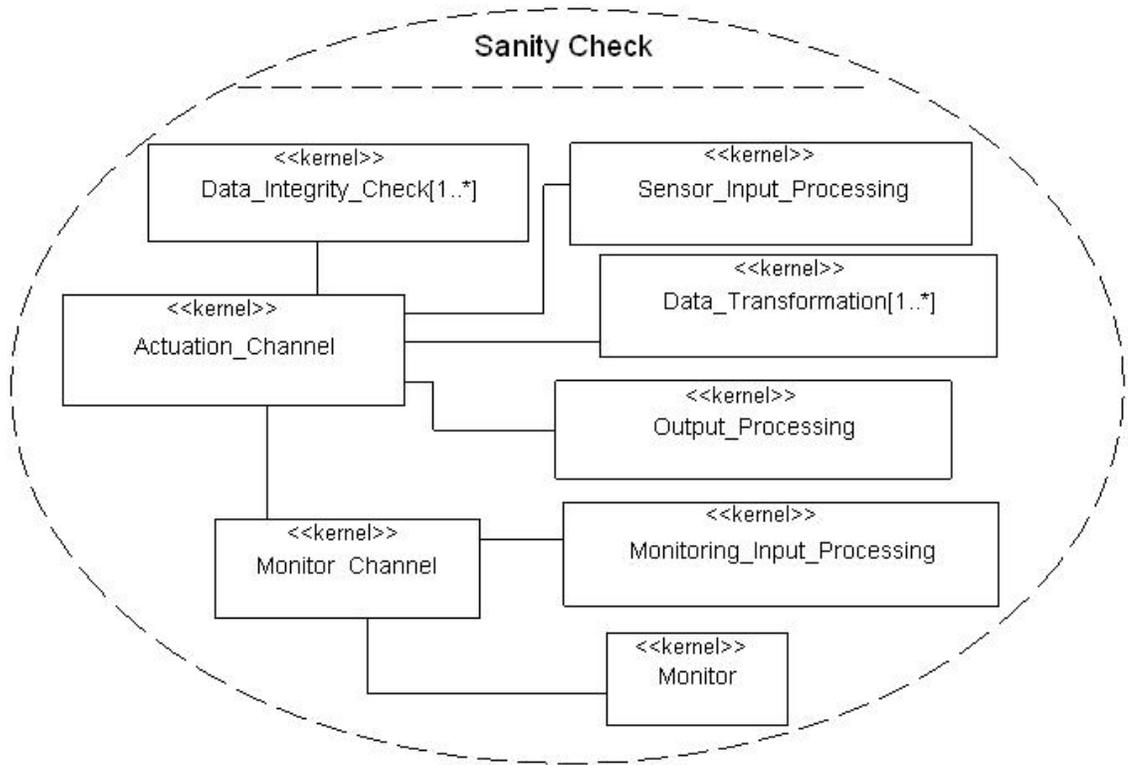


Figure A-18 Collaboration diagram for the Sanity Check design pattern

A.19 Watchdog

The Watchdog design pattern (Douglass 2003) is a lightweight design pattern to improve system reliability by making sure the processing is going as expected. This design pattern is included because it provides a lightweight approach to providing reliability.

This design pattern is suitable to use on DRE applications that have reliability requirements.

The collaboration diagram for the Watchdog design pattern is depicted in Figure A-19. It is composed of a single component called the Watchdog. When the process is going as expected, the Watchdog receives stroking messages from the component it is monitoring. If it does not receive a stroking message within a given amount of time, the watchdog assumes a fault has occurred and sends out an alarm.

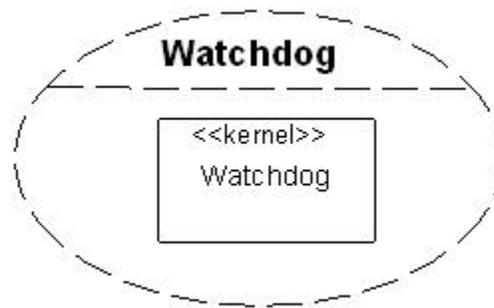


Figure A-19 Collaboration diagram for the Watchdog design pattern

A.20 Layers

The Layers (Gomaa 2005; Douglass 2003; Buschmann et al. 1996) design pattern is used to structure software architectures into groupings. In a layered architecture, the upper level layers rely on services provided by the lower layers. This enables upper layer components to be modified independently of the lower layers. This design pattern is included in because it provides a structure for organizing software architectures. This pattern is suitable to use on any DRE application. There is no collaboration diagram for

this design pattern because it does not contain any components. Rather it is composed of layers that are used to organize and structure other components.

A.21 Five-Layer Architecture

The Five-Layer Architecture (Douglass 2003) is a specialized version of the Layers design pattern that is suitable for DRE applications. The Five-Layer Architecture is organized into five specific layers, which are the Application, User Interface, Communication, Abstract Operating System, and Abstract Hardware. This design pattern is included because it provides a structure for organizing software architectures. This pattern is suitable to use on any DRE application. There is no collaboration diagram for this design pattern because it does not contain any components. Rather it is composed of layers that are used to organize and structure other components.

A.22 Asynchronous Message Communication

The Asynchronous Message Communication (Gomaa 2005) design pattern is a form of communication where the producer does not wait on the consumer and the producer does not need a reply. This design pattern is included because it specifies how component communication will occur. This design pattern is suitable for any DRE system.

Figure A-20 depicts the collaboration diagram for the Asynchronous Message Communication design pattern. There is one component, which is the Message Queue connector. It is designed as a monitor that encapsulates a message queue and synchronizes the operations to send and receive messages. The producer is suspended if

the queue is full and reactivated when the slot becomes available. After the producer successfully adds a message to the queue, it continues its execution. The consumer is suspended if the message queue is empty and reactivated when a message is added to the queue (Gomaa 2000).

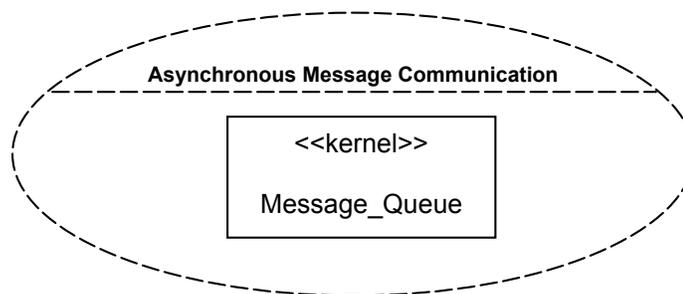


Figure A-20 Collaboration diagram for Asynchronous Message Communication design pattern

A.23 Bidirectional Asynchronous

The Bidirectional Asynchronous Message Communication (Gomaa 2005) design pattern is a form of communication where the producer does not wait on the consumer, but the producer does eventually need to receive a reply from the consumer. This design pattern is included because it specifies how component communication will occur. This design pattern is suitable for any DRE system.

The Bidirectional Asynchronous Message Communication design pattern is composed of two components depicted in Figure A-21. The connectors are designed as monitors that encapsulate a message queue and synchronize the operations to send and receive messages. The Message_Queue is used for sending messages from the Producer to the Consumer. Using this connector, the producer is suspended if the queue is full and reactivated when the slot becomes available. The consumer is suspended if the message queue is empty and reactivated when a message is added to the queue (Gomaa 2000). The Non_Blocking_Message_Queue is used for sending replies from the Consumer to the Producer. Using this connector, the consumer is suspended if the queue is full and reactivated when the slot becomes available. After the consumer successfully adds a reply to the queue, it continues its execution. The producer must periodically check the queue for incoming replies because it is not suspended if the message queue is empty.

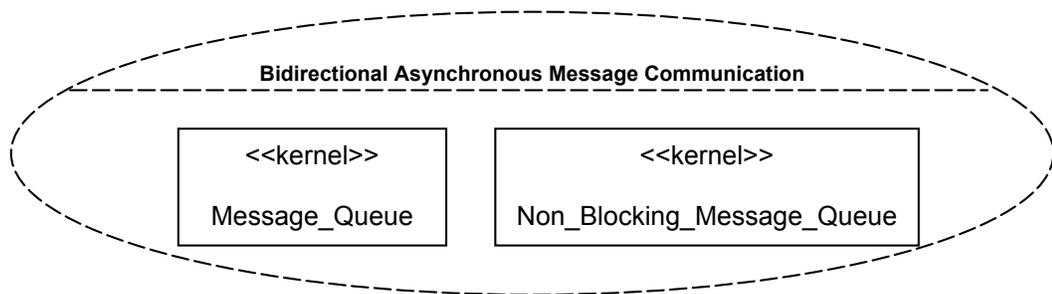


Figure A-21 Collaboration diagram for Bidirectional Asynchronous Message Communication design pattern

A.24 Asynchronous Communication with Callback

The Asynchronous Communication with Callback (Gomaa 2005) design pattern is a form of communication where the producer does not wait on the consumer and the producer does not need a reply. In this design pattern, the producer sends a callback handle to the consumer along with the message. After the consumer finishes, the producer then uses the handle to call the producer's operation remotely. This design pattern is included because it specifies how component communication will occur. This design pattern is suitable for any DRE system.

Figure A-22 depicts the collaboration diagram for the Asynchronous Message Communication with Callback design pattern. The Message_Queue connector is designed as a monitor that encapsulates a message queue and synchronizes the operations to send and receive messages. The producer is suspended if the queue is full and reactivated when the slot becomes available. After the producer successfully adds a message and a callbackHandle to the queue, it continues its execution. The consumer is suspended if the message queue is empty and reactivated when a message is added to the queue. After the consumer finishes and is ready to send its response, it uses the callback handle to call the operation remotely.

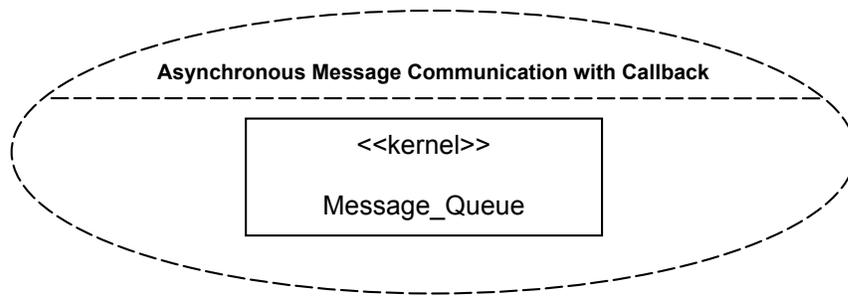


Figure A-22 Collaboration diagram for Asynchronous Message Communication with Callback design pattern

A.25 Synchronous with Reply

The Synchronous with Reply (Gomaa 2005) design pattern is a form of communication where the producer must wait on the consumer to finish and reply. This design pattern is included because it specifies how component communication will occur. This design pattern is suitable for any DRE system.

The collaboration diagram for the Synchronous Communication with Reply is depicted in Figure A-23. The Message_Buffer_And_Response connector is a monitor that encapsulates a message and response buffer. The connector synchronizes the operations to send, receive, and reply. After a producer sends a message, it is suspended until a reply is posted from the consumer. The consumer is suspended if the message buffer is empty (Gomaa 2000).

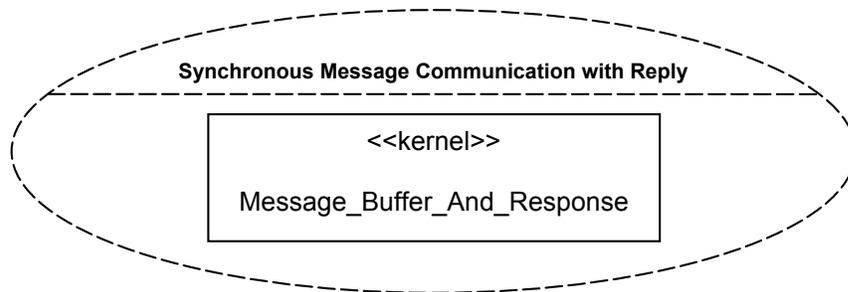


Figure A-23 Collaboration diagram for Synchronous Communication with Reply design pattern

A.26 Synchronous without Reply

The Synchronous without Reply (Gomaa 2005) design pattern is a form of communication where the producer must wait on the consumer to receive the message and the producer does not need to receive a reply. This design pattern is included because it specifies how component communication will occur. This design pattern is suitable for any DRE system.

Figure A-24 depicts the collaboration diagram for the Synchronous Communication without Reply design pattern. The Message_Buffer component is a connector, which is a monitor that encapsulates a message buffer and synchronizes the operations to send and receive. After a producer sends a message, it is suspended until the consumer receives

the message. The consumer is suspended if the message buffer is empty (Gomaa 2000).

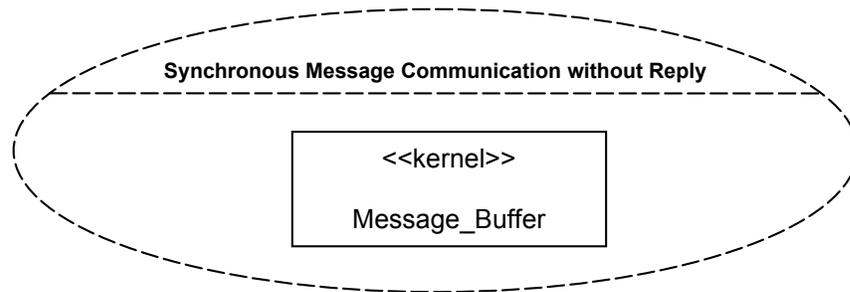


Figure A-24 Collaboration diagram for Synchronous Communication without Reply design pattern

A.27 Brokered Communication

The Brokered Communication (Gomaa 2005; Buschmann et al. 1996; Douglass 2003) design pattern is a form of communication where a broker acts as an intermediary between client and servers. This design pattern is included because it provides a flexible way to structure communication between clients and servers. This design pattern is suitable to use on DRE systems that require location transparency.

The collaboration diagram for the Brokered Communication design pattern is depicted in Figure A-25. This design pattern is composed of three components, which include the Client, Broker, and Server. The Client sends a request to the Broker. The brokered

communication can occur in two ways. In the first approach, the Broker looks up the location of the Server and forwards the request to the Server. The Server processes the request and sends the response back to the Broker. The Broker in turn sends the response back to the Client. In the second approach, the Broker looks up the location of the Server and sends a service handle back to the Client. The Client then uses the handle to make a request to the Server and the Server responses back to the Client. All components in this design pattern are required; however there is variability in the number of clients and servers which is modeled with the one-or-more multiplicity.

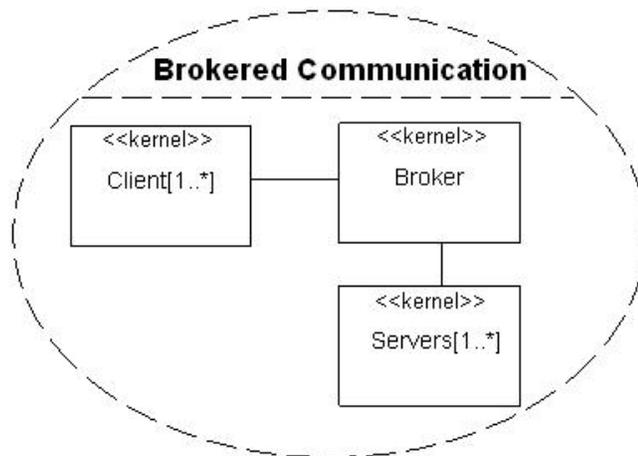


Figure A-25 Collaboration diagram for the Broker Forwarding design pattern

APPENDIX B. UNMANNED SPACEFLIGHT SOFTWARE PRODUCT LINE

B.1 Unmanned Spacecraft Flight Software Product Line Use Case Model

This subsection describes the use case model for the FSW SPL. With the reverse engineering approach, first the uses cases are developed for the selected FSW systems. Then they are analyzed to identify the kernel and variable uses cases for the FSW SPL.

B.1.1 Small Spacecraft System Problem Description

The small spacecraft system is a low Earth orbiting spacecraft that relies on ground station commanding to perform its operations. To control the spacecraft's attitude, the FSW takes attitude measurements and downlinks them to the ground station for processing. Engineers on the ground then determine the appropriate actions that need to be taken to adjust the spacecraft's attitude. Then these commands are uplinked back to the FSW to execute. Similarly, the FSW collects housekeeping data from all the onboard hardware and downlinks it to the ground for analysis. If adjustments are needed to the hardware, then the ground station will uplink the appropriate commands back to the FSW to execute.

The spacecraft's hardware design is undemanding and only requires minimal commanding from the FSW. The spacecraft's payload devices are single sensors that constantly collect data. Therefore they require minimal commanding from the FSW. The thermal control is completely passive thus does not communication FSW. Additionally, the spacecraft does not need to perform any maneuvers and it placed into its orbit. Consequently, it does not have any onboard propulsion.

B.1.2 Large Spacecraft System Problem Description

The large spacecraft system has an interplanetary mission. By necessity, the FSW needs to operate the spacecraft autonomously. This is because in deep space it will have limited contacts with the ground station, which makes ground station based real-time commanding near impossible. Thus, the FSW is not only responsible to take attitude and orbit measurements, but it also must determine and execute the appropriate adjustments. Additionally, the FSW is responsible for logging and reporting its actions to the ground station. This information will be included during its scheduled contacts with the ground station. Similarly, the FSW collects housekeeping data from all the onboard hardware and analyzes it for faults or problems with the hardware. If a problem is detected, the FSW takes and logs the corrective actions it performs. This log will be downlinked to the ground station during the next contact.

The spacecraft hardware design is more complex due to its deep space mission. As the spacecraft travels through deep space, it needs to maintain a specific temperature range to ensure the hardware performs as expected. To handle the harsh environment as it travels through space, the spacecraft needs active thermal control. The FSW will manage the thermal control using thermistors, heaters, and louvers. Additionally, to achieve its final orbit, the FSW must follow and execute a flight plan. This will require use of onboard propulsion and guidance and control algorithms.

Finally, the large spacecraft will contain several payload packages that are composed of multiple sensors. The payload packages are quite complex and some even require maneuvers from the spacecraft to take optimal measurements. Therefore the spacecraft's payload will require significant commanding from the FSW.

B.1.3 Time-Triggered Spacecraft System Problem Description

The Time-Triggered Spacecraft System will be designed to be time-driven rather than event driven. NASA is moving toward time-triggered architectures and other new technologies because “NASA says commercially available products, technologies and standards such as TTEthernet will help protect system design investments and can generate significant savings in capital-intensive programs over very long life-cycles” (Walko 2009). The Time-Triggered Spacecraft System will have the similar functional requirements and hardware devices as the Large Spacecraft System, however it is required to have strict temporal predictability.

B.1.4 Small Spacecraft System Use Case Model

The use case model for Small Spacecraft System is depicted in Figure B-1. It is composed of four use cases which are briefly described here:

- 1. Collect Spacecraft Data.** When commanded by the ground station, the FSW must collect, format, and store a low volume of payload and housekeeping data.
- 2. Execute Commands.** After the FSW receives a command load from the ground station, the FSW must store, validate, and determine the order of execution for ground commands that is appropriate for the spacecraft's current mode. Additionally, the FSW must also execute ground commands. In order to know when to execute the commands the FSW must maintain a spacecraft clock. The required volume of commands that must be processed by the software is low.
- 3. Respond to Spacecraft Faults.** Periodically check spacecraft IO devices for faults. If a fault is detected, the FSW should log the problem and report it to the ground station during the next contact. If the fault involves the critical spacecraft hardware, the FSW should switch to the redundant hardware and log that this action was performed. If the problem prevents that spacecraft from operating properly, then the FSW must put the spacecraft in safe mode, log that this action was performed, and report this information to the ground station during the next contact.

4. **Uplink/Downlink Telemetry.** During prescheduled contacts with the ground, the FSW must be able to send a small volume of real-time and playback telemetry to the ground station. The FSW must also be able to receive telemetry from the ground station through the spacecraft's receiver and then process the telemetry data.

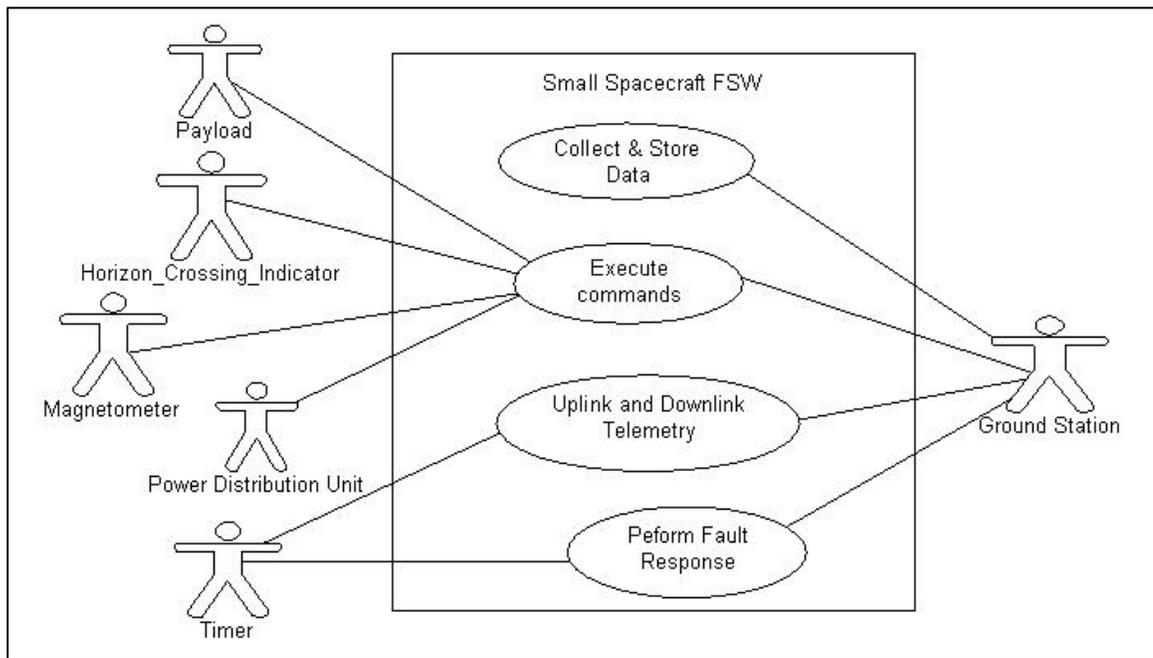


Figure B-1 Small Spacecraft System use cases

B.1.5 Large Spacecraft System Use Case Model

The use case model for the Large Spacecraft System is depicted in Figure B-2. It is composed of ten use cases, which are briefly described here:

1. **Collect Spacecraft Data.** When commanded by the ground station, the FSW must collect, format, and store a high volume of housekeeping data. The payload data and its housekeeping data will be collected by the FSW at a predetermined update rate. The FSW must have flexibility built into its formatting algorithms. This is because the spacecraft has a long life span and changes to the telemetry format standard are anticipated during its lifetime.
2. **Execute Commands.** After the FSW receives a command load from the ground station, the FSW must store, validate, and determine the order of execution that is appropriate for the spacecraft's current mode. Additionally, the FSW must also execute the ground commands. In order to know when to execute the commands the FSW must maintain a spacecraft clock. The required volume of commands that must be processed by the software is high. Additionally, FSW requires the ability to support new commands using the spacecraft hardware's existing interfaces to achieve its mission.
3. **Perform Fault Response.** Periodically check spacecraft IO devices for faults. If a fault in the spacecraft's hardware is detected, the FSW should log the problem and attempt to correct the problem. If the problem cannot be corrected and it prevents that spacecraft from operating properly, then the FSW must put the spacecraft in safe mode. All faults and actions taken in response to a fault are logged and reported to the ground station during the next contact.

4. **Uplink/Downlink Telemetry.** During prescheduled contacts with the ground, the FSW must be able to send a high volume real-time and playback telemetry to the ground station. The FSW must also be able to receive and process telemetry from the ground station.
5. **Operate Payload.** The FSW must operate spacecraft's payload devices to achieve its mission. Additionally, during events of interest, the FSW must also enable the appropriate payload instruments to collect data at higher sampling rates.
6. **Manage Power.** Periodically, the FSW checks the spacecraft's power levels using input from the power distribution unit and makes the appropriate adjustments.
7. **Control Attitude.** Periodically, the FSW must check the spacecraft's attitude using inputs from the sun sensors, inertial measurement unit, and star tracker and make the appropriate adjustments to maintain the required attitude range.
8. **Maintain Temperature.** Periodically, the FSW must check the spacecraft's temperature and make the appropriate changes to keep the temperature within the required temperature range.
9. **Maintain Flight Plan.** Periodically, the FSW must check the spacecraft's flight path and make the appropriate adjustments to ensure the desired flight path is maintained.
10. **Maintain Orbit.** Periodically, the FSW must check the spacecraft's orbit and make the appropriate orbital adjustments to ensure the desired orbit is maintained.

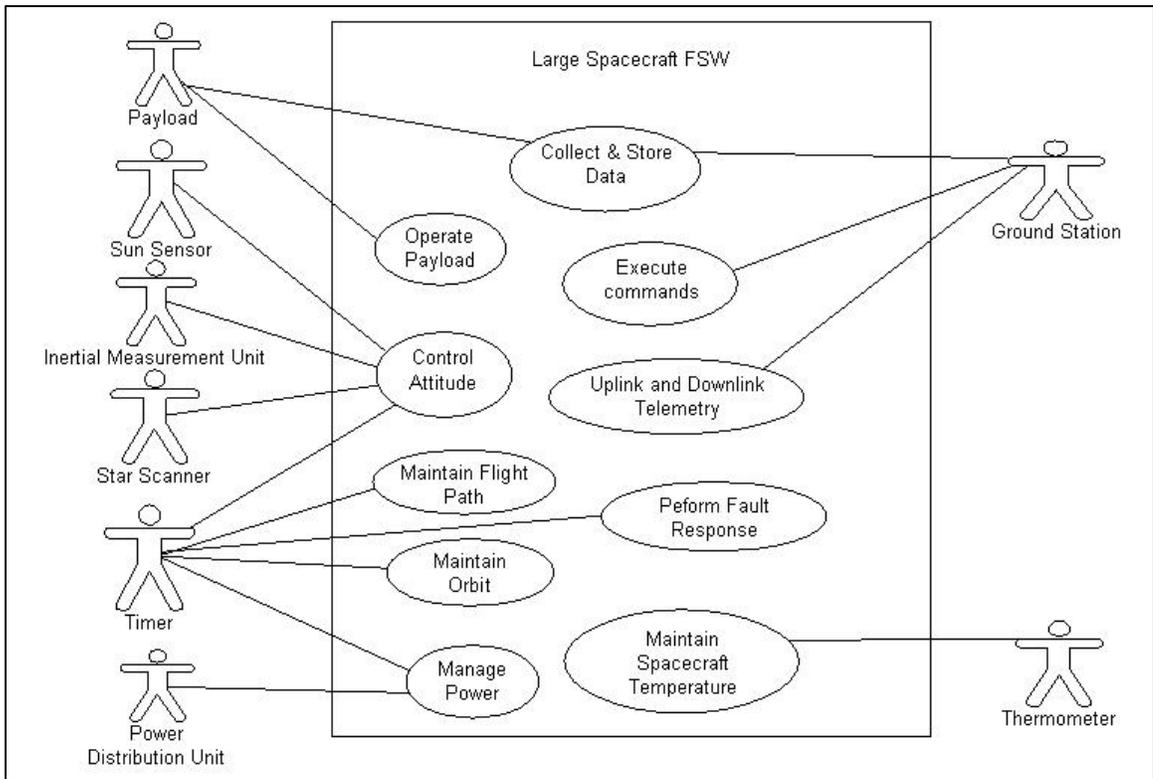


Figure B-2 Large Spacecraft System use cases

B.1.6 Time-Triggered Spacecraft System Use Case Model

The use case model for the Time-Triggered Spacecraft System is the same as the Large Spacecraft System since it has the same mission as the Large Spacecraft. However there are slight differences in the Execute Command use case description. On the Time-Triggered Spacecraft the spacecraft clock is provided by the hardware therefore the FSW is not required to maintain the spacecraft clock. Additionally, the FSW must have strict temporal predictability.

B.1.7 FSW SPL Use Case Model

This subsection describes the development of use case model. The development of the use case model follows the process described in the PLUS method. This process is demonstrated using the FSW SPL, which was developing following SPL reverse engineering approach. With the reverse engineering approach, first the uses cases are developed for the selected FSW systems. Then they are analyzed to identify the kernel and variable uses cases for the FSW SPL.

The FSW SPL use case model is depicted below in Figure B-3. First, analysis of the FSW SPL reveals that there are six kernel actors that are used by all SPL members, as seen in Figure B-3. However, in some cases there are different variants of the actors. For example, the Large Spacecraft uses star tracker, sun sensors, and an inertial measurement unit to provide attitude measurements, while the Small Spacecraft uses horizon crossing indicators and a magnetometer. To capture these differences the actor is modeled as Attitude Measurement Device, which can have several variants. There is also one optional actor called Temperature Sensor, which is only provided by some SPL members.

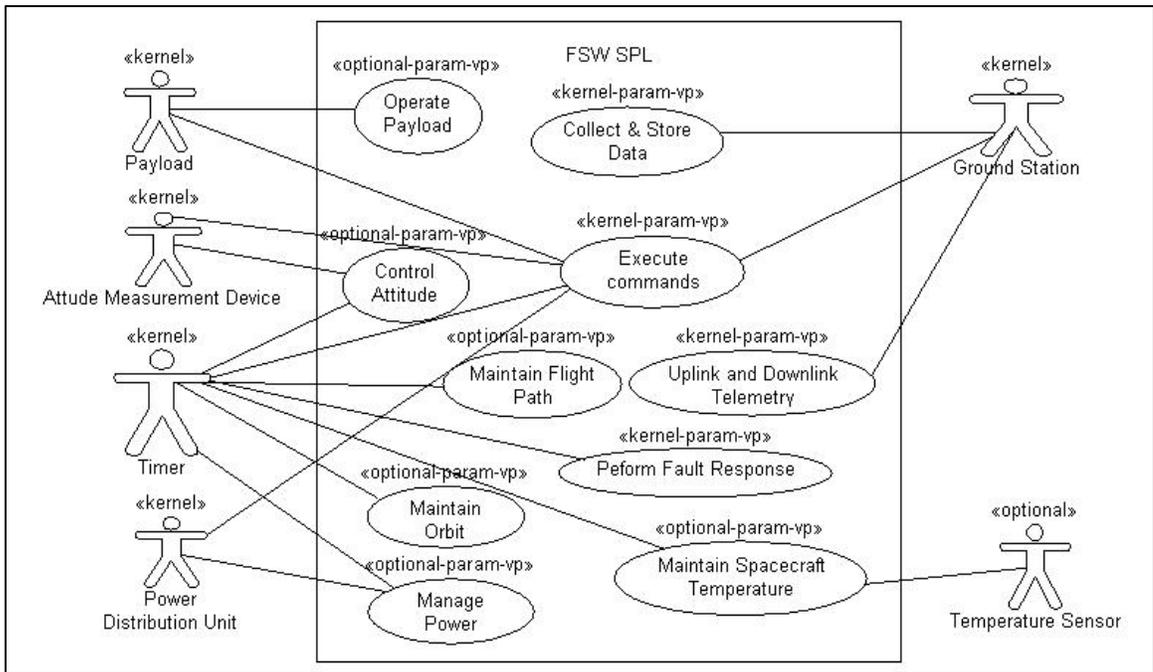


Figure B-3 FSW SPL use case model

Next, analysis of the FSW SPL reveals that there are four kernel use cases that are used by all SPL members, as seen in Figure B-3. However, there are slight differences in the use cases. Therefore variation points are used to capture these differences. The value of the variation point is set during the application engineering process and remains unchanged for a given SPL member. The variation points for the kernel use cases are described below.

1. Collect & Store Spacecraft Data.

- a. **Collection Trigger.** This variation point captures the variability in the triggers that causes software to collect data. For example, on the Small Spacecraft all data collection is driven by ground commands. Alternatively, on the Large Spacecraft the collection of payload data and housekeeping data is event driven by predetermined time intervals.
- b. **Algorithm Flexibility.** This variation point captures the required flexibility of the data formatting algorithms. For instance due to the Large Spacecraft's long lifetime, it needs the ability to update the formatting algorithms if the telemetry standard changes. Conversely, the Small Spacecraft has a short lifespan and changes to the telemetry format are not anticipated. Therefore it does not require algorithm flexibility.
- c. **Data Volume.** The Data Volume variation point captures variability in the amount of data collected, formatted, and stored also varies among the SPL members.
- d. **Attitude Control Devices.** This variation point captures variability in the type of attitude control devices used on the spacecraft. For example, the Small Spacecraft uses a torque rod to control its attitude, while the Large Spacecraft utilizes a reaction control system (RCS). This variation point influences the type and frequency of data collected from these devices.
- e. **Attitude Determination Devices.** This variation point captures variability in the type of attitude determination devices used on the spacecraft. For

example, the Small Spacecraft uses a horizon crossing indicator and magnetometer to measure attitude, while the Large Spacecraft utilizes a inertial reference unit and different celestial reference devices. This variation point influences the type and frequency of data collected from these devices.

- f. Memory Storage Devices.** This variation point captures variability in the type of memory storage devices used on the spacecraft. For example, some spacecraft use tape recorders, while others utilize random access memory (RAM) devices. These variation points influence the type and frequency of data collected from these devices.
- g. Antenna Type and Antenna Number.** The type and number of antenna a spacecraft needs varies between spacecraft. For instance the Small Spacecraft System only requires one low gain antenna, while the Large Spacecraft System needs a low gain antenna, a medium gain antenna, and a movable high gain antenna. This variation point influences the type and frequency of data collected from these devices.
- h. Power Devices and Power Appendage.** Captures the variability in the type of power devices a spacecraft uses to generate power. These variation points influence the type and frequency of data collected from these devices.
- i. Temperature Control Devices and Temperature Sensors.** This variation point captures whether or not the spacecraft has temperature measurement devices and temperature sensor devices. For example the Small Spacecraft

does not, while the Large Spacecraft uses thermistors and heaters. This variation point influences whether or not data from this device needs to be collected.

j. Propulsion Thrusters. This variation point captures whether or not the spacecraft has propulsion thrusters. For example the Small Spacecraft does not require thruster, while the Large Spacecraft requires thrusters to execute its flight path. This variation point influences whether or not data from this device needs to be collected.

k. Payload Devices. This variation point captures the variability in the payload devices used on spacecraft. The payload devices are unique to each spacecraft and thus the variants cannot be defined at the SPL level. This variation point influences the type and frequency of data collected from these devices.

2. Execute Commands.

a. Command Volume. Captures the variability in the amount of commands the software needs to process. For instance, the Small Spacecraft System only needs to process a small number of commands while the Large Spacecraft System must execute a significant number of commands.

b. Command Flexibility. This variation point is added because only some SPL members need the ability to support new commands. For example, the Large Spacecraft has a complex mission and the way in which payload devices are used to perform the mission may change, thus new commands need to be

supported. Conversely, the in Small Spacecraft the mission is very simple and the lifespan is very short so changes to the command structure are not anticipated.

- c. **Strict Temporal Predictability.** Captures whether not a spacecraft is required to have strict temporal predictability. For example, the Small Spacecraft does not, while the Time-Triggered Spacecraft does.
- d. **Modes.** This variation points captures the variability in the spacecraft's modes. For example, the Small Spacecraft has launch, normal, and safe modes, while the Large Spacecraft has Launch, Normal, Maneuver, Safe Mode and Standby.
- e. **Spacecraft Clock.** This variation point captures whether or not the FSW is required to maintain the spacecraft clock or if it is provided by hardware. For example, the Time-Trigger Spacecraft has global time provided by the time triggered architecture, while the Small Spacecraft's FSW must maintain the spacecraft clock.
- f. **Spacecraft IO Devices.** This use case is influenced by all the Spacecraft IO device variation points because it influences how the FSW performs its commanding. Therefore all of the spacecraft IO device variation points, which were previously described in the Collect and Store Spacecraft Data use case, are added to this use case.

3. **Perform Fault Response.**

- a. **Fault Response Capability.** This use case has significant variability because the amount of fault response capability included in the spacecraft varies significantly. For example, the Small Spacecraft System is only required to respond to critical faults such as a failure with the antenna. All non-critical faults detected are downlinked to the ground station for resolution. However, the Large Spacecraft must be able to handle the critical faults as well as non-critical faults due to its limited real-time commanding from the ground station.
- b. **Spacecraft IO Devices.** This use case is influenced by all the Spacecraft IO device variation points because it influences how the FSW performs its fault response. Therefore all of the spacecraft IO device variation points, which were previously described in the Collect and Store Spacecraft Data use case, are added to this use case.

4. Uplink/Downlink Telemetry.

- a. **Antenna Type and Antenna Number.** Captures the type and number of antenna a spacecraft needs. For instance the Small Spacecraft System only requires one low gain antenna, while the Large Spacecraft System needs a low gain antenna, a medium gain antenna, and a movable high gain antenna. This variation point influences the type and frequency of data collected from these devices.
- b. **Data Volume.** This variation point captures the amount of data that is transmitted and received. For example, the Small Spacecraft only needs to

transmit a small amount, while the Large Spacecraft needs to transmit a high volume of data.

After the kernel use cases are identified, the remaining use cases that are only realized by some SPL members become optional and alternative use cases. In the FSW SPL there are six optional use cases all with variation points.

1. **Operate Payload.**

- a. **Payload Devices.** This variation point is used to capture the unique payload devices used on the spacecraft and how the FSW operates them.

2. **Control Attitude.**

- a. **Stabilization Technique.** This variation point captures the attitude stabilization technique. For instance the Small Spacecraft is spin stabilized, while the Large Spacecraft is three-axis stabilized. Each stabilization technique can use different hardware to accomplish its purpose.

- b. **Attitude Control Devices.** This variation point captures variability in the type of attitude control devices used on the spacecraft. It influences how the FSW interacts with the attitude control hardware.

- c. **Attitude Determination Devices.** This variation point captures the different hardware devices used to measure the attitude. It influences how the FSW interacts with the attitude determination hardware.

- d. **Attitude Algorithms.** This variation point captures the different algorithms used to compute attitude and make adjustments.
3. **Control Power.**
- a. **Power Devices and Power Appendage.** This variation point captures the difference in the power devices used on the spacecraft. It influences how the FSW performs is power management.
4. **Maintain Temperature.**
- a. **Temperature Measurement Devices and Temperature Sensors.** This variation point captures the variability in the type of temperature measurement devices and temperature sensor devices. This variation point influences how the FSW performs its thermal control.
5. **Maintain Flight Path.**
- a. **Flight Path Algorithms.** This variation point captures the different algorithms that different spacecraft use to monitor, track, and control their flight path.
6. **Maintain Orbit**
- a. **Orbital Adjustment Algorithms.** This captures the variability in the algorithms that are used to compute and adjust a spacecraft's orbit.
 - b. **Attitude Determination Devices.** This variation point captures the different hardware devices used to measure the attitude and orbit. It influences the orbital adjustment algorithm.

B.2 Collect and Store Spacecraft Data Use Case Activity Diagram Model

The next use case scenario is the Collect and Store Spacecraft Data use case, which involves collecting, formatting and storing spacecraft payload and housekeeping data. The activity for this use case is depicted in Figure B-4. It has three main steps which are Collect Data, Format Telemetry, and Store Data. First the data is collected during the Collect Data adaptable input step, which is influenced by the Collection Trigger variation point. Then there is an execution condition decision point, where if the data collected is already formatted into telemetry packets the flow continues onto the Store Data adaptable output step. If the data collected is not formatted, then flow moves to the Format Telemetry adaptable internal step before going to the Store Data adaptable output step.

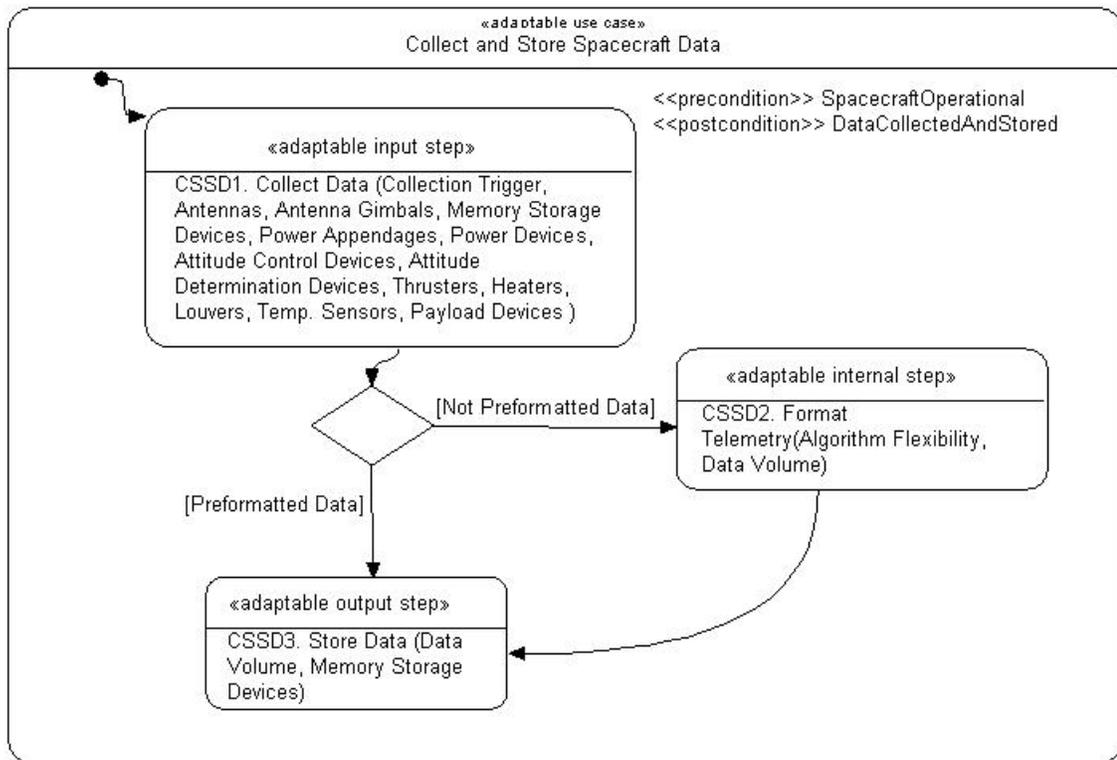


Figure B-4 Collect and Store Spacecraft Data use case activity diagram

Since each of the adaption steps can be further refined without combinatorial explosion, they are refined in sub-activity diagrams. First the Collect Data adaptable input step is further elaborated in Figure B-5. This step begins with an execution condition decision point, which captures the type of data that is collected. If payload data needs to be collected, the flow then follows one path. If the type of data that needs to be collected is housekeeping data (i.e. not payload data), then the flow of activities continues along another path. Next, another decision point is encountered along both paths. However,

this decision point is a feature condition, where the SPL feature impacts the flow of execution. After the data is collected based on the feature condition, all paths merge back together then exit the step.

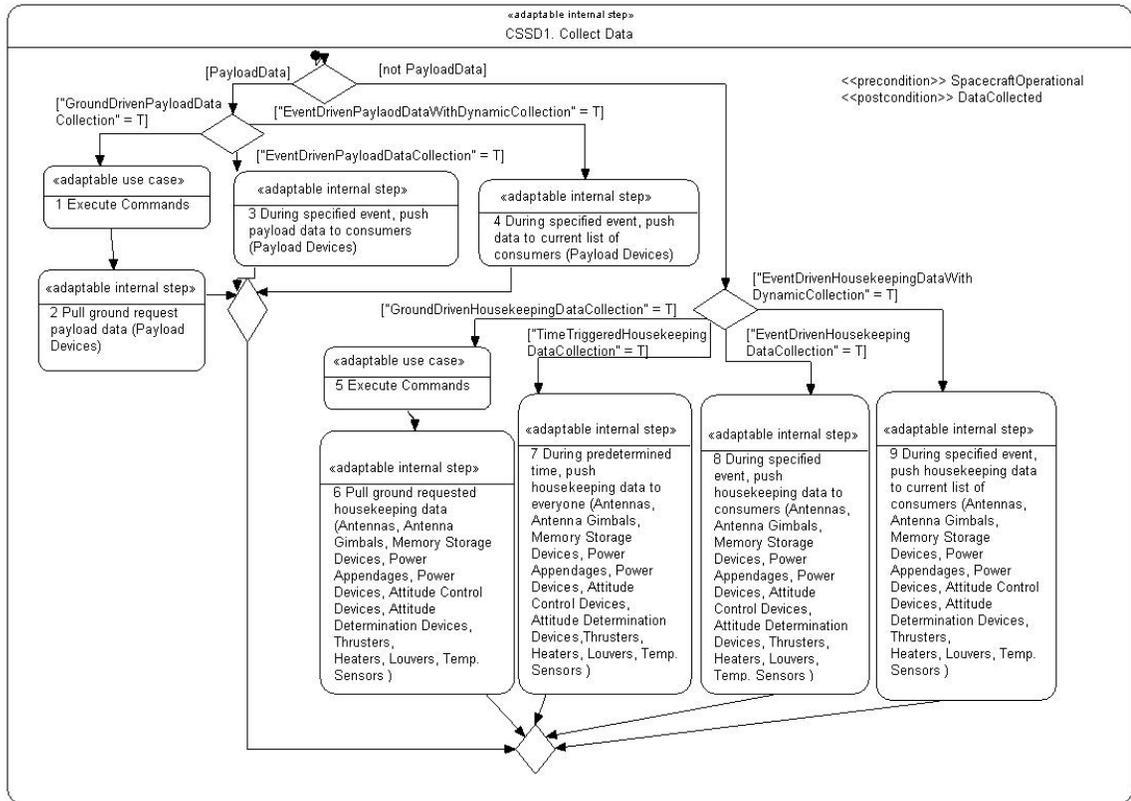


Figure B-5 Sub-activity diagram for Collect Data step

Each step in the Collect Data sub-activity diagram is adaptable, therefore to fully specify the use case scenario sub-activity diagrams should also be created for these steps.

However, steps 2-4 depend on the Payload Devices variation point whose specific variants are unknown and therefore cannot be elaborated. Steps 6-9 depend on a multitude of variation points, thus they are also not elaborated in sub-activity diagrams because it would create combinatorial explosion. Instead the all the potential individual interactions will remain abstracted at this level.

After the data is collected, the next step in the use case scenario is an execution decision point, as seen in Figure B-4. If the data collected is not already formatted into telemetry packets, then it must be formatted in the Format Telemetry internal adaptable step. The sub-activity diagram for this step is shown in Figure B-6. This step has several alternative feature-based conditions. First, if TelemetryFormationReliability feature is set to SignificantCheck then one path is taken. Along this path there is another decision point that corresponds to the TelemetryFormation feature. During any one of these paths, if a problem is detected, then the flow continues into the Perform Fault Response use case activity diagram. Otherwise if everything goes as expected, then the scenario ends.

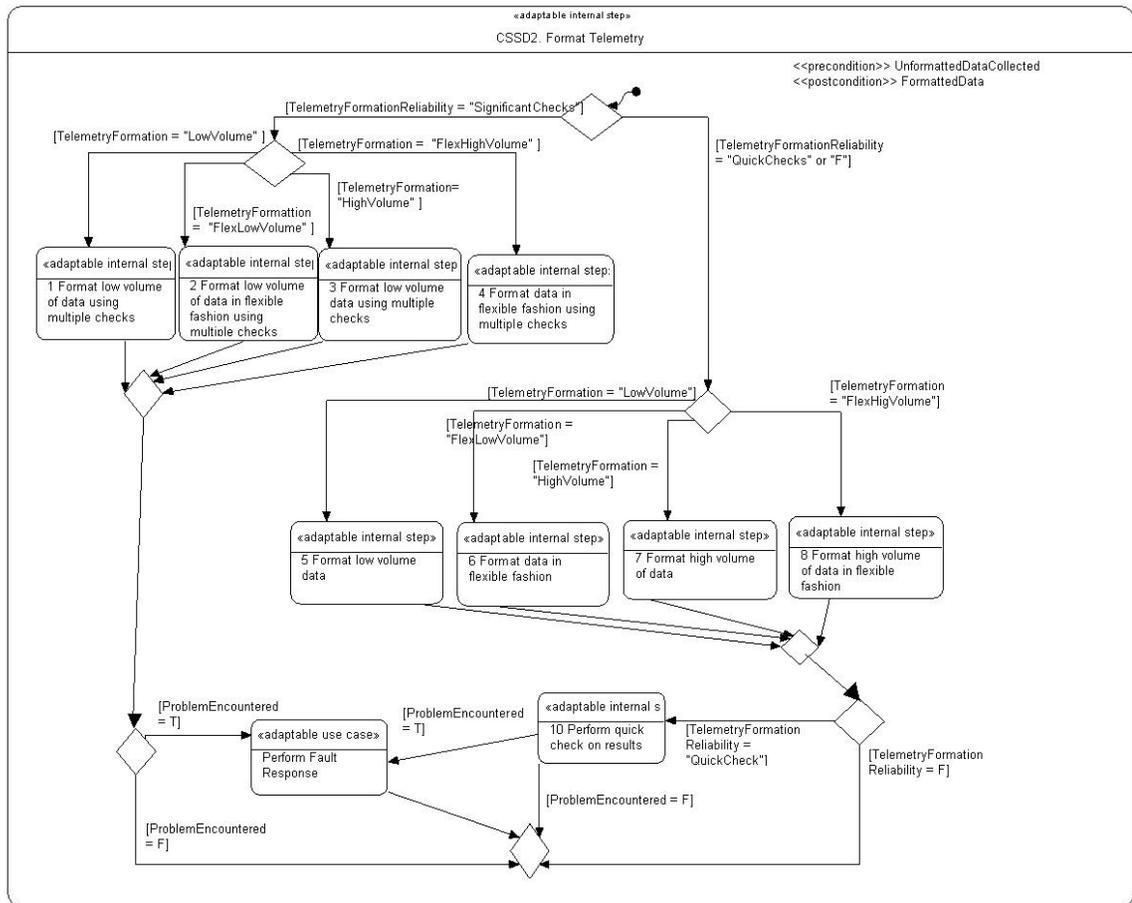


Figure B-6 Sub-activity diagram for Format Telemetry step

The other main path in the Format Telemetry step is taken when the TelemetryFormationReliability feature is either QuickCheck or False. Along this path there is another decision point that corresponds to the TelemetryFormation feature. After the telemetry is formatted the paths merge and a feature-based decision point is reached. If the TelemetryFormationReliability feature is false then the scenario ends. If

the TelemetryFormationReliability feature is QuickCheck then a simple check is performed on the results. If a problem is detected, then the flow continues into the Perform Fault Response use case activity diagram. Otherwise if everything goes as expected, then the scenario ends. Finally, the last step in the Collect and Store use case scenario is to store the data. The sub-activity diagram for this adaptable step is depicted in Figure B-7. This step has several alternative paths based on the TelemetryStorageandRetrieval and MemoryStorageDevice features. At the end of this step the telemetry data is stored in the spacecraft's onboard memory.

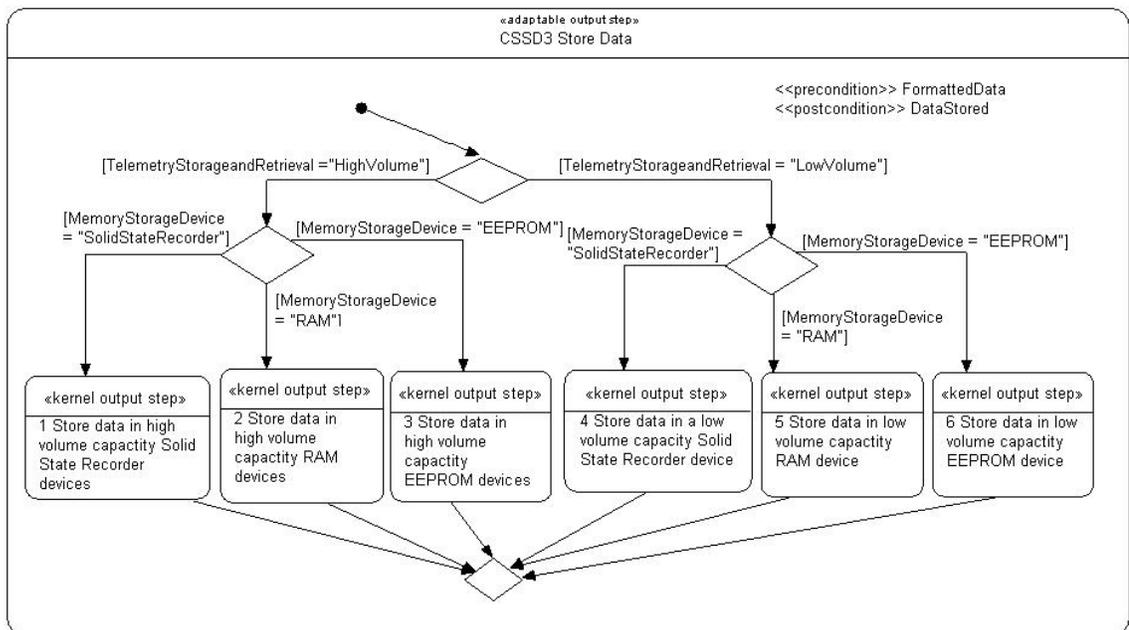


Figure B-7 Sub-activity for Store Data step

B.3 Unmanned Space Flight Software Product Line Conceptual Static Model

The first analysis modeling artifact produced is the conceptual static model. This artifact places emphasis on identifying the physical devices that interface with the SPL and is developed following the PLUS method (Gomaa 2005). Variability among the physical devices is captured using SPL stereotypes. The conceptual static model is derived based on FSW SPL features. The physical devices that all FSW SPL members must also interface with are marked with the <<kernel>> stereotype. The physical devices that only some of the FSW SPL members must interface with are marked with the <<optional>> stereotype. In some cases, there are variant physical devices used that perform similar functions. These physical devices are denoted using the <<variant>> stereotype and modeled as subclasses to the base class that captures the commonalities of the variants (Gomaa 2005).

The process for developing a conceptual static model is illustrated using the FSW SPL. Based on the FSW SPL's Command Execution pattern specific feature group and Housekeeping Data Collection pattern specific feature group there are several kernel devices, which include antennas, transmitters, receivers, payload devices, memory storage devices, attitude control devices, attitude determination devices, power distribution units (PDUs), and power devices. The FSW SPL conceptual static model capturing these physical devices is depicted below in Figure B-8.

Within some of these kernel devices there are variant physical devices that can be used. First, there are several variants for the antenna and attitude control devices. These variants are derived from the optional features in the Antenna and Attitude Control Device pattern variability feature groups, respectively. Second, there are also several options for memory storage devices, power devices, and attitude control devices. These variants are derived from the alternative features in the Memory Storage Device and Power Device, Attitude Control Device pattern variability feature groups, respectively. Since these sets of variants are derived from alternative features, they are marked as mutually exclusive to indicate that only one variant can be selected from each group.

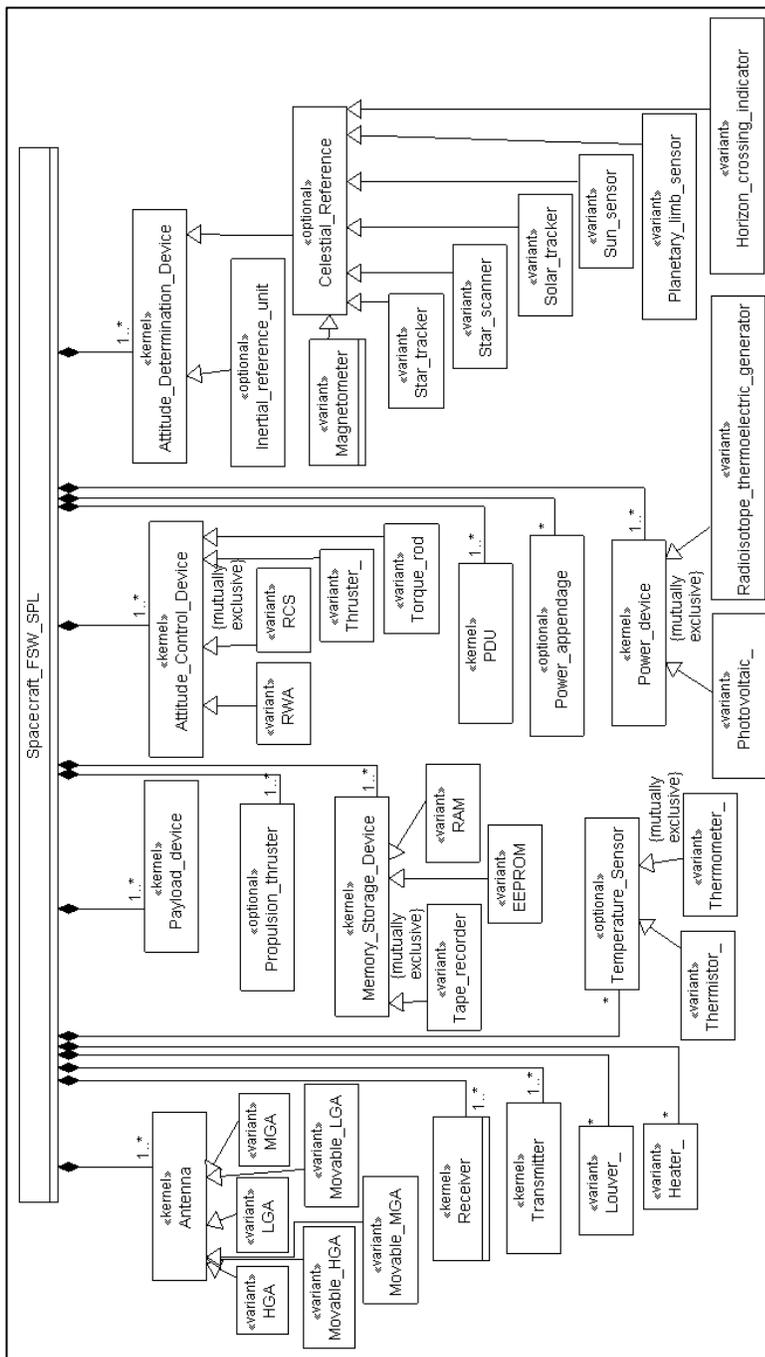


Figure B-8 FSW SPL conceptual static model

In addition to kernel physical devices, the FSW SPL also has optional physical devices that are only used by some SPL members. As identified in the Antenna Gimbal, Temperature Control Device, Temperature Sensor, Power appendage, and Propulsion Thruster pattern variability features/feature groups, these optional devices include antenna gimbals, temperature control devices, temperature sensors, power appendages, and propulsion thrusters. Within some of these kernel devices there are variant physical devices that can be used. First, there are variant types of temperature control devices, which are derived from the optional features in the Temperature Control pattern variability feature group. Second, there are also variant types of temperature sensors. These are derived from the alternative features in the Temperature Sensor pattern variability feature group. Since these variants are derived from alternative features, they are marked as mutually exclusive to indicate that only one variant can be selected.

B.4 Unmanned Space Flight Software Product Line Context Model

Another major artifact produced from the FSW SPL analysis modeling phase is the SPL context diagram. The purpose of the context diagram is to identify the SPL boundary to the external environment. Additionally, it also captures at a very high level, how the SPL interfaces with the physical devices including the device variability. It is developed following the process described in the PLUS method (Gomaa 2005).

The SPL context modeling process is illustrated using the FSW SPL. Figure B-9 shows the context diagram for the FSW SPL. The external devices and their variability are

derived from the FSW SPL feature model and FSW SPL conceptual static model in Figure B-8.

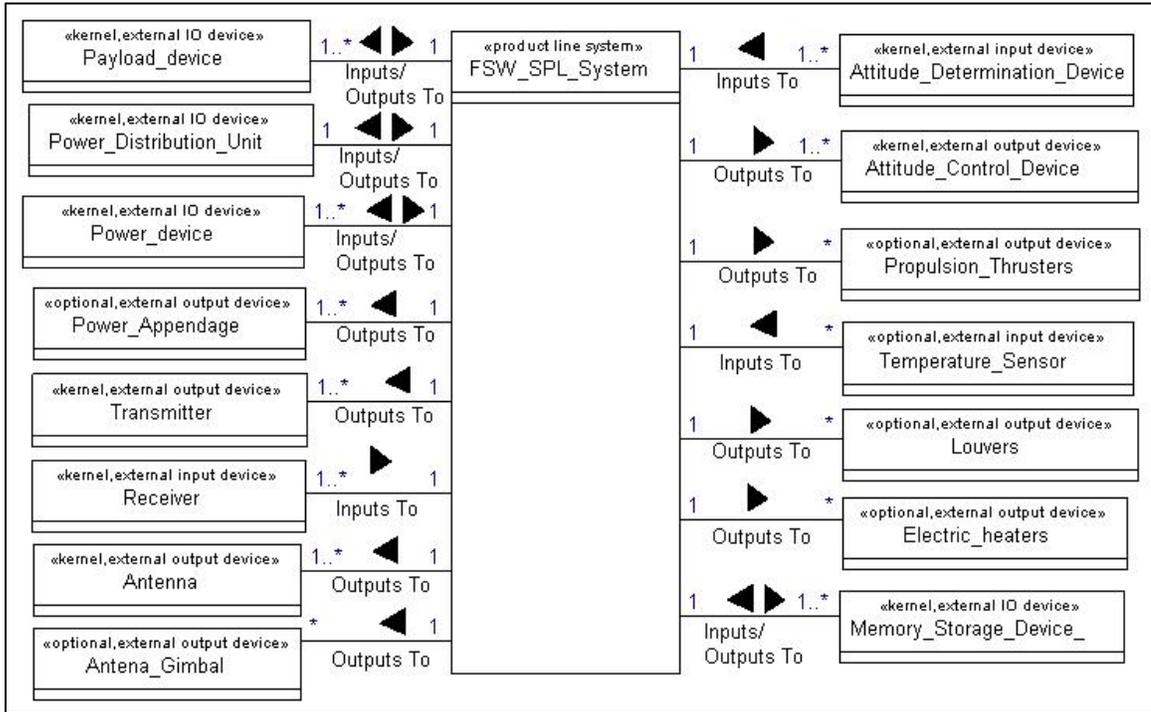


Figure B-9 FSW SPL context diagram

B.5 Unmanned Space Flight Software Product Line Subsystem Structuring

The next key artifact from the analysis modeling phase is the subsystem structuring diagram. The purpose of the subsystem structuring diagram is to identify the SPL subsystems and their variability. It is developed following the process described in the PLUS method (Gomaa 2005).

The SPL context modeling process is illustrated using the FSW SPL. Figure B-10 depicts the subsystem structuring for the FSW SPL. Based on the FSW SPL feature model, the FSW SPL can be divided into five kernel subsystems with parameterized variation points that are required by all SPL members. These subsystems include the command and data handling (C&DH), payload, telemetry tracking and control, power, fault management and attitude control subsystem. Additionally, there are three optional subsystems with parameterized variation points that are provided by some SPL members. These subsystems include propulsion, guidance and control, and thermal subsystems.

Figure B-10 also depicts the subsystem dependencies. The command and data handling subsystem relies on all the other subsystems because ground commands it is responsible for must be executed with the appropriate subsystem's devices and it also relies on the other subsystems for data. The fault management subsystem relies on all the other subsystems to perform fault detection and report faults to the fault management subsystem.

Figure B-10 also shows that the Guidance and Control subsystem relies on the Propulsion subsystem. This is because the Guidance and Control subsystem utilizes the Propulsion subsystem to execute specific maneuvers to execute the flight path.

Finally, there is also a dependency between the Payload subsystem and the Attitude Control subsystem. This dependency is drawn because certain payload devices require the spacecraft to be accurately positioned before they can collect data.

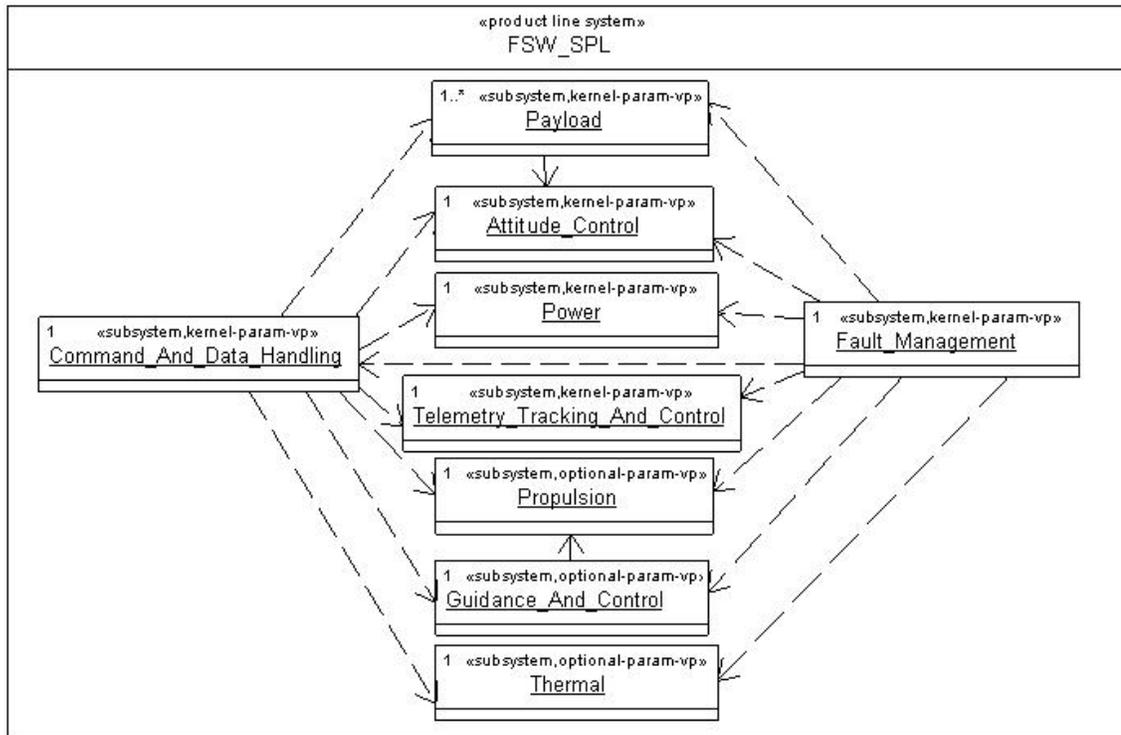


Figure B-10 FSW SPL subsystem structuring and dependencies

Another view of the subsystem structuring is a refinement of the SPL context diagram. This diagram is depicted in Figure B-10. This diagram shows the subsystems along with the allocation to the physical devices. It can be seen from Figure B-10 that the devices

are grouped functionally with the subsystems. For example, all the devices associated with attitude control and attitude determination are associated to the attitude control subsystem.

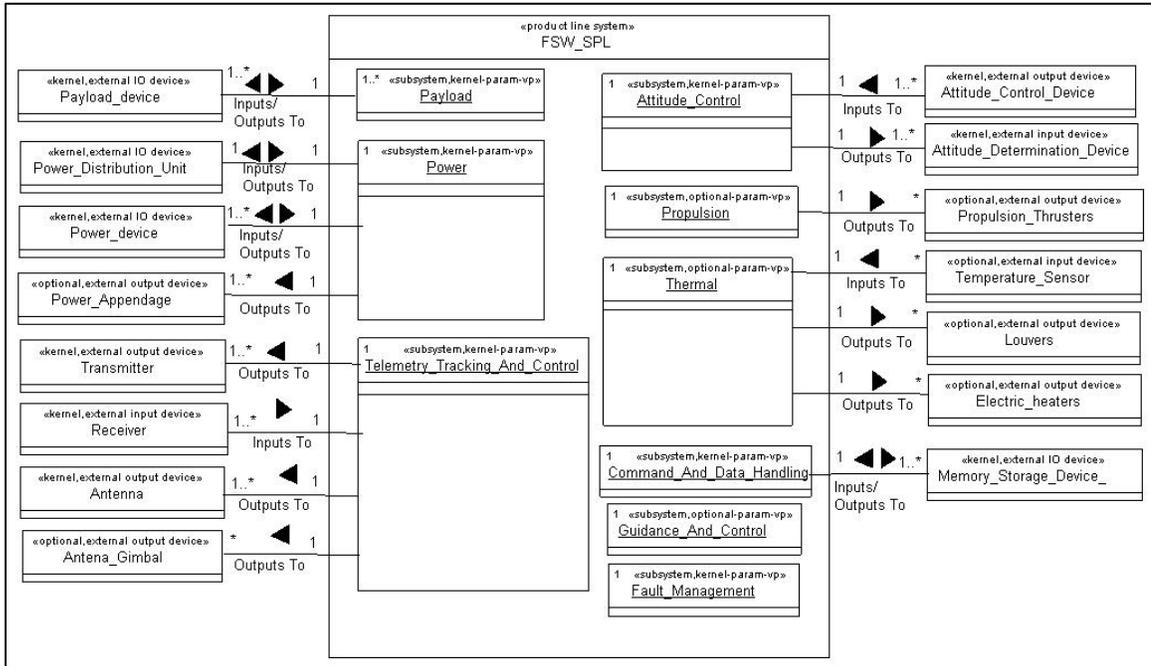


Figure B-11 Subsystem allocation to physical devices

B.6 Unmanned Spacecraft Flight Software Product Line Feature to Design

Pattern Mapping

B.6.1 Dynamic Model for Low Volume Command Execution with Flexible

Commands

First, the objects that participate in the Low Volume Command Execution with Flexible Commands pattern specific feature are modeled and examined. This pattern specific feature is derived from the Execute Commands use case where the Command Volume variation point is low, Command Flexibility variation point is true, and Strict Temporal Predictability variation point is false. This feature is slightly different from Low Volume Command Execution pattern specific feature because it must process commands in a manner that enables the ability to support new commands using the existing input, output, and IO component interfaces.

Figure B-12 shows a communication diagram for Low Volume Command Execution with Flexible Commands pattern specific feature. This set of interactions is similar to the Low Volume Command Execution because due to the small amount of commands that need processing, one controller receives and processes a set of ground commands from the Telemetry Tracking and Control Subsystem. Additionally, it also has the same additional components derived from the pattern variability features. This set of interactions differs because instead of directly calling a particular input, output or IO component, the controller creates a command objects and sets the parameterized values. The command objects are then sent to the Invoker to be stored before they are ready to execute. Once they are ready to execute, the Invoker sends them to the appropriate `Command_Dispatcher`.

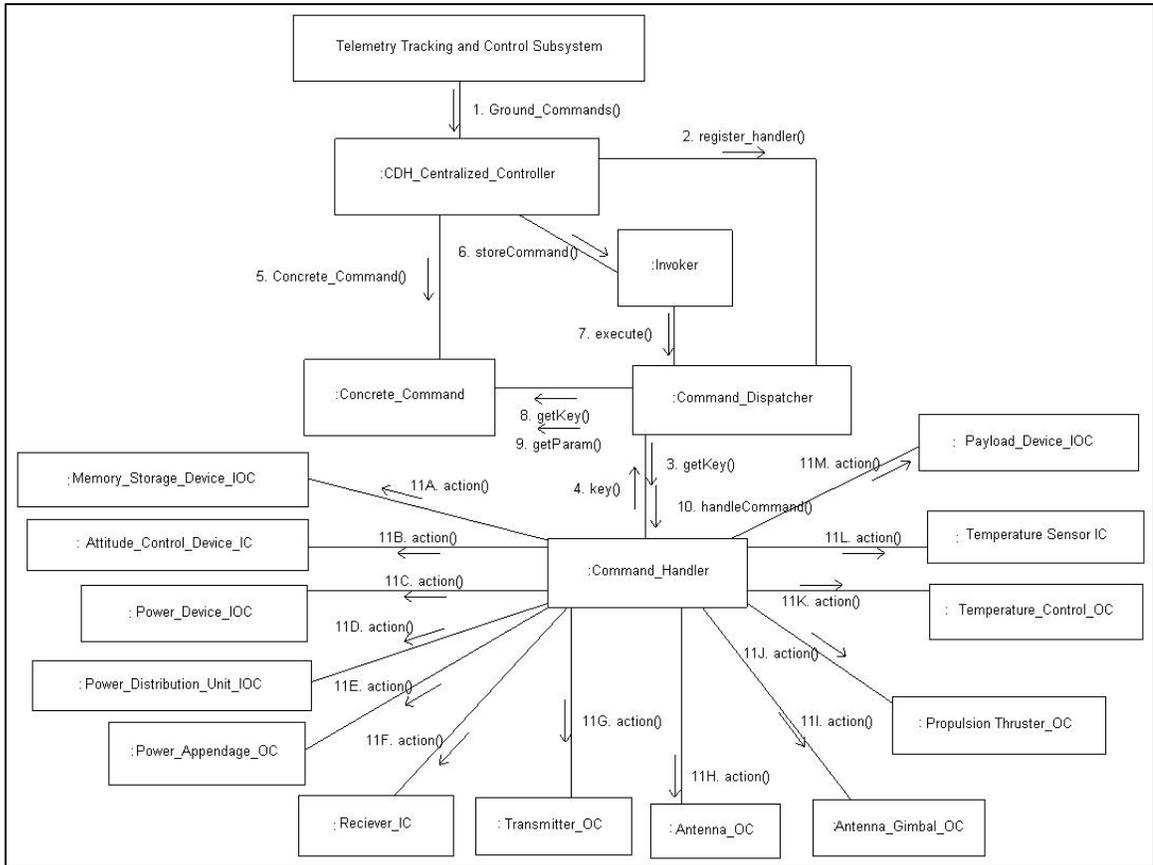


Figure B-12 Low Volume Command Execution with Flexible Commands communication diagram

The Command_Dispatcher reads the receiver and the parameterized action from the command and sends the information to the appropriate Command_Handler to execute. The Command_Handler then invokes the appropriate actions on the input, output, and IO components. The additional set of interactions moves the control logic into the Command_Handler and the Command_Dispatcher dynamically manages the Command_Handler through a registration process. Therefore, if a new command is

required, a new command object and updated Command_Handler are created and can be registered during run-time. This provides the ability to quickly add new commands during run-time.

Finally, the interactions involved in the communication diagram are analyzed and applicable design patterns are identified. The interactions in this feature are consistent with Centralized Control design pattern (Gomaa 2011) and the Command Dispatcher design pattern (Dupire & Fernandez 2001). Thus the Low Volume Command Execution with Flexible Commands pattern specific feature is mapped to the pre-integrated combination of the Centralized Control and Command Dispatcher design patterns.

B.6.2 Dynamic Model for High Volume Command Execution with Flexible Commands

Next the dynamic model for the High Volume Command Execution with Flexible Commands pattern specific feature is built and examined. This feature is derived from the Execute Commands use case where Command Volume is high, Command Flexibility is true, and the Strict Temporal Predictability is false. This feature is similar to the High Volume Command Execution pattern specific feature because it contains all the same components derived from the feature and pattern variability feature. However, it differs because it must process commands in a manner that enables the ability to support new commands using the existing input, output, and IO component interfaces. To address this additional requirement, the same strategy that was used in the Low Volume Command

Execution with Flexible Commands in section B.6.1 can be applied. However, multiple localized Command Handlers will be used rather than just one.

Finally, the interactions involved in this feature are analyzed and applicable design patterns are identified. The interactions in this feature are consistent with Hierarchical Control design pattern (Gomaa 2011) and the Command Dispatcher design pattern (Dupire & Fernandez 2001). Thus the High Volume Command Execution with Flexible Commands pattern specific feature is mapped to the pre-integrated combination of the Hierarchical Control and Command Dispatcher design patterns.

B.6.3 Dynamic Model for Time-Triggered Volume Command Execution with Flexible Commands

Next the dynamic model for the Time Triggered Command Execution with Flexible Commands pattern specific feature is built and examined. This feature is derived from the Execute Commands use case where Command Volume is either high or low, Command Flexibility is true, and the Strict Temporal Predictability is true. This feature is similar to Time Triggered Command Execution pattern specific feature because contains all the same components derived from the feature and pattern variability feature. However, it differs because it must process commands in a manner the enables the ability to support new commands using the existing input, output, and IO component interfaces. Therefore the same strategy that was used in the Low Volume Command Execution with

Flexible Commands in section B.6.1 can be applied. However, multiple localized Command Handlers will be used rather than just one.

Finally, the interactions involved in this Time Triggered Command Execution with Flexible Commands pattern specific feature are analyzed and the design patterns are identified. The interactions in this feature are consistent with Distributed Control design pattern (Gomaa 2011) and the Command Dispatcher design pattern (Dupire & Fernandez 2001). Thus the Time Triggered Command Execution with Flexible Commands pattern specific feature is mapped to the pre-integrated combination of the Distributed Control and Command Dispatcher design patterns.

B.6.4 Dynamic Model for Spacecraft Clock

The Spacecraft Time pattern specific feature is derived from the Execute Commands use case. This feature involves keeping track of the spacecraft's time and distributing time to the required components. This use case has a variation point call Algorithm because the algorithm used can vary significantly. For example, "the spacecraft clock may be very simple, incrementing every second and bumping its value up by one, or it may be more complex, with several main and subordinate fields that can track and control activity at multiple granularities" (NASA Jet Propulsion Laboratory 2011). Therefore the specific internal algorithm cannot be set until the application engineering phase.

The interaction diagram for the Spacecraft Clock involves sending time updates to the required components. In this scenario, the Spacecraft Clock component maintains the spacecraft's time and it must send time updates to a set of components. In the FSW SPL, the set of components requiring time updates is fixed because the onboard configuration does not change after launch. This type of interaction is consistent with the Multicast design pattern, which sends data to a predetermined set of consumers. Thus the Multicast design pattern is mapped to the Spacecraft Clock pattern specific feature.

B.7 Unmanned Spacecraft Flight Software Product Line Architectural Design Pattern Modeling

B.7.1 FSW Distributed Control Architectural Design Pattern

The Distributed Control design pattern is mapped to the Time Triggered Command Execution feature. This process for updating the DRE Distributed Control executable design patterns is feature driven, where it is systematically updated to reflect the needs of the SPL pattern specific and pattern specific features. The updated collaboration and interaction architectural views are depicted below in Figure B-13 and Figure B-14. All the peer-to-peer messages among the distributed controllers are not modeled to keep the diagrams readable.

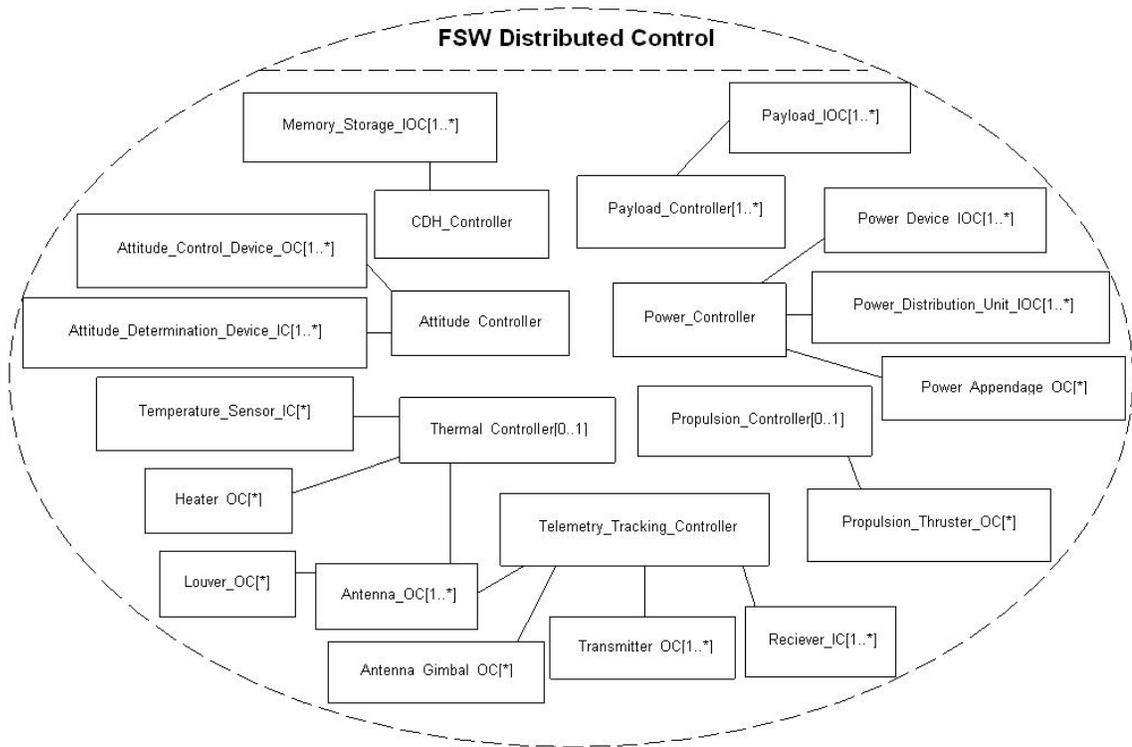


Figure B-13 FSW Distributed Control Collaboration Diagram

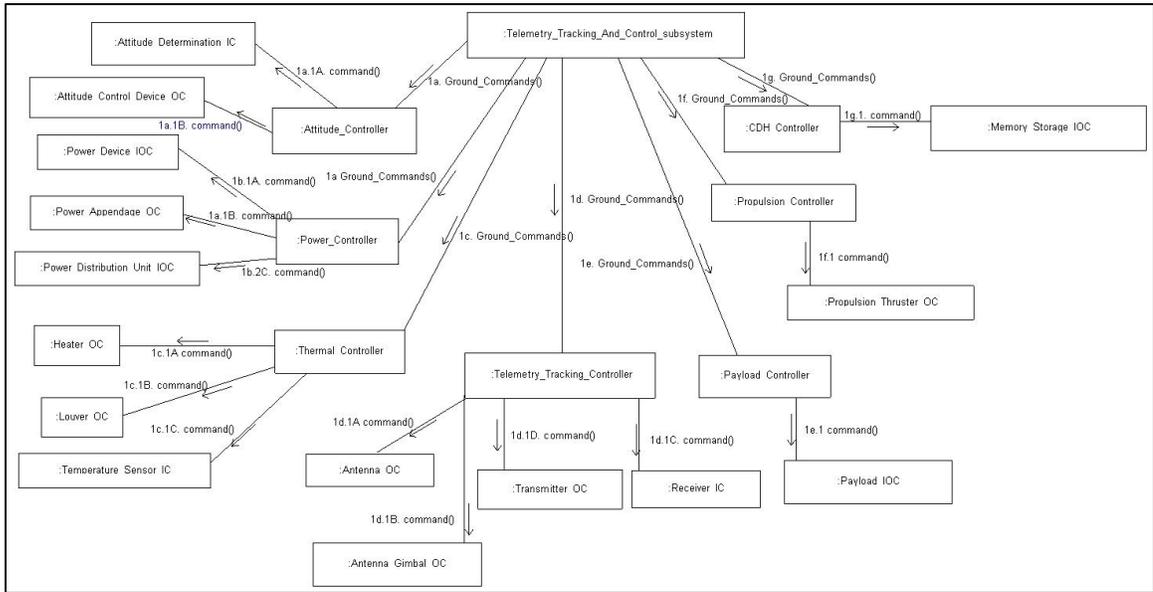


Figure B-14 FSW Distributed Control Interaction diagram

B.7.2 FSW Hierarchical Control Executable Design Pattern

The Hierarchical Control design pattern is mapped to the High Volume Command Execution feature. This process for updating the DRE Distributed Control executable design patterns is feature driven, where it is systematically updated to reflect the needs of the SPL pattern specific and pattern specific features. The updated collaboration and interaction architectural views are depicted below in Figure B-15 and Figure B-16.

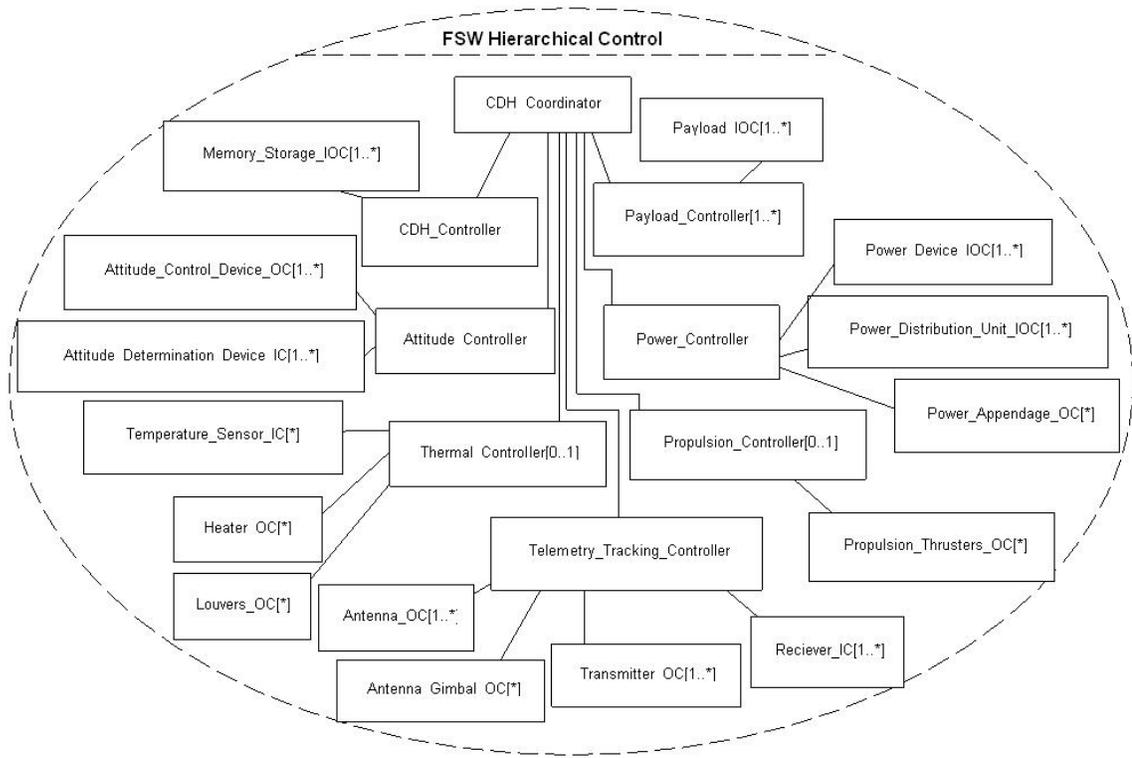


Figure B-15 FSW Hierarchical Control Collaboration Diagram

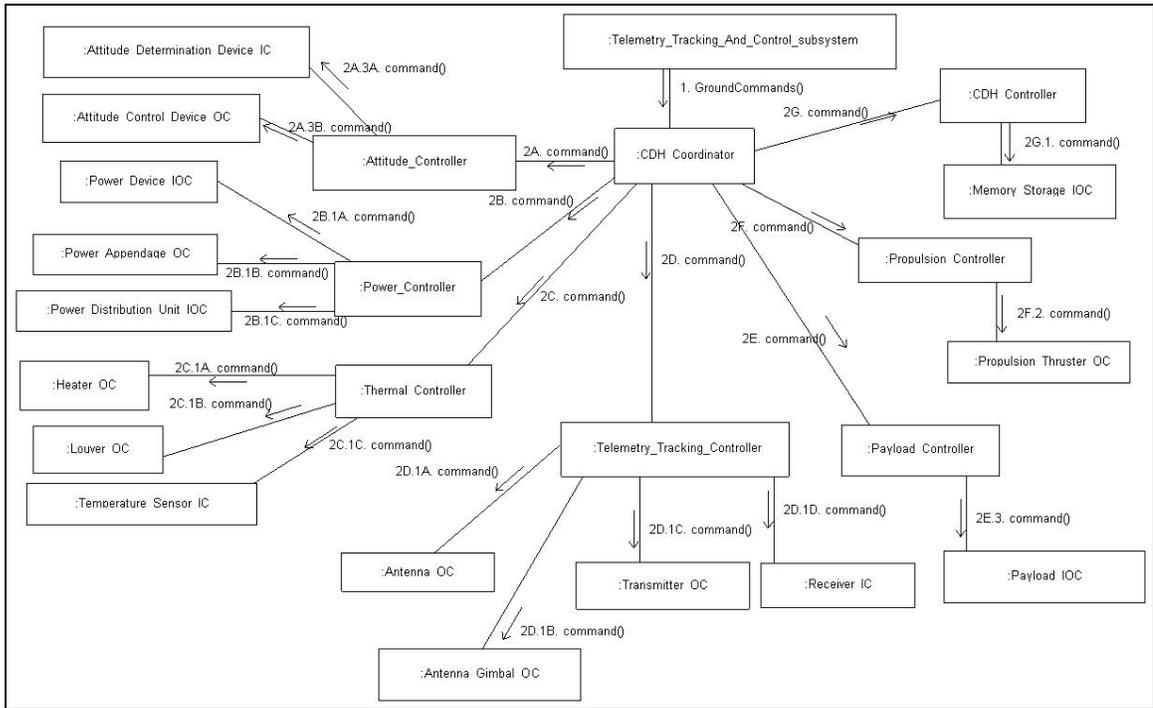


Figure B-16 FSW Hierarchical Control Interaction diagram

B.8 Collect and Store Spacecraft Data Design Pattern Interconnection

The collect and store space data use case scenario captures how the spacecraft will manage the collection and storage of data. This use case has three main steps which are Collect Data, Format Telemetry, and Store Data, as illustrated the use case's activity diagram in Figure B-4. Each of these steps is further elaborated using sub-activity diagrams. The interaction overview diagram derived from the activity diagram has a similar structure. Figure B-17 is the interaction overview diagram for the Collect and Store Use case. In the interaction overview diagram each of the steps is converted into a

reference to another interaction overview diagram, which further refines the step. Each of these interaction diagrams is described below in more detail.

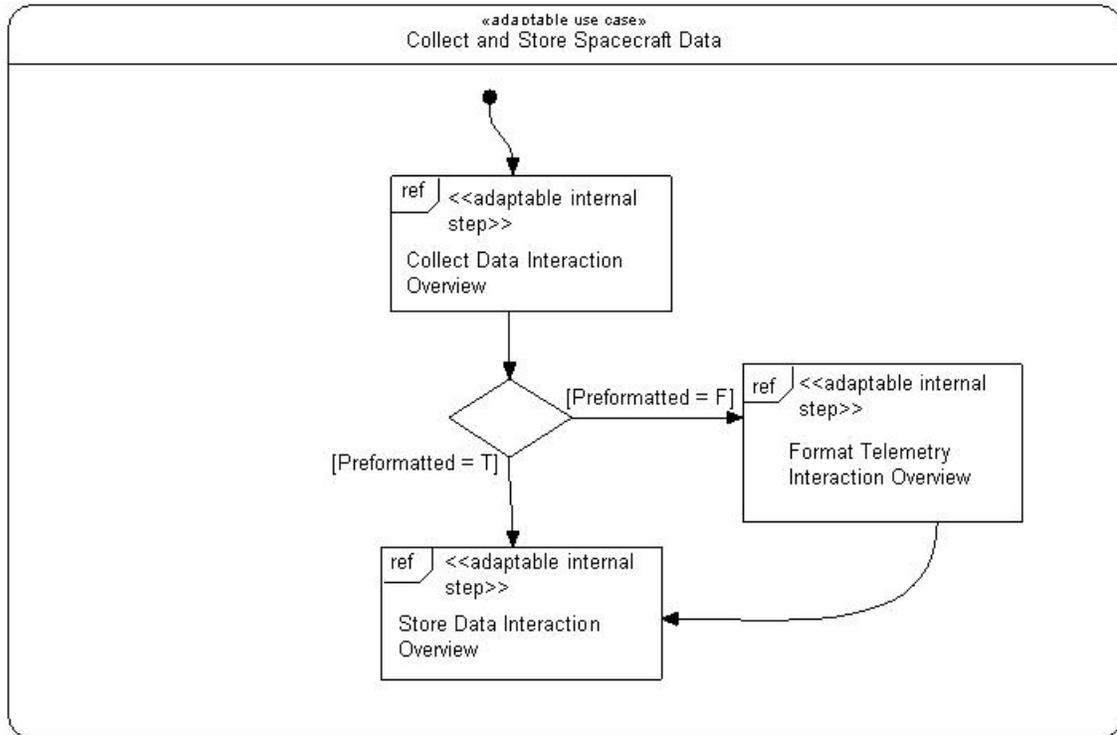


Figure B-17 Interaction overview diagram for Collect Housekeeping Data use case scenario

First, the collect data interaction overview diagram is created which captures the optional ways a spacecraft may collect data. Figure B-18 illustrates the design patterns that are involved in the various options, which is based on the Collect Data activity diagram in Figure B-5. All the activity steps in the Collect Data activity diagram are converted to

references to design pattern interaction diagrams that achieve that step or another interaction overview diagram. For example, if the collection of payload data is required to be ground driven, then route marked with the feature condition “GroundDrivenCollectPayloadData” = T path is taken through the interaction diagram. On this path, the first activity is to execute the ground command. This step is supported by the Execute Commands interaction overview diagram, which models the alternative ways in which commands are executed. Then a request is made to collect the housekeeping using the Collect Payload sequence diagram from the FSW Housekeeping Multiple Client Multiple Server design pattern. The design patterns in the Execute Commands interaction overview diagram interact with the FSW Housekeeping Multiple Client Multiple Server design pattern since they appear sequentially in the interaction overview diagram. Therefore these design patterns must be interconnected.

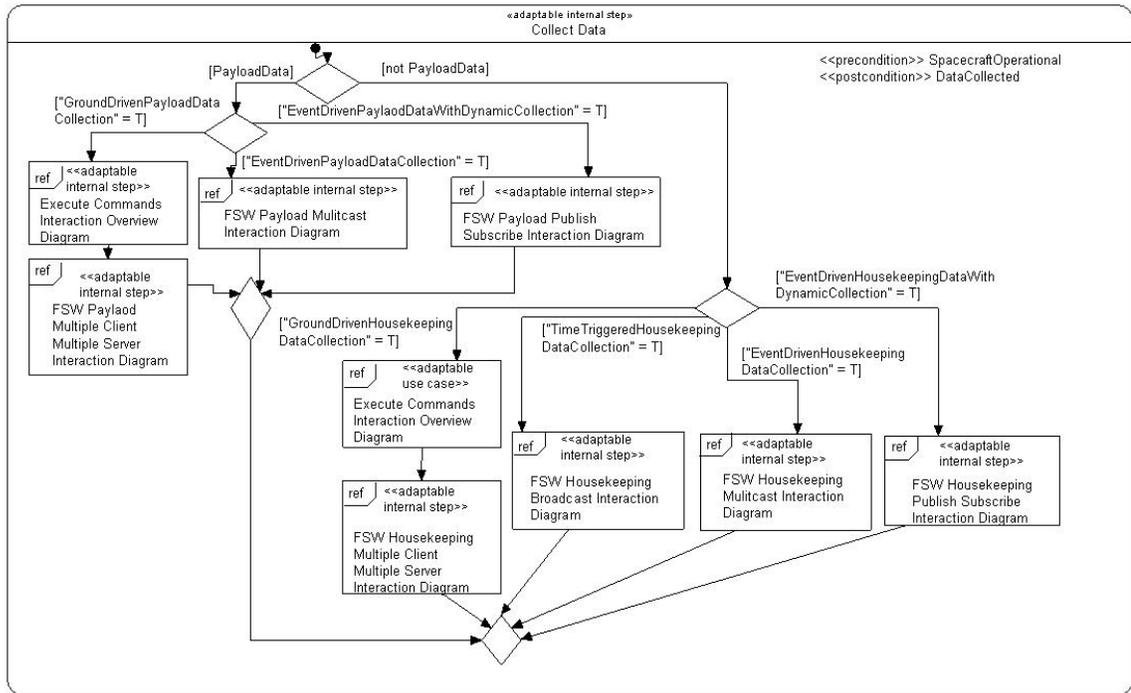


Figure B-18 Collect Data Interaction Overview

The interaction overview diagrams for the remaining two steps in this use case are also created following the same process.

B.9 Unmanned Spacecraft Flight Software Product Line Message Communication

The specific type of message communication used in the FSW SPL is as follows:

- Asynchronous Message Communication and Bidirectional Asynchronous Message Communication. The FSW SPL was primarily designed to level asynchronous communication for all one-way communication and bi-directional

asynchronous communication for all two-way communication. This decision was made because it provided the most flexibility in the system and efficient use of concurrent components.

- Synchronous Message Communication with Reply. The FSW SPL did use limited synchronous messages communication in the data processing functionality of the command and data handling system. The raw data is transformed into telemetry packets using a serial process where the producers must wait on the consumers to finish. Therefore synchronous messages communication was used in this instance.
- Publish/Subscribe Message Communication. The Publish/Subscribe message communication pattern was used in the FSW SPL to ensure the consistency with the Layered architecture. For example, some components in the Control layer need to be notified when a fault is detected by a component in the Fault Detection layer. Therefore to ensure the Control Layer is dependent on the Fault Detection layer the publish/subscribe communication is used. Where the components in the Control Layer subscribe to fault detection alerts from components in the Fault Detection Layer. Then when a fault is detected the component in the Control layer is notified.

APPENDIX C. SNOE CASE STUDY

C.1 SNOE Collect and Store Spacecraft Use Case Activity Modeling

The second use case that involves the command and data handling subsystem is the Collect and Store Spacecraft Data use case. It has three main adaptable steps which are Collect Data, Format Telemetry, and Store Data. Each of these steps has its own sub-activity. First, the Collect Data adaptable step for SNOE is depicted in Figure C-1. This sub-activity diagram has two feature based conditions corresponding the Payload Data Collection feature group and Housekeeping Data Collection feature group. SNOE utilizes the Ground Driven Payload Data Collection and Ground Driven Housekeeping Data Collection features from this group. Therefore just the paths associated with these features are utilized in SNOE, as seen in Figure C-1.

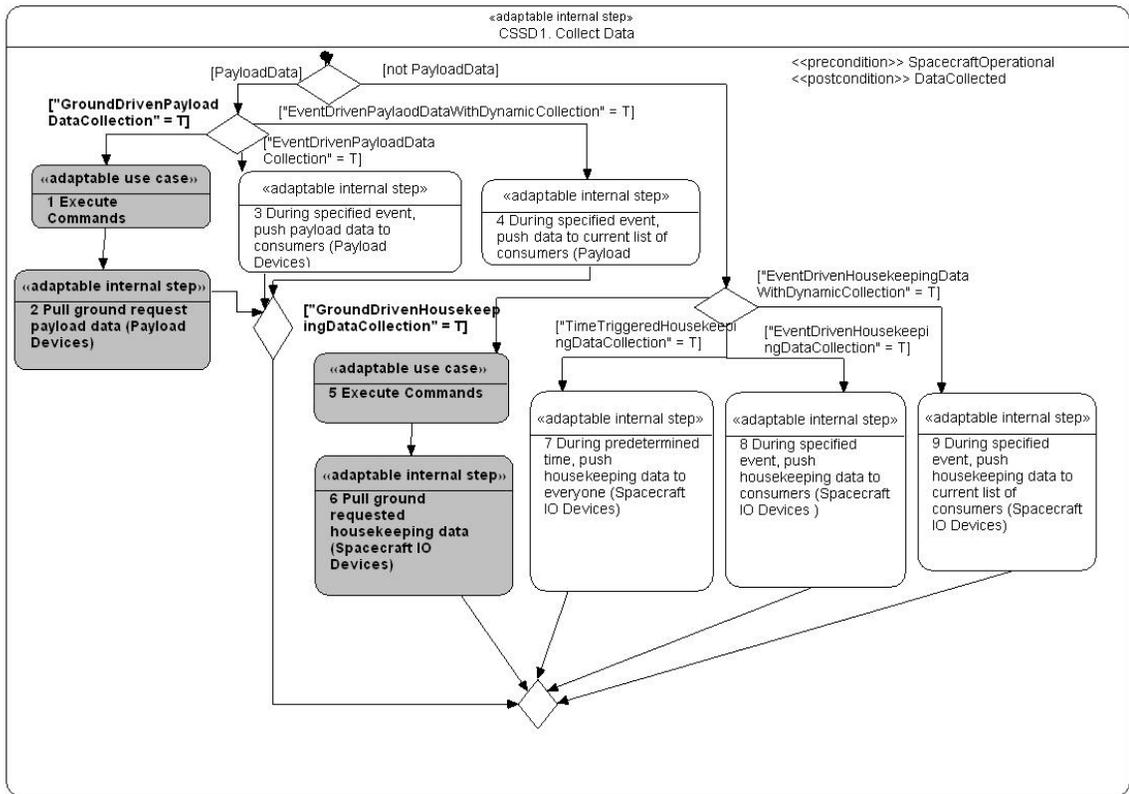


Figure C-1 SNOE's Collect Data sub-activity diagram

The second adaptable step is Format Telemetry. This adaptable step for SNOE is depicted in Figure C-2. This sub-activity diagram has two feature based conditions corresponding to the Telemetry Formation and Telemetry Formation Reliability pattern specific feature groups. SNOE does not use any features from the Telemetry Formation Reliability and uses the Low Volume Telemetry Formation feature. Therefore the path

corresponding to this feature selection is taken through the activity diagram, as seen in Figure C-2.

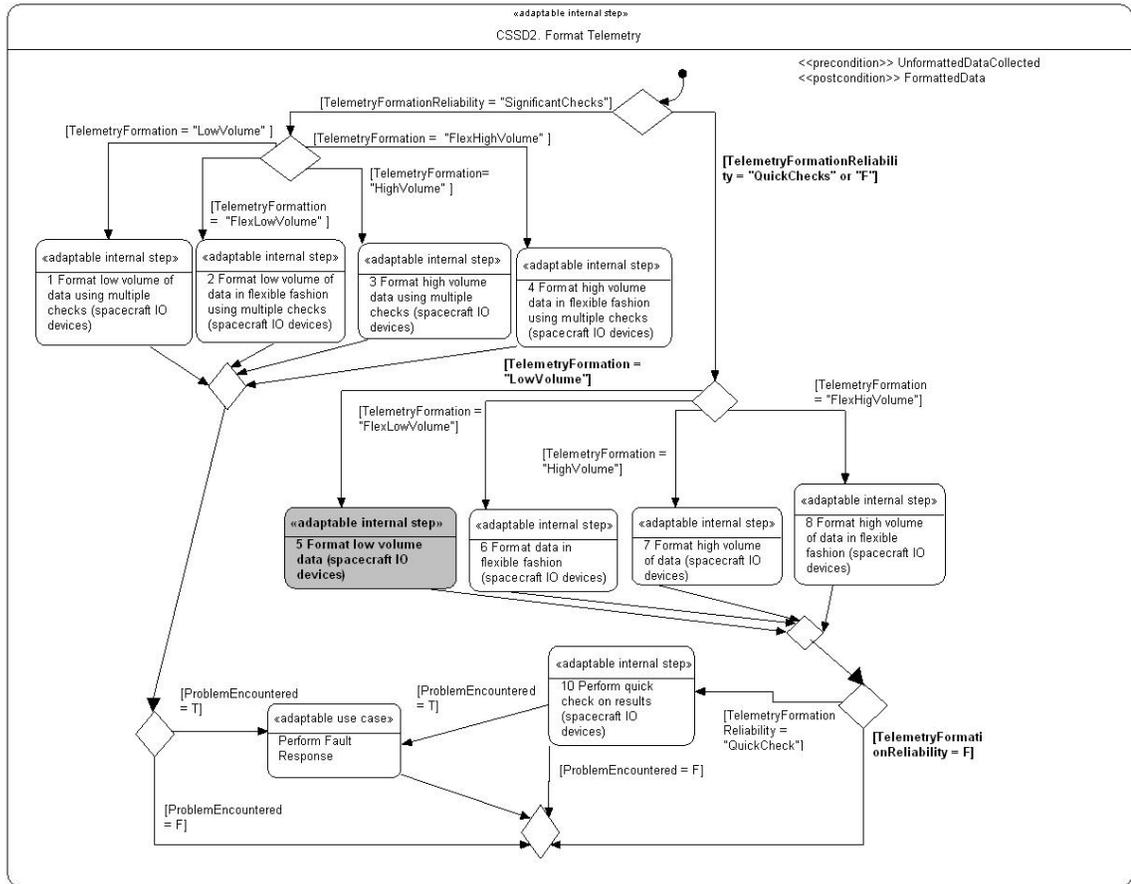


Figure C-2 SNOE's Format Telemetry sub-activity diagram

The final adaptable step in the Collect and Store Spacecraft Data use case is Store Data. This adaptable step has two feature based conditions corresponding to the Telemetry

Storage and Retrieval pattern specific feature group and the Memory Storage Device Type pattern variability feature group. From these groups, SNOE selected the Low Volume Telemetry Storage and Retrieval and EEPROM alternatives. Therefore the path associated with these features is utilized as seen in Figure C-3.

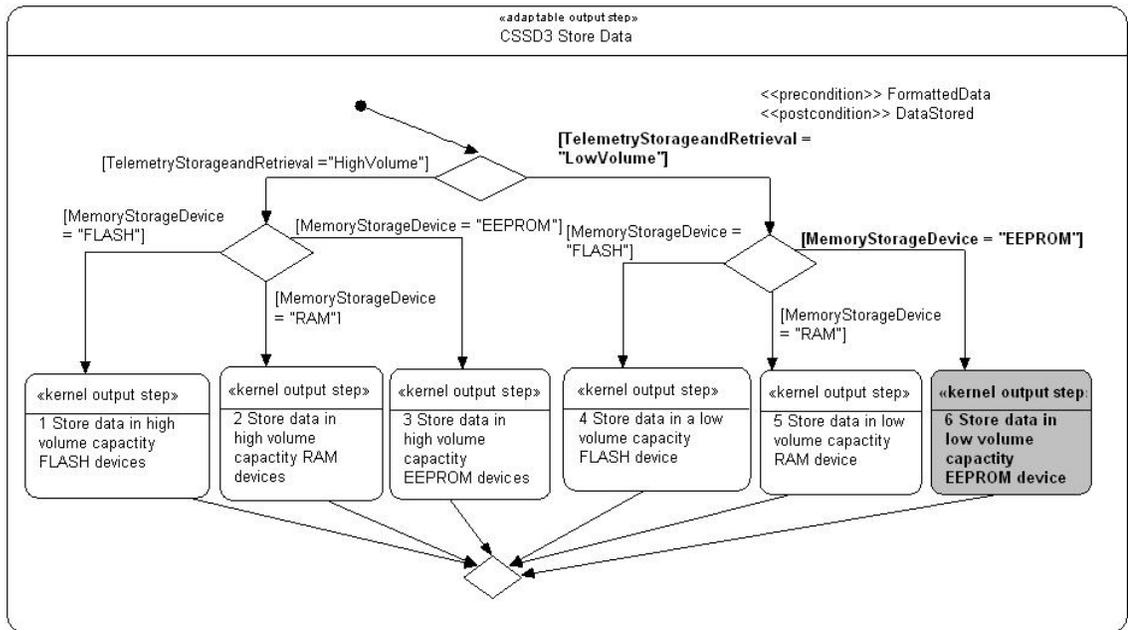


Figure C-3 SNOE's Store Data sub-activity diagram

C.2 SNOE Architectural and Executable Design Patterns

After the FSW architectural executable design patterns are selected, they must be customized to the SNOE to create SNOE level architectural and executable design

patterns following the process outlined in Chapter 5. The following subsections describe the SNOE specific changes made to some the design patterns.

C.2.1 SNOE Telemetry Client Server Executable Design Pattern

The SNOE Telemetry Client Server Executable Design Pattern contains the components necessary to store telemetry data in a single location. The components and their behavior are fully specified in the FSW SPL and thus there are not SNOE specific customizations to this design pattern.

C.2.2 SNOE Payload Multiple Client Multiple Server Executable Architectural Design Pattern

The next executable design pattern realized in SNOE is the FSW Payload Multiple Client Multiple Server executable design pattern. This design pattern is used to selectively collect payload data. Since the payload devices are unique to each SPL member, this design pattern needs to be customized to reflect SNOE's payload following the process described in Chapter 5.

The first architectural view that is updated is the collaboration diagram. The Payload_DClient and Payload_DServer components from the FSW SPL Multiple Client Multiple Server design pattern are replaced by the SNOE specific variants, as seen in Figure C-4. Since SNOE is required to selectively collect the payload data, separate data clients are created for each payload instrument. Additionally, since each payload

instrument has its own data buffer, separate server components are created for each payload instrument.

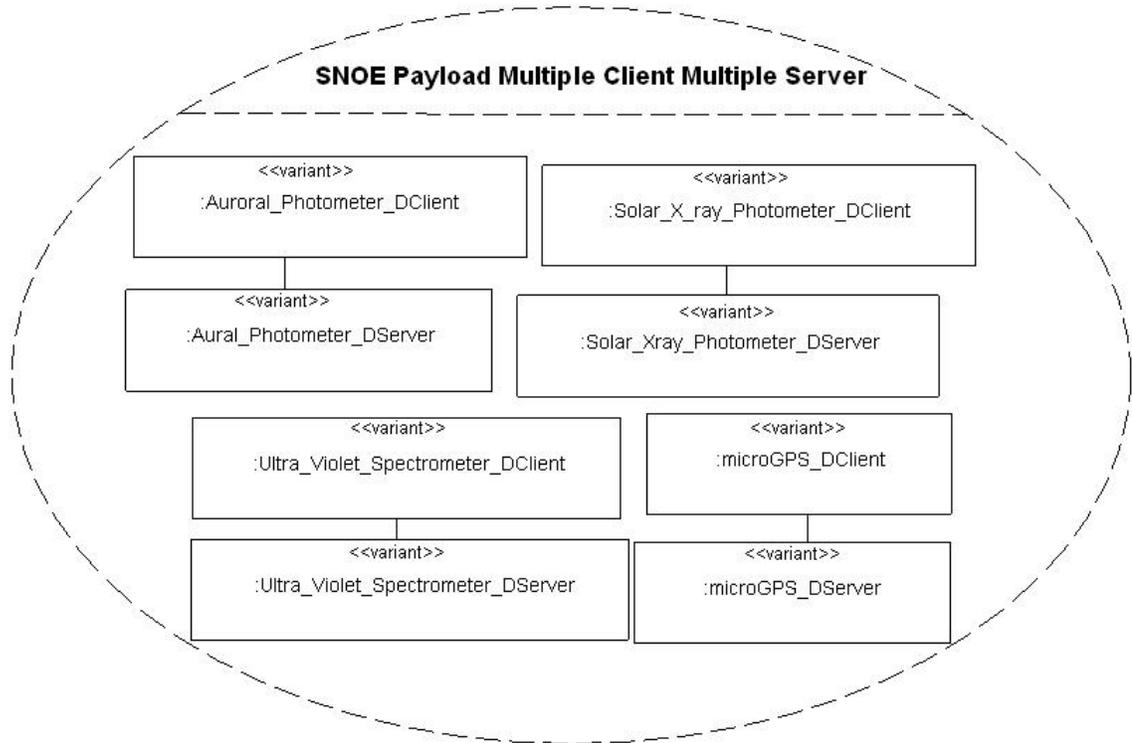


Figure C-4 Collaboration diagram for SNOE’s Payload Multiple Client Multiple Server executable design pattern

Next, the object interactions for the FSW Payload Multiple Client Multiple Server executable design pattern are customized for the SNOE application. As discussed in Chapter 5, this involves updating the representative object interactions with the precise

set of object interactions using the application's variants. The SNOE Multiple Client Multiple Server design pattern involves selectively collecting payload data, therefore SNOE specific interaction diagrams are created for each type of payload data that needs to be collected. The collect payload data interaction diagram from the FSW Payload Multiple Client Multiple Server executable design pattern, described in Chapter 6, contains a general purpose Payload_DClient and Payload_DServer components. Now that the specific payload variants devices are known, the interaction diagram is updated to reflect this information. The interaction diagram for collecting microGPS data is depicted in Figure C-5.

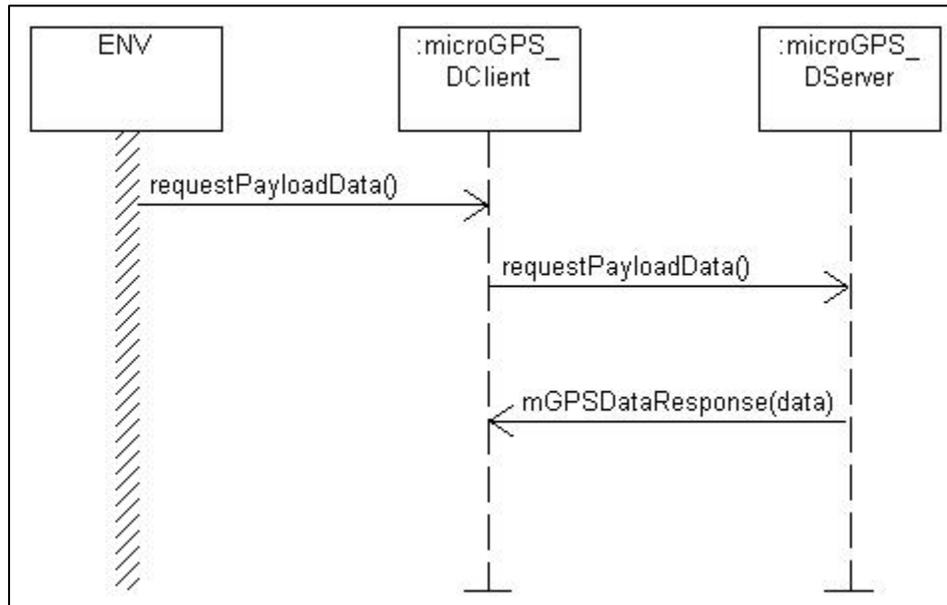


Figure C-5 Collect microGPS data scenario for SNOE

It can be seen that the FSW SPL components are replaced with the SNOE specific components and interactions. Similar diagrams are also created for the collecting auroral photometer, ultraviolet spectrometer, and solar x-ray photometer data.

Finally, the last architectural view that needs to be customized is the component diagrams. As discussed in Chapter 5, this involves customizing the component diagram with the application specific variants. Additionally, the ports and connectors for any application unique variants must also be modeled. Figure C-6 shows the updated component diagram for the SNOE Payload Data Multiple Client Multiple Server executable design pattern.

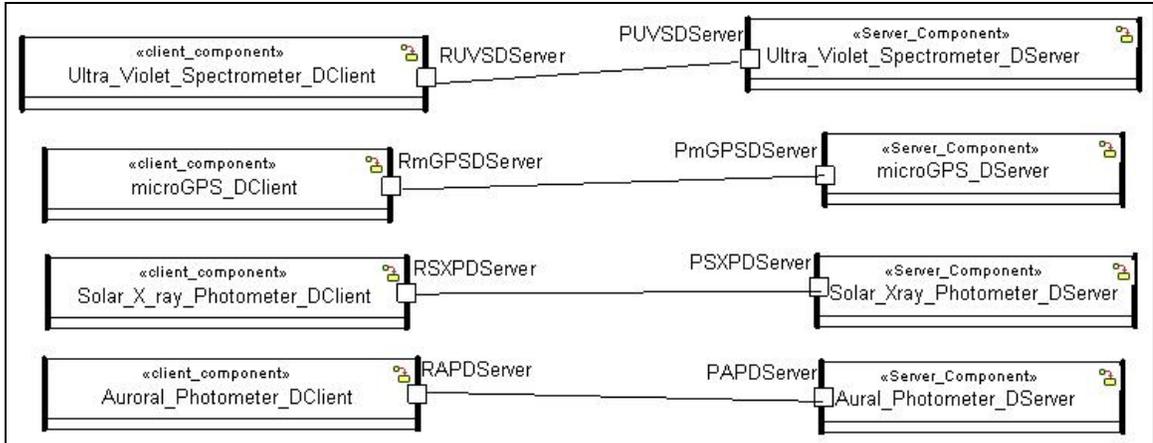


Figure C-6 SNOE Payload Data Multiple Client Multiple Server component diagram

It can be seen that the FSW Payload_DClient and Payload_DServer components are customized to the SNOE unique variants. The ports and connectors are added between the appropriate clients and servers are also shown in Figure C-6. Additionally, the interfaces are also updated to reflect the SNOE's unique variants. Figure C-7 shows that the connected components have compatible interfaces.

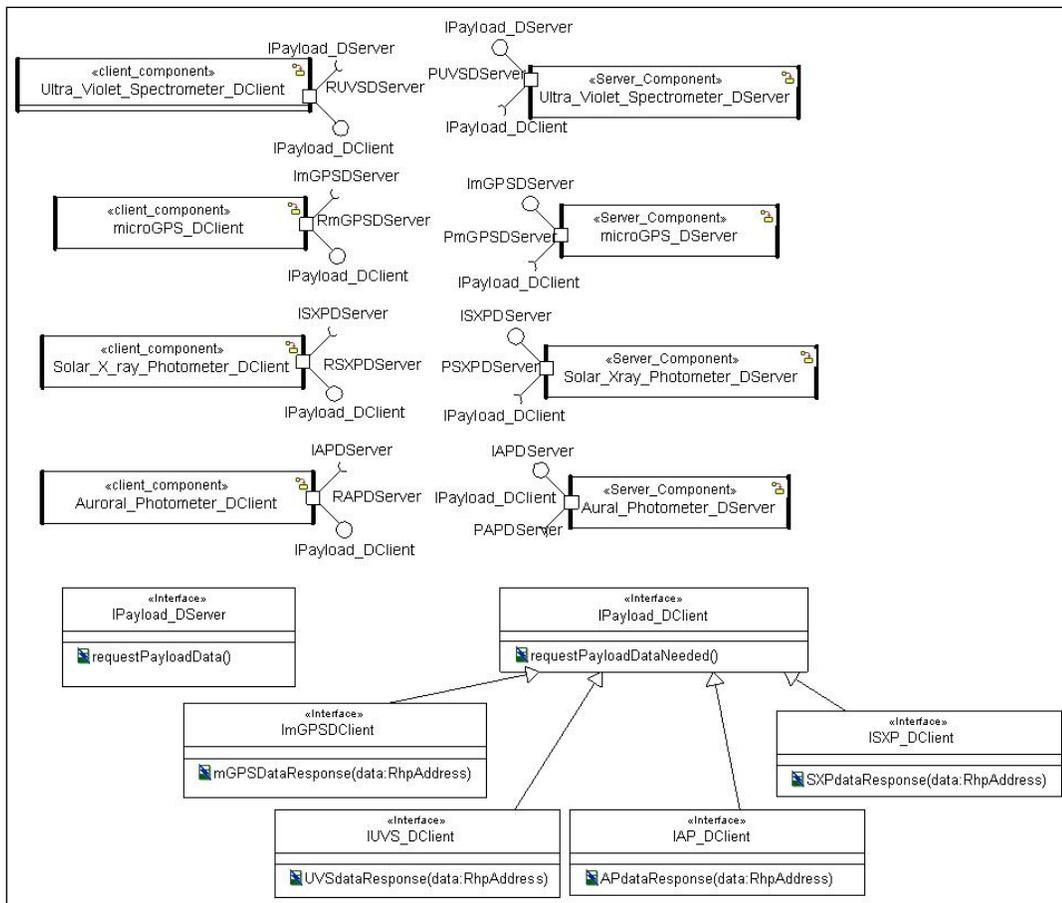


Figure C-7 SNOE Payload Data Multiple Client Multiple Server port design

The interfaces for the servers are the same as the Payload_DServer. The interfaces for the clients are essentially the same as the Payload_DClients, however the response messages are customized to reflect the instruments' data.

C.2.3 SNOE Payload Multiple Client Multiple Server Executable Design Pattern

In addition to updating the architectural views, the executable version of the design pattern also needs to be customized for SNOE. This is performed for each client and server in this design pattern. The specific steps involved in updating the state machine are follows.

First, the microGPS_DCClient component is responsible collecting the microGPS data from the microGPS_DServer. It is a SNOE specific variant of the FSW Payload_DCClient component from the FSW Payload Data Multiple Client Multiple Server executable design pattern. The state machine for the SNOE specific microGPS_DCClient component is depicted in Figure C-8. The SNOE specific variant does not refine the behavior of this component therefore no additional states are added. Next, now that the specific server the microGPS_DCClient requests the data from is known, this information is added to the actions on the state machine. This information is captured on the transition from the Preparing_Request state to the Idle state. The action `OUT_PORT(RmicroGPSDServer)->GEN(requestPayloadData(msg))` indicates that a request for payload data is being sent to the microGPS_DServer component through the RmicroGPSDServer port. Finally, the SNOE specific processing logic within the Preparing_Request and Processing_Response

states is added as On Entry actions. However, this information is not depicted in Figure C-8 in an effort to make the diagram readable.

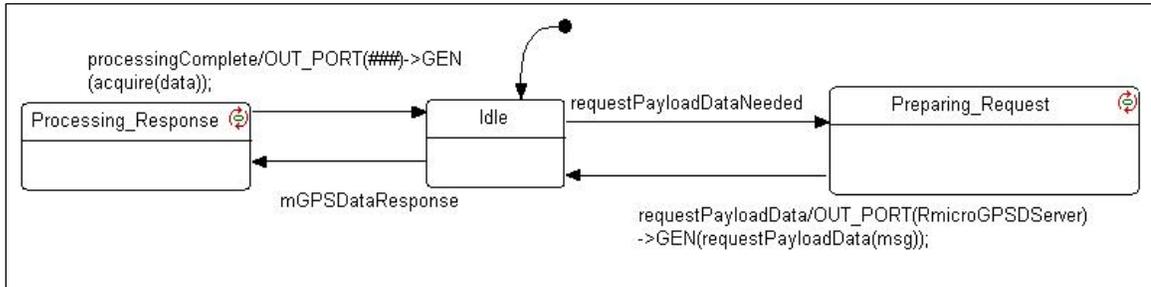


Figure C-8 microGPS_DClient state machine

Next the microGPS_DServer, Auroral_Photometer_DClient, Auroral_Photometer_DServer, Solar_Xray_Photometer_DClient, Solar_Xray_Photometer_DServer, Ultraviolet_Spectrometer_DClient and Ultraviolet_Spectrometer_DServer are also updated following a similar process.

C.2.4 SNOE Memory Storage Device Watchdog Executable Design Pattern

The FSW Memory Storage Device Watchdog Executable Design Pattern contains the components necessary to monitor the memory storage device for faults. The components and their behavior are common at the FSW and thus there are no SNOE specific customizations to this design pattern.

C.3 SNOE Collect and Store Spacecraft Data Design Pattern Interconnection

The Collect and Store Spacecraft Data use case has three main adaptable steps which are Collect Data, Format Telemetry, and Store Data. Each of these steps has its own sub-activity. First, Collect Data adaptable step for SNOE is depicted in Figure C-9. This sub-interaction overview diagram has two feature based conditions corresponding the Payload Data Collection feature group and Housekeeping Data Collection feature group. SNOE utilizes the Ground Driven Payload Data Collection and Ground Driven Housekeeping Data Collection features from this group. Therefore just the paths associated with these features are utilized in SNOE, as seen in Figure C-9.

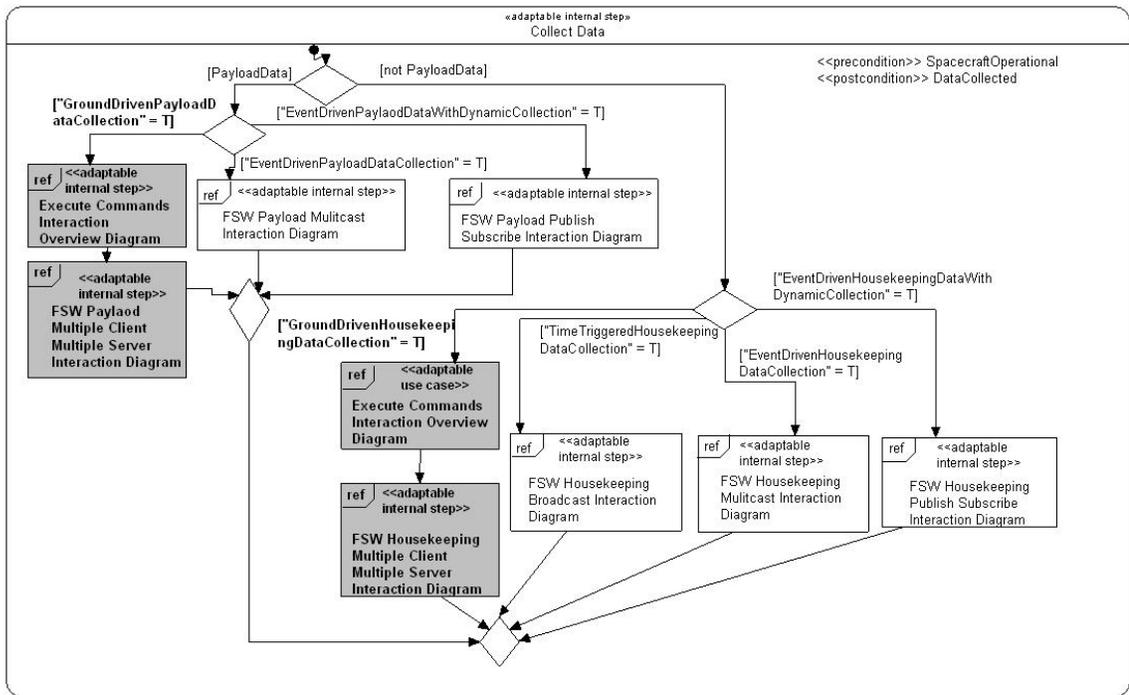


Figure C-9 SNOE's Collect Data sub-interaction overview diagram

The second adaptable step is Format Telemetry. This adaptable step for SNOE is depicted in Figure C-10. This step has two feature based conditions corresponding to the Telemetry Formation and Telemetry Formation Reliability pattern specific feature groups. SNOE does not use features from the Telemetry Formation Reliability and uses the Low Volume Telemetry Formation feature. Therefore the path corresponding to this feature selection is taken, as seen in Figure C-10.

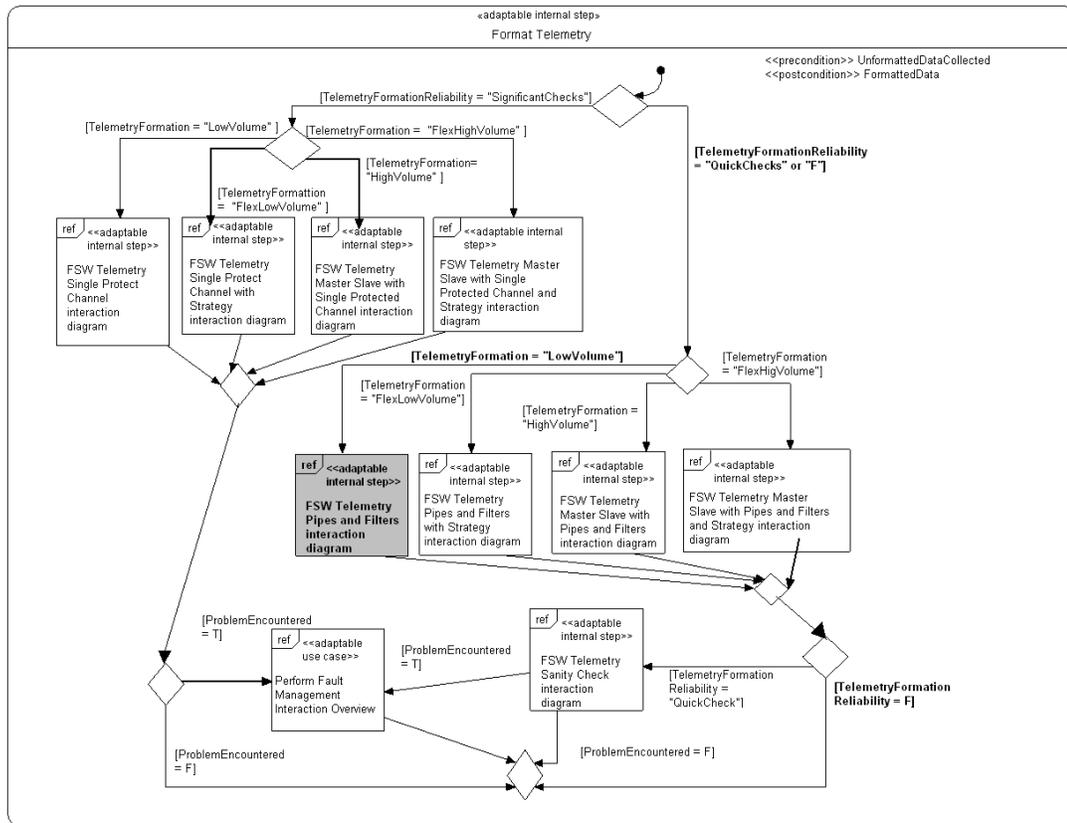


Figure C-10 SNOE's Format Telemetry sub-interaction overview diagram

The final adaptable step in the Collect and Store Spacecraft Data use case is Store Data. This adaptable step has two feature based conditions corresponding to the Telemetry Storage and Retrieval pattern specific feature group and the Memory Storage Device Type pattern variability feature group. From these groups, SNOE selected the Low Volume Telemetry Storage and Retrieval and EEPROM alternatives. Therefore the path associated with these features is utilized as seen in Figure C-11. Additionally, the ports and connectors between the design patterns not used and removed.

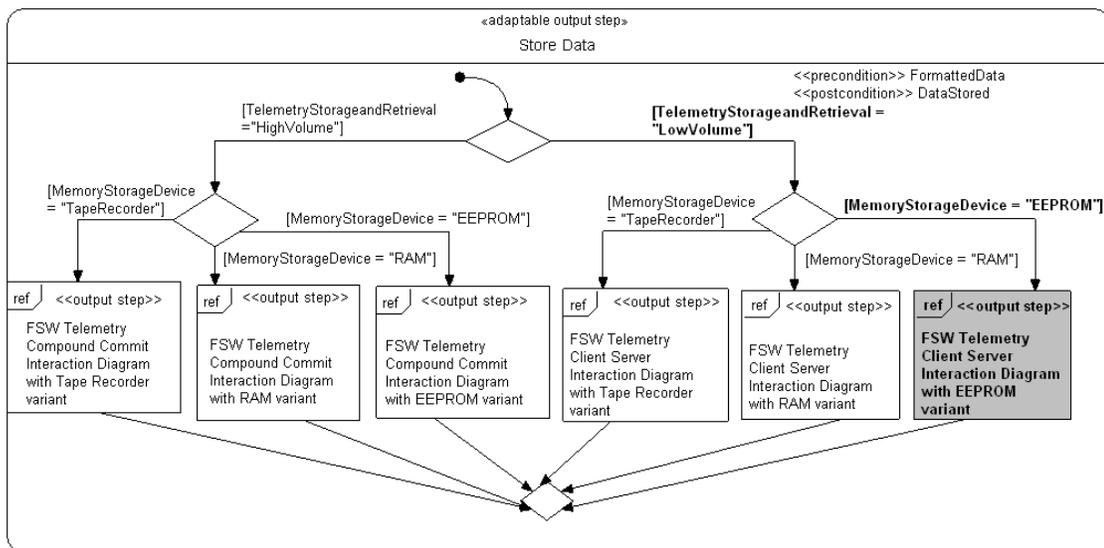


Figure C-11 SNOE's Store Data sub-interaction overview diagram

APPENDIX D. SOLAR TERRESTRIAL RELATIONS OBSERVATORY (STEREO) CASE STUDY

This appendix describes the Solar TERrestrial RELations Observatory (STEREO) case study. This case study is based on NASA's STEREO mission which is a two year mission to study the nature of coronal mass ejections (CMEs) by providing the first ever three dimensional images of the Sun. Information used in this case study is available here (John Galloway 1998; Johns Hopkins University Applied Physics Laboratory n.d.; Johns Hopkins University Applied Physics Laboratory n.d.; M.L. Kaiser et al. 2008; NASA Goddard Space Flight Center n.d.; Galvin et al. 2008; Howard et al. 2008; Luhmann et al. 2008; Anon n.d.; Anon 2007). Whenever possible, information from the real-world case study was used. However, when information was not available, assumptions were made based on personal experience in the flight software domain.

In order to take advantage of software architectural design patterns, STEREO will be designed as a member of the spacecraft FSW SPL described in Chapter 6. The application engineering of the STEREO FSW from the SPL is described as follows:

- Section D.1 provides a description of STEREO's FSW

- Section D.2 describes the requirements modeling phase for STEREO's FSW
- Section D.3 describes the analysis modeling phase for STEREO's FSW

D.1 Problem Description

The STEREO mission is a two year mission from NASA with a goal to provide the first ever three dimensional images of the Sun by studying the nature of coronal mass ejections (CME). The mission involves using two nearly identical three-axis stabilized spacecraft in heliocentric orbit, which is an orbit around the sun. Since the spacecraft is far away from Earth, the STEREO FSW relies less on real-time ground commanding and more on autonomous functionality. Additionally, since STEREO operates in a heliocentric orbit it requires guidance and control algorithms along with propulsion hardware to achieve and maintain its orbit. Additionally, the FSW needs to support maneuvering of STEREO for its scientific data collection, communications, power generation, and thermal control.

The STEREO spacecraft contains four payload instrument packages to accomplish its scientific mission. The payload packages are In-situ Measurements of Particles And CME Transients (IMPACT), PLAsma and SupraThermal Ion Composition (PLASTIC), STEREO/WAVES (S/WAVES), and Sun Earth Connection Coronal and Heliospheric Investigation (SECCHI). The IMPACT package measures solar wind electrons, energetic electrons, protons, heavier ions, and the in situ magnetic field strength and direction. PLASTIC measures the composition of heavy ions in the ambient plasma, protons, and

alpha particles. S/WAVES measures the generation and evolution of traveling radio disturbances. Finally, SECCHI uses remote sensing imagers and coronagraphs to track CMEs. A depiction of the STEREO spacecraft body along with the location of the various payload packages is depicted in Figure D-1.

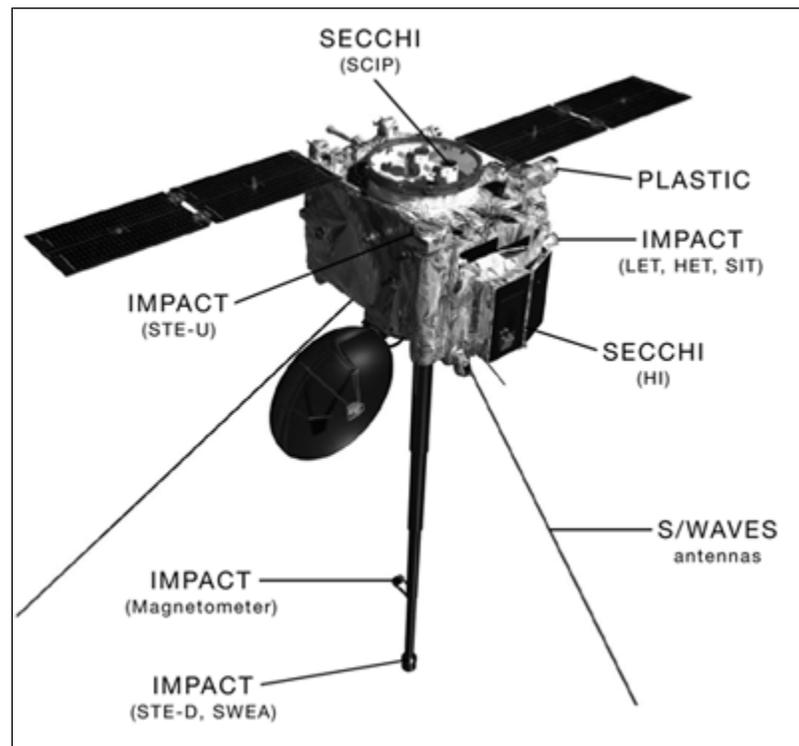


Figure D-1 STEREO spacecraft

The four payload packages contain a total of 23 different instruments to perform its scientific mission. The instruments and their associated payload package are listed below

in Table D-1. STEREO's payload packages are complex and therefore require more interaction and commanding from the FSW.

Table D-1 STEREO scientific instruments

Package	No.	Instrument
SECCHI	1	Ultraviolet imager
	2	Whitelight coronagrapher
	1	Heliospheric imager
	1	Guide telescope
PLASTIC	1	Solar wind sector small channel
	1	Solar wind sector main channel
	1	Wide angle partition
S/WAVES	3	Stacter antenna
	5	Radio receivers
IMPACT	1	Magenetometer
	1	Solar wind electron analyzer
	1	Suprathermal electron telescope
	1	Solar electron and proton telescope
	1	Suprathermal ion telescope
	1	Low energy telescope
	1	High energy telescope

The payload instruments collect data 24 hours a day, even when the spacecraft is in communication with the ground. During events of interest, the FSW must also enable the appropriate instruments to collect data at higher sampling rates. The FSW is responsible for performing some processing on the data such as data compression and formatting the data into telemetry packets. Additionally, the FSW must collect and store data from all

the payload packages. The data is pushed from the instrument data buffers to the FSW during predetermined time intervals and using predefined data rates. The only exception to this process SECCHI's imagers. This is because the payload packages generate around five gigabits of data per day with over 90% of the data coming from SECCHI images. To avoid overburdening the FSW, the SECCHI image data will be directly written to memory through Discrete Interface Card.

STEREO maintains its orientation in space using a three-axis stabilization technique, as opposed to a spin stabilization technique. In STEREO's three-axis stabilization technique, reaction wheel assemblies (RWAs) are mounted on various sides of the spacecraft and the appropriate RWAs are fired to make slight changes to the spacecraft's orientation. To adjust the spacecraft's attitude, adjustments are managed by firing thrusters to push STEREO to the proper attitude. Attitude determination and control is managed autonomously onboard the spacecraft by the FSW. The FSW determines the attitude and orientation using measurements from one star tracker, six sun sensors, and one Inertial Reference Unit (IRU). If the FSW determines that the attitude and orientation are out of the acceptable range, then it must determine and send the appropriate commands to the RWAs and/or thrusters to adjust the spacecraft's attitude.

STEREO uses two movable solar array appendages to generate power. The FSW is responsible for positioning the solar arrays toward the sun. Onboard power is controlled

using a power distribution unit (PDU). To maintain a consistent temperature, STEREO spacecraft uses both active and passive means of thermal control. The active measures include thermistors and electric heaters to ensure the spacecraft remains a consistent temperature throughout the spacecraft since one side of the STEREO is facing the Sun and the other does not. The FSW is responsible for monitoring the spacecraft temperatures and sending commands to the appropriate heaters to adjust the temperature. The passive thermal control measures do not interface with the FSW.

STEREO downlinks its data once per day to the ground through NASA's Deep Space Network (DSN) station in Canberra, Australia. To communicate with the ground STEREO contains a low gain antenna (LGA), medium gain antenna (MGA), and high gain antenna (HGA) that will be used depending on where the spacecraft is in orbit. The FSW is responsible to selecting and using the right antenna at the right time. The LGA and MGA are fixed, however the STEREO FSW is responsible for autonomously controlling the HGA so that is pointed at Earth.

All data and commands exchanged between STEREO and the ground station are formatted using Consultative Committee for Space Data Systems (CCSDS) standards. It is the responsibility of the FSW to format all spacecraft data using this format. During normal operations STEREO transmits telemetry at a 480 kbps rate and receives commands at a 125 kbps rate. When the spacecraft is out of contact of the ground, the

FSW writes all data to multiple EEPROM solid state recorders and downlinks the data during the next contact period.

When STEREO is out of contact with the ground it must autonomously detect, log, and respond to onboard faults. The fault management functionality on STEREO is quite extensive since STEREO is not in constant contact with the ground station. This is performed by collecting and monitoring engineering data from all the hardware devices. The engineering data for the payload packages is pushed to the FSW during predefined time interval using predefined data rates. The engineering data from the rest of the hardware is pulled by the FSW. Examples of faults to detect include single-event-upset recovery and response to monitored voltage or current telemetry conditions with pre-programmed actions. Additionally, the FSW is responsible to switching to the redundant hardware if faults are detected with the primary hardware. The redundant hardware on board includes an inertial measurement unit (IMU), three independent thruster groups, an extra reaction wheel assembly, transmitter, and receiver. All engineering data, faults detected, and corrective actions performed must also be reported to the ground.

The STEREO spacecraft contains two flight computers, one for the guidance and control processing and one for all other processing. Both flight computers are Mongoose V CPUs clocked at 12 MHz running the Nucleus+ RTOS. The throughput capacity of each of STEREO's flight computers is estimated at 9.6 MIPS. STEREO's flight computers

are also equipped with 2MByte SRAM, 4MByte EEPROM, 1MByte console boot ROM. The subsystem intercommunication is handled using MIL-STD-1553B. Additionally, a 7.5Gbits capacity solid state recorder is also provided with the C&DH flight computer for storing telemetry. It has an 8Mb/s simultaneous read and write rate. Additionally, a Housekeeping IO card is used to perform the Health and Safety monitoring of the spacecraft using engineering data.

In addition to the two main CPUs, STEREO's payload packages also contain separate electronics to control for the payload. The SECCHI package utilizes a RAD750 processor that is capable of 120MIPS also with several circuit boards. The S/WAVES package contains an ADSP processor from the ADSP-2100 family for controlling the payload instruments. Additionally, the S/WAVES and IMPACT packages share a common SA33000 microprocessor along with RAM and ROM for data processing. IMPACT also utilizes an 80CRH196KD microprocessor and circuit boards to control the payload instruments. Finally, the PLASTIC package utilizes over 30 circuit boards including Field Programmable Gate Arrays (FPGAs).

D.2 STEREO Flight Software Requirements Modeling

The first step in developing STEREO's flight software architecture is to perform requirements modeling to scope the STEREO SPL member. This is accomplished by deriving STEREO's requirements modeling artifacts from the FSW SPL requirements

modeling artifacts described in Chapter 6. The following subsections describe this process in more detail.

D.2.1 STEREO Feature Model

The first step in the requirements modeling phase is to select the features that STEREO will realize from the FSW Feature Model described in Chapter 6. First the C&DH feature model is examined. The final pattern specific C&DH feature model for STEREO is depicted in Figure D-2, where the features selected are highlighted using a bold font and a gray background. By definition, STEREO must provide all of the FSW SPL C&DH kernel pattern specific features. There is only one kernel feature which is the Memory Storage Device Fault detection feature.

Additionally, STEREO must also provide one feature from the exactly-one-of pattern specific feature groups. The FSW SPL C&DH feature model contains three of these feature groups. The feature selection from each of these groups is described below.

- **Command Execution.** STEREO's payload and hardware requires a large amount of commanding to execute. Additionally, it does not need the ability to support new commands or strict temporal predictability. Therefore the High Volume Command Execution feature is selected.

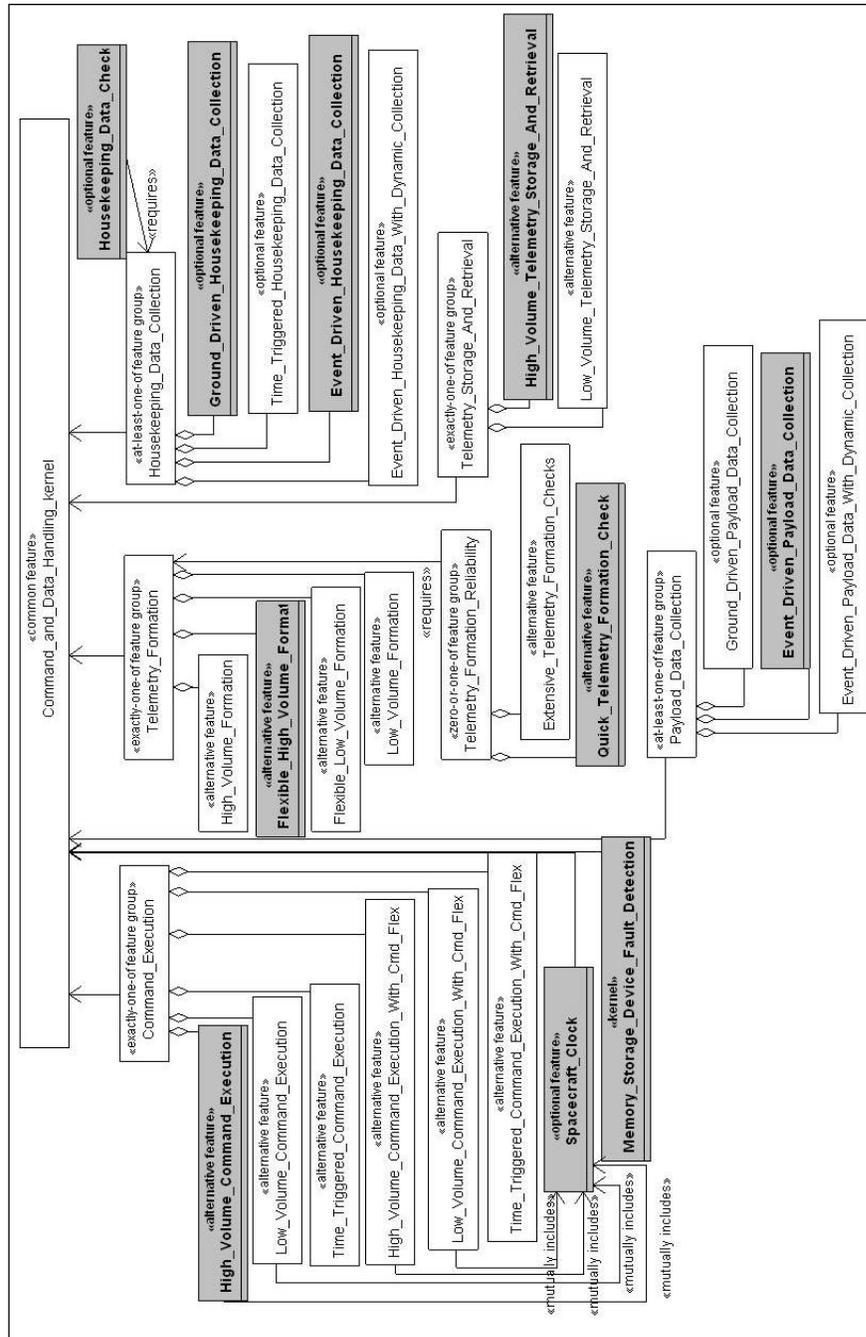


Figure D-2 STEREO C&DH Feature Model

- **Telemetry Formation.** STEREO does produce a large volume of payload and housekeeping data. However, the other subsystems are required to format their data locally. Therefore the telemetry formation performed by the C&DH subsystem is low because it only formats the C&DH housekeeping data. Additionally, STEREO follows the CCSDS standard and needs to be updated if the standard changes. Thus the Flexible Serial Formatting feature is the best fit for STEREO.
- **Telemetry Storage and Retrieval.** Due to the large volume of payload and housekeeping data produced by STEREO, the High Volume Telemetry Storage and Retrieval feature is selected.

Next, STEREO must also provide one or more features from the at-least-one-of pattern specific feature groups. The FSW C&DH feature model contains two of these feature groups. The feature selection from each of these groups is described below.

- **Payload Data Collection.** On STEREO, all of the payload data is sent to the C&DH subsystem at predefined intervals and rates. Therefore only the Event Driven Payload Data Collection feature is selected.
- **Housekeeping Data Collection.** According to STEREO's description in section D.1, all of the payload data and payload housekeeping data is sent to the C&DH subsystem at predefined intervals and rates. Therefore the Event Driven Housekeeping Data Collection feature is selected. The collection of housekeeping data from the other subsystems is ground driven. Therefore the Ground Driven Housekeeping Data Collection feature is also selected.

Next, the optional pattern specific features that STEREO should provide are determined. In the FSW C&DH feature model there are two optional features. The optional features are Spacecraft Clock and Housekeeping Data Checks. According to the STEREO description, trending is required of the thus the Housekeeping Data Checks feature is selected. STEREO's High Volume Command Execution feature mutually includes the Spacecraft Clock pattern specific feature. Therefore, STEREO must provide the optional Spacecraft Clock pattern specific feature.

Finally, the last step in STEREO's feature model is to determine the optional C&DH pattern specific feature groups that STEREO should provide. Optional pattern specific feature groups include the zero-or-one or zero-or-more feature groups. In the FSW SPL model there is one optional feature groups, which is Format Telemetry Reliability. According to the description of STEREO, the FSW does have the responsibility to detect faults in the telemetry processing and report problems to the ground. Thus the Telemetry Formation Quick Check is selected from this group.

Next, the pattern specifics the STEREO realizes are selected from the FSW SPL's pattern specific features. The final C&DH pattern variability feature model for STEREO is depicted in Figure D-3, where the features selected are highlighted using a bold font and a gray background. By definition, STEREO must provide one feature from the FSW SPL C&DH exactly-one-of pattern variability feature groups.

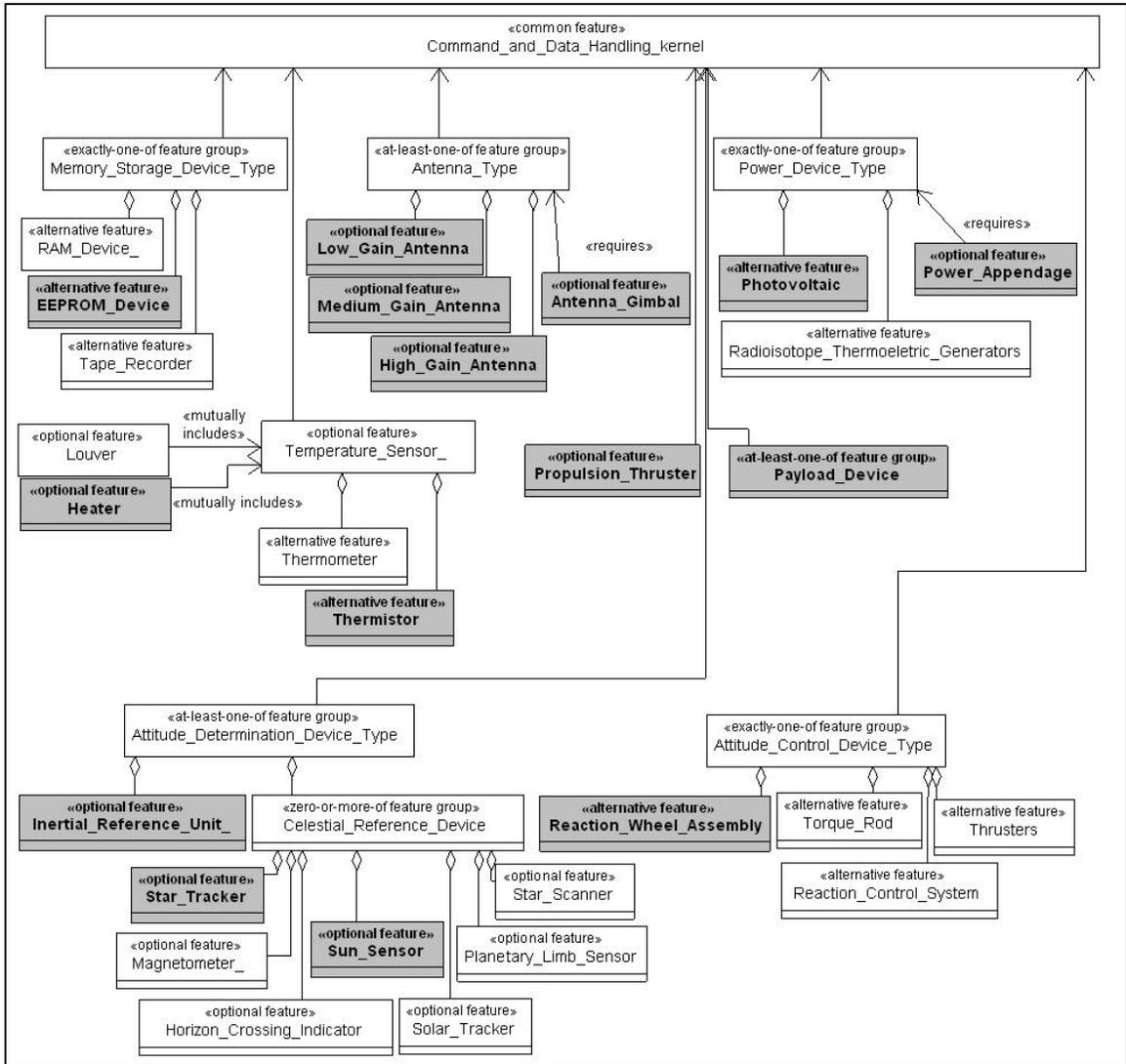


Figure D-3 STEREO pattern variability feature selection

The FSW SPL C&DH feature model contains three of these feature groups. The feature selection from each of these groups is described below.

- **Memory Storage Device Type.** According to STEREO's description, all telemetry data is stored in FLASH solid state recorders. Therefore the FLASH Device feature is selected
- **Power Device Type.** STEREO's use two solar panels which are used to generate power. Therefore the photovoltaic feature is selected.
- **Attitude Control Device Type.** According to STEREO's description the attitude is adjusted by using a reaction wheel assembly (RWA). Thus the RWA feature is selected.

Next, STEREO must also provide one or more features from the at-least-one-of pattern specific feature groups. The FSW C&DH feature model contains two of these feature groups. The feature selection from each of these groups is described below.

- **Antenna Type.** To perform its communication, STEREO utilizes an immobile low gain antenna (LGA), an immobile medium gain antenna (MGA), and a moveable high gain antenna (HGA). Therefore the LGA, MGA, and HGA pattern variability features are selected.
- **Attitude Determination Device Type.** STEREO measures its attitude with a star tracker, multiple sun sensors, and an inertial reference unit (IRU). Therefore these pattern variability features are selected.

Next, the optional pattern variability features that STEREO should provide are determined. In the FSW SPL C&DH pattern variability feature model there are three optional features.

- **Propulsion Thruster.** STEREO orbits around the sun and must execute a flight plan to achieve its orbit. Therefore it must have propulsion and as a consequence the propulsion thruster pattern variability feature is selected
- **Antenna Gimbal.** STEREO does require a moveable high gain antenna (HGA), therefore this feature is selected.
- **Power Appendage.** According to STEREO's description, its solar panels are attached to movable solar panel wings, thus this feature is selected.
- **Heater.** STEREO provides active thermal control. According to STEREO's description it uses heaters to adjust the spacecraft's temperature. Therefore this feature is selected.
- **Louver.** STEREO provides active thermal control, however it does not utilize louvers to adjust the spacecraft's temperature. Thus this feature is not selected

Finally, the optional pattern variability feature groups that STEREO should provide are identified. In the FSW C&DH feature model there is only one optional feature groups, which is the Temperature Measurement Device feature group. STEREO provides active thermal control and measures the spacecraft's temperature with multiple thermistors.

D.2.2 STEREO Use Case Model

This subsection describes the use case model for the STEREO SPL member. The use case model for STEREO is derived from the FSW SPL use case model defined in Chapter 6. The final use case model for STEREO is depicted in Figure D-4, where the use cases and actors selected are highlighted using a bold font and a gray background. By definition, STEREO must provide all of the kernel use cases. These include the Collect & Store Data, Execute Commands, Perform Fault Management, and Uplink and Downlink Telemetry. STEREO must also interface with all the kernel actors.

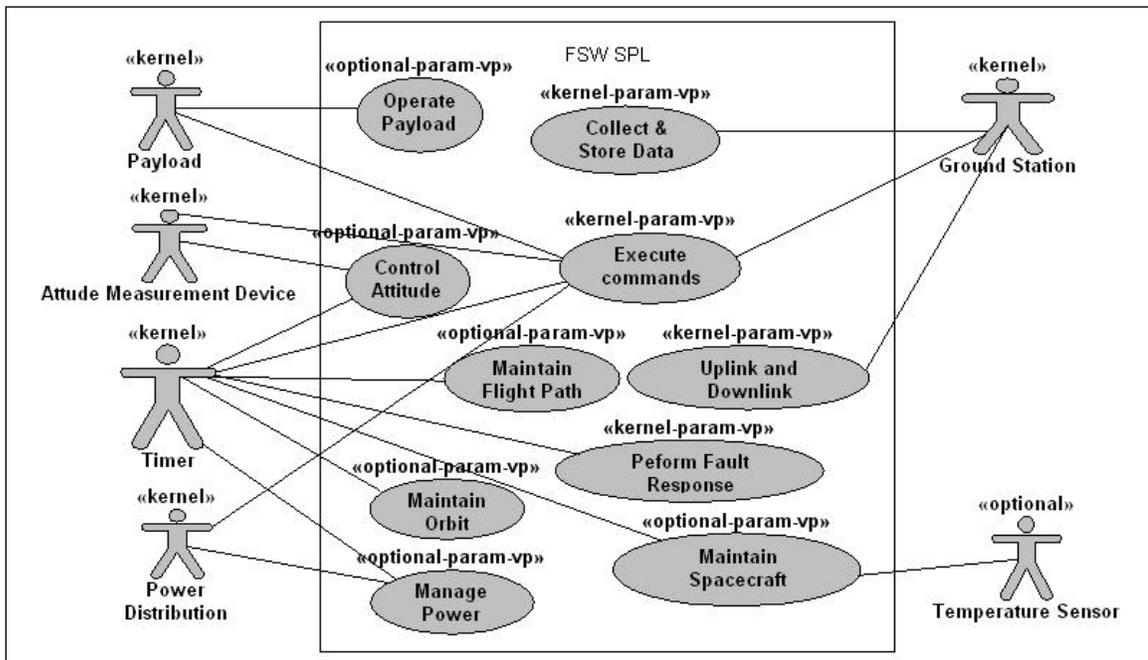


Figure D-4 STEREO Use Case Model

Next, the optional use cases and actors that are applicable for STEREO are determined. Based on STEREO's feature selection, it can be determined that all the optional use cases are realized by STEREO.

D.2.3 STEREO Use Case Activity Model

The next step in the application engineer process is to customize the FSW SPL use case activity models for STEREO. This is accomplished by only utilizing the feature based condition paths that corresponds to STEREO's feature selection. This is demonstrated using the Execute Commands use case. This use case involves executing commands from the ground station to ensure the spacecraft is not put into an unsafe state and the actions are appropriate for the spacecraft's mode. STEREO's activity diagram for this use case is depicted in Figure D-5. This use case scenario begins with a feature condition on the Command Execution pattern specific feature group. STEREO selected the High Volume Command Execution feature, thus the path that corresponds to this feature selection is taken as seen in Figure D-5.

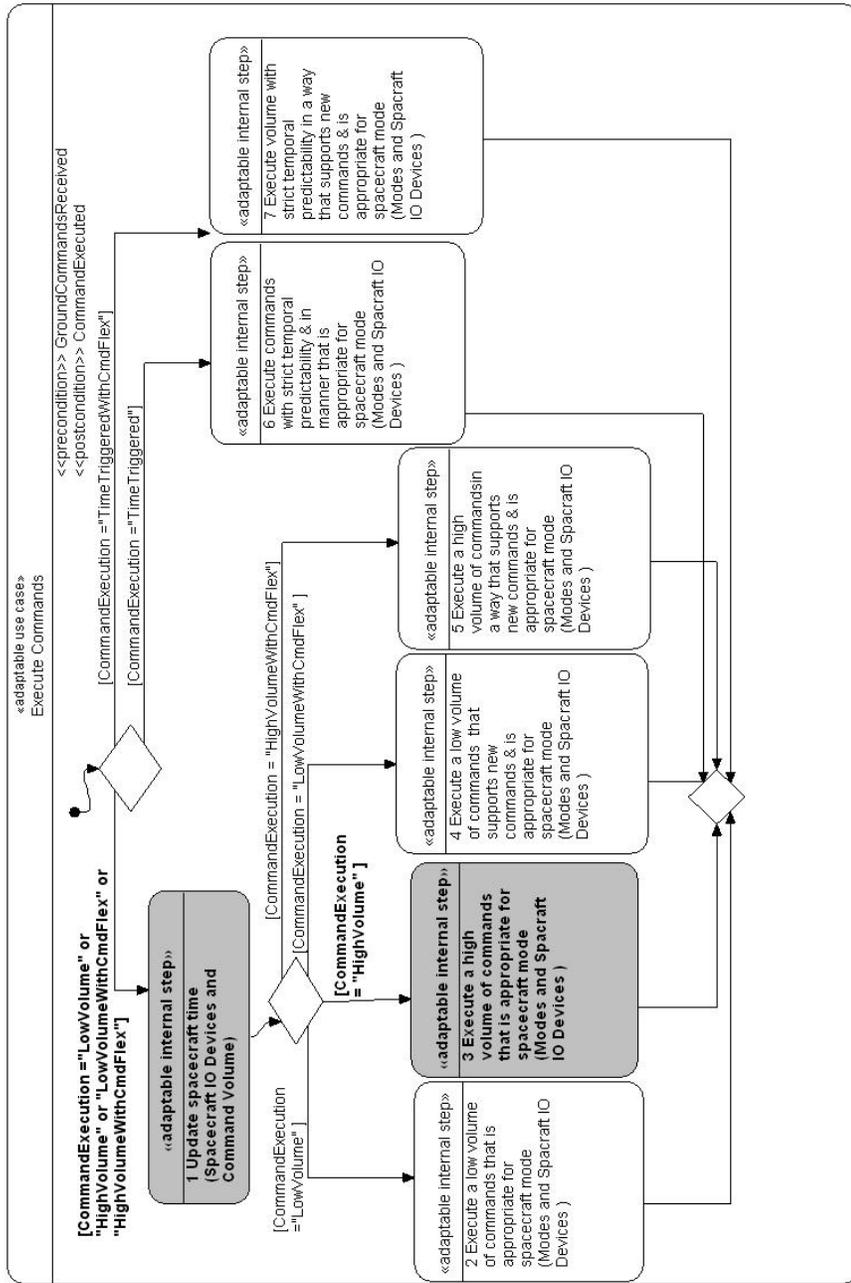


Figure D-5 STEREO Execute Commands use case activity model

D.2.4 STEREO Conceptual Static Model

The next requirements modeling artifact is STEREO's conceptual static model, which is derived from the spacecraft FSW SPL conceptual static model described in Chapter 6. The conceptual static model for STEREO is depicted below in Figure D-6, where the optional and variant devices selected are highlighted using a bold font and a gray background. By definition, STEREO must provide the kernel devices. Therefore the transmitter, receiver, payload device, and power distribution unit (PDU) is selected from the SPL conceptual static model, as seen in Figure D-6.

Next, the optional devices and variant devices are selected based on STEREO's feature selection. According to STEREO's feature selection, STEREO utilizes an immobile low gain antenna (LGA), an immobile medium gain antenna (MGA), a moveable high gain antenna (HGA), a transmitter, and a receiver. Therefore these variants are selected from the Antenna pattern variability feature group.

According to the STEREO feature model, STEREO provides active thermal control using thermistors and heaters. Therefore these devices are selected from the FSW SPL conceptual static model as depicted in Figure D-6. Additionally, STEREO is required to store a high volume of data using multiple EEPROM devices, hence the EEPROM memory device variant is selected.

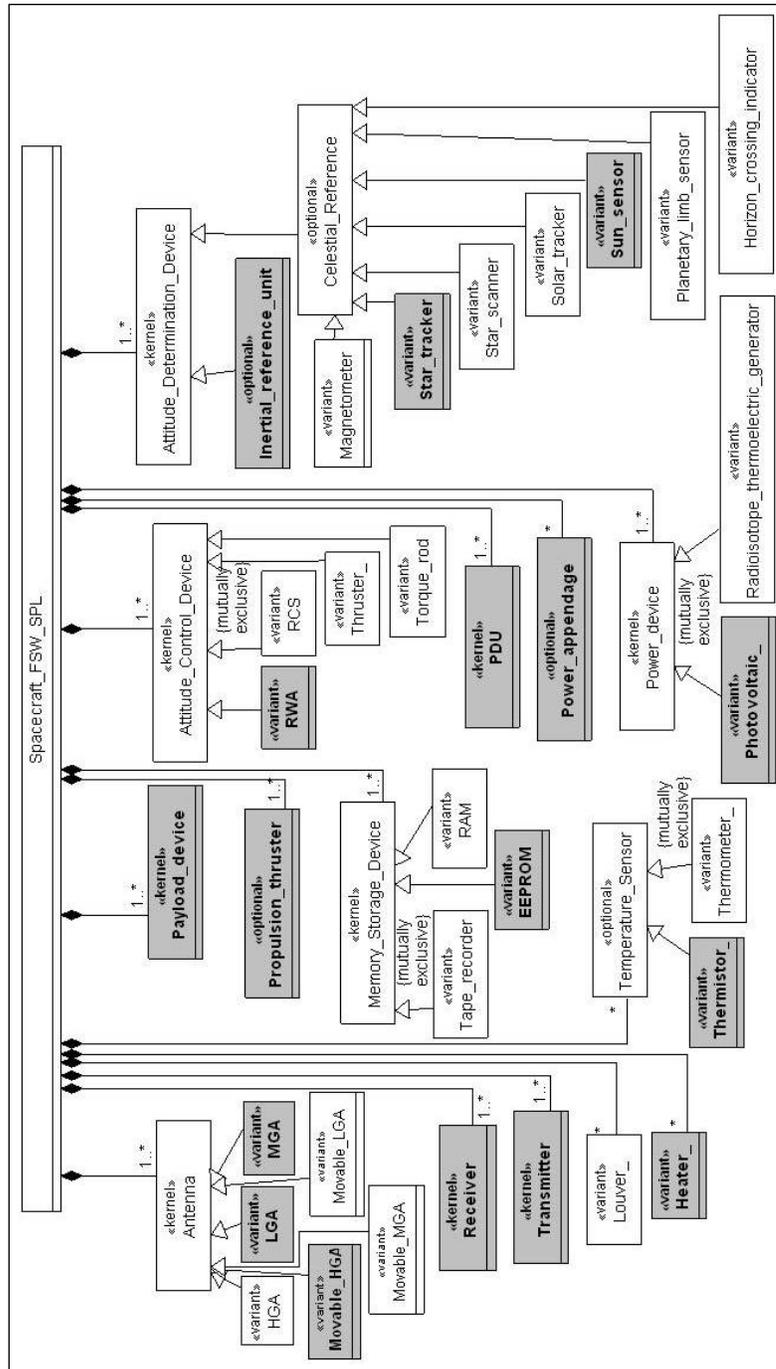


Figure D-6 STEREO conceptual static model

Next, STEREO orbits around the sun and must execute a flight plan to achieve its orbit. Therefore it must have propulsion and as a consequence the propulsion thrusters are selected. According the problem description, STEREO controls its attitude using reaction wheel assemblies (RWAs), thus the RWA variant is selected from the attitude control devices. STEREO measures its attitude with a star tracker, multiple sun sensors, and an inertial reference unit (IRU). Therefore these variants are selected for the attitude determination devices.

Finally, STEREO FSW interfaces with the power distribution unit (PDU) and movable solar panels to generate and control the spacecraft's power. Consequently, the kernel PDU and photovoltaic variant and power appendage are selected.

D.2.5 STEREO Context Model

Another major artifact produced from the requirements modeling phase is the context diagram. The context diagram is used to help define the system's boundary with the external environment. STEREO's context diagram is derived from the FSW SPL context diagram described in Chapter 6. Figure D-7 depicts the SPL context diagram with the variable devices selected for STEREO. The variable devices selected are highlighted using a bold font and a gray background. Figure D-7 shows that STEREO does not interface with louvers because it only uses temperature sensors and heaters to provide active thermal control. STEREO must interface with the rest of the external devices to meet its requirements.

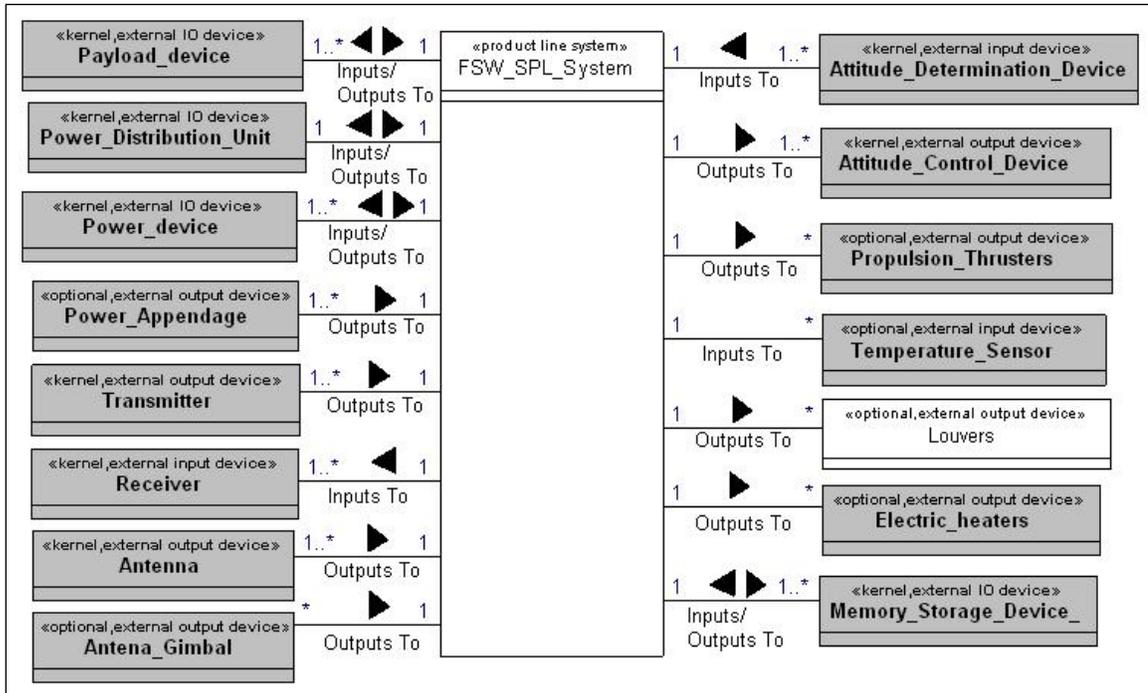


Figure D-7 STEREO SPL context diagram

Next, the STEREO context diagram can be refined to depict the actual variant devices that STEREO uses. Figure D-8 illustrates the final system context diagram for STEREO. STEREO contains four payload packages, which are the In-situ Measurements of Particles And CME Transients (IMPACT), PLASMA and SupraThermal Ion Composition (PLASTIC), STEREO/WAVES (S/WAVES), and Sun Earth Connection Coronal and Heliospheric Investigation (SECCHI). There are a total of 23 different instrument across the payload packages. To keep the diagram readable, the payload packages and not the instruments are depicted in Figure D-8. STEREO uses multiple

solid state recorder for its memory storage device and photovoltaics for its power devices, thus these variants are shown in the system context diagram. Additionally, the optional power appendages, thermal devices, and propulsion thrusters that STEREO uses are also depicted in the system context diagram. All of the variant antennas and optional antenna gimbal that STEREO utilizes for its communications are also depicted Figure D-8. Finally, the specific devices that STEREO utilizes for its attitude control and attitude determination are depicted in the system context diagram.

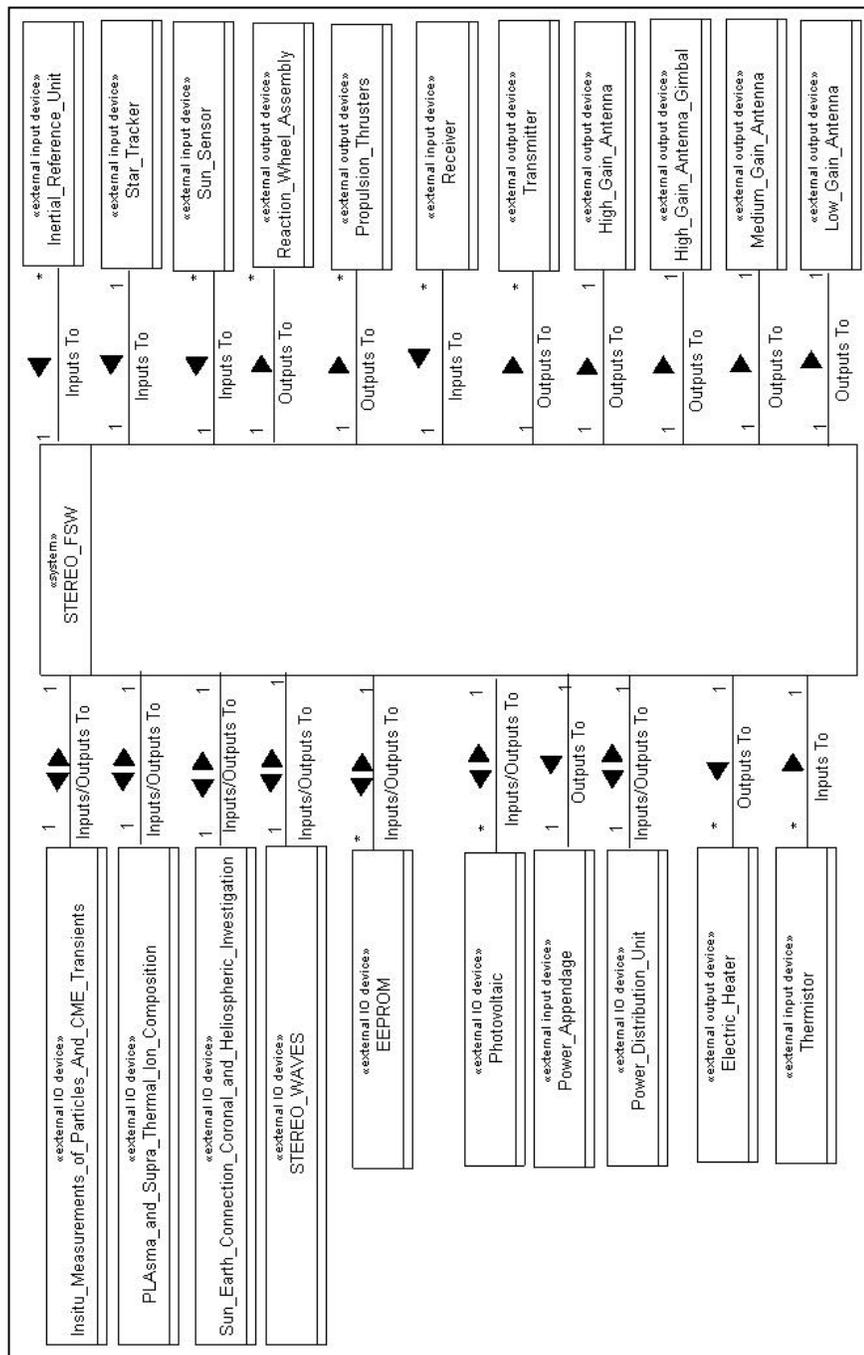


Figure D-8 STEREO system context diagram

D.2.6 STEREO Subsystem Structuring

Another key artifact from the requirements modeling phase is the subsystem structuring diagram. The subsystem structuring for STEREO is identified from the FSW SPL subsystem structuring described in Chapter 6. The major relationships between STEREO's subsystems are shown in Figure D-9, where the subsystems realized highlighted using a bold font and a grayed background.

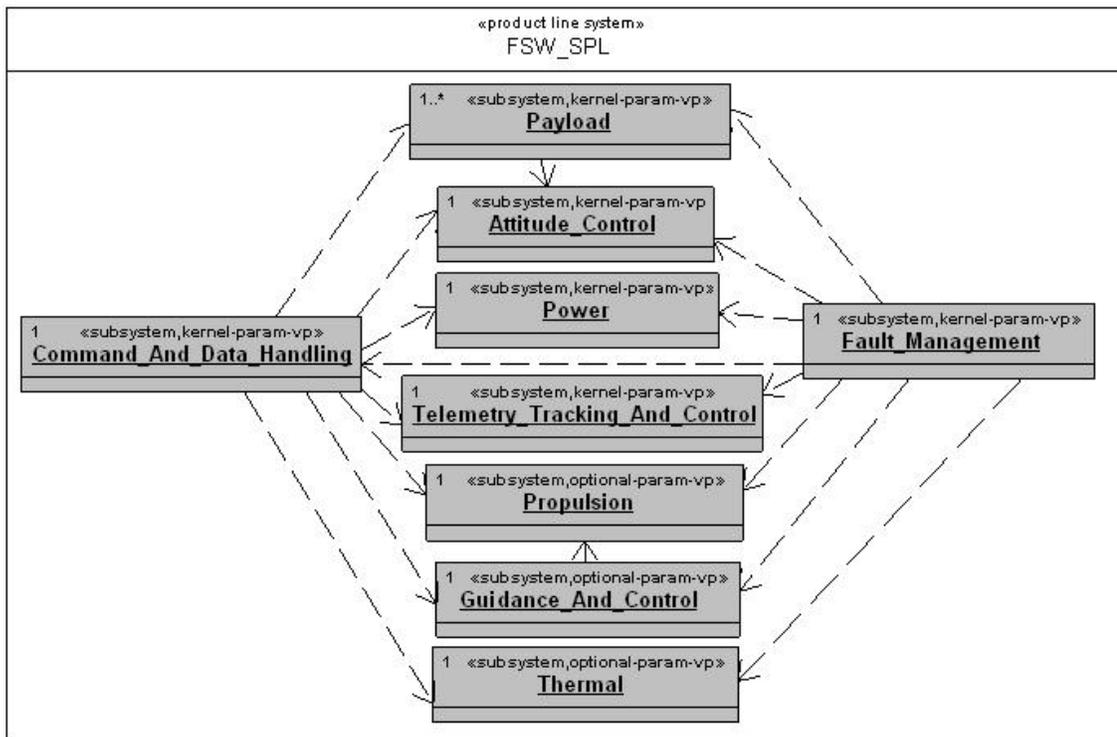


Figure D-9 STEREO subsystem dependency model

STEREO must provide the five kernel subsystems; therefore it has Command and Data Handling, Telemetry Tracking and Control, Attitude Control, Payload, and Power. It is clear from the STEREO feature model that STEREO also provides extensive fault management functionality. Therefore the optional Fault Management subsystem is realized. It is also clear from STEREO's feature model that it does provide active thermal control, active propulsion, and guidance and control. Thus the Thermal, Propulsion and Guidance and Control subsystems are also selected for STEREO.

Another view of the subsystem structuring is a refinement of the context diagram, which is depicted in Figure D-10. This diagram shows the subsystems along with the allocation to STEREO's actual physical devices. It can be seen from Figure D-10 that the allocation of devices follows the FSW SPL allocation where devices are grouped functionally with the subsystems. For example, all the devices associated with attitude control and attitude determination are associated to the attitude control subsystem.

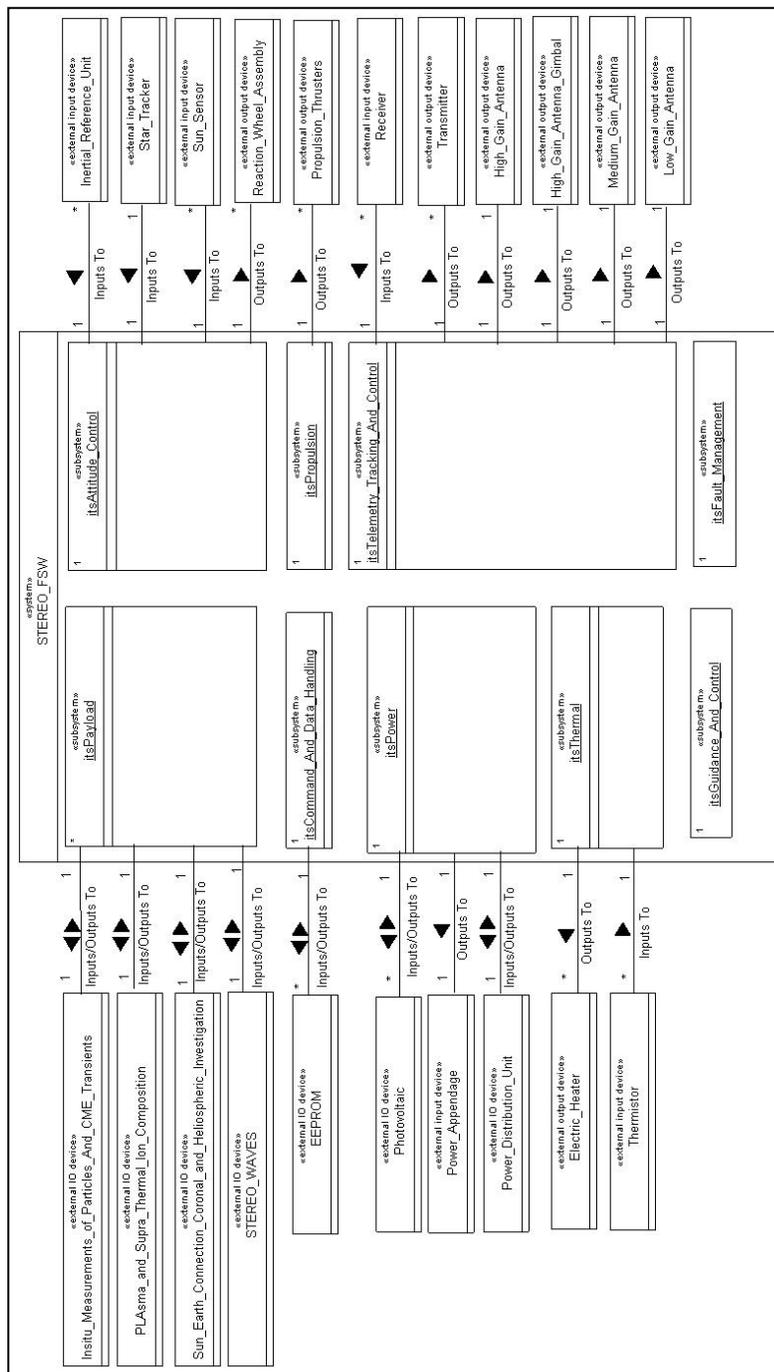


Figure D-10 STEREO subsystem allocation to physical devices

D.3 STEREO Analysis Modeling

After the requirements modeling phase is finished, the next phase is to perform the analysis modeling. The purpose of this phase to develop the basic structure for STEREO's FSW architecture. Since STEREO is a member of the FSW SPL, its analysis model is also derived from the FSW SPL analysis model described in Chapter 6.

D.3.1 STEREO Executable Design Pattern Selection

The first step in the analysis modeling is to select the design patterns that are appropriate for STEREO's features using the FSW SPL feature to design pattern mapping captured in Chapter 6.

Table D-2 is a subset of the FSW SPL feature to design pattern mapping that lists just the design patterns that correspond to STEREO's C&DH features.

Table D-2 STEREO Design Pattern Selection

Feature Group	Variability	Feature	Design Pattern
<<exactly-one-of feature group>> Command Execution	alternative	High Volume Command Execution	FSW Hierarchical Control
<<exactly-one-of feature group>> Telemetry Storage and Retrieval	alternative	High Volume Telemetry Storage and Retrieval	FSW Compound Commit
<<exactly-one-of feature group>>	alternative	Flexible High Volume Telemetry Formation	FSW Master Slave with Pipes and Filters &

Telemetry Formation			Strategy
<<exactly-one-of feature group>> Format Telemetry Reliability	alternative	Quick Check	FSW Sanity Check
<<at-least-one-of feature group>> Payload Data Collection	optional	Event Driven Payload Data Collection	FSW Payload Data Multicast
<<at-least-one-of feature group>> Housekeeping Data Collection	optional	Ground Driven Housekeeping Data Collection	FSW Housekeeping Data Client Server
	optional	Event Driven Housekeeping Data Collection	FSW Housekeeping Data Multicast
N/A	optional	Housekeeping Data Checks	FSW Housekeeping Checks Multicast
N/A	optional	Spacecraft Clock	FSW Spacecraft Clock Multicast
N/A	kernel	Memory Storage Device Fault Detection	FSW Memory Storage Device Watchdog

D.3.2 STEREO Hierarchical Control Architectural Design Pattern

After the FSW architectural and executable design patterns are selected, they must be customized to STEREO to create STEREO architectural and executable design patterns following the process outlined in Chapter 5. This process is briefly illustrated using the collaboration diagram for STEREO's Hierarchical Control Design Pattern, depicted in Figure D-11. It can be seen that this collaboration diagram contains the STEREO specific components and multiplicities based on STEREO's feature selection.

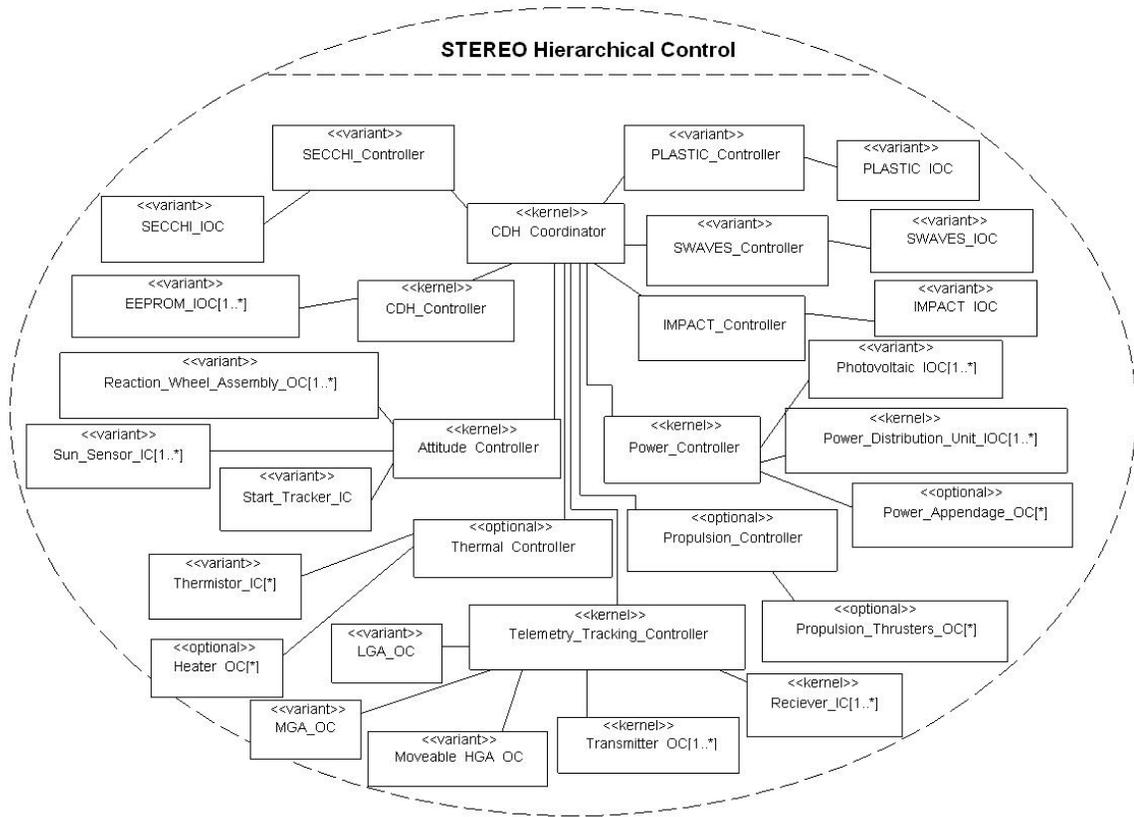


Figure D-11 STEREO Hierarchical Control collaboration diagram

D.3.3 STEREO Design Pattern Interconnection

The next step is to customize the design pattern interconnections for STEREO. This involves customizing the interaction overview diagrams based on the application's selected features. Any design patterns that are not used based on the selected features should be removed from the interaction overview diagram. Additionally, the associated ports and connectors should also be removed.

First, the Execute Commands interaction overview begins with a feature based condition on the Command Execution pattern specific feature group. STEREO selected to use the High Volume Command Execution feature, thus only the path involving this feature is selected. Therefore the other paths are removed for STEREO. The final interaction overview diagram STEREO's Execute Commands is depicted in Figure D-12.

Additionally, the ports and connectors between the components in the FSW Spacecraft Clock executable design pattern and the FSW Hierarchical Control executable design pattern are removed.

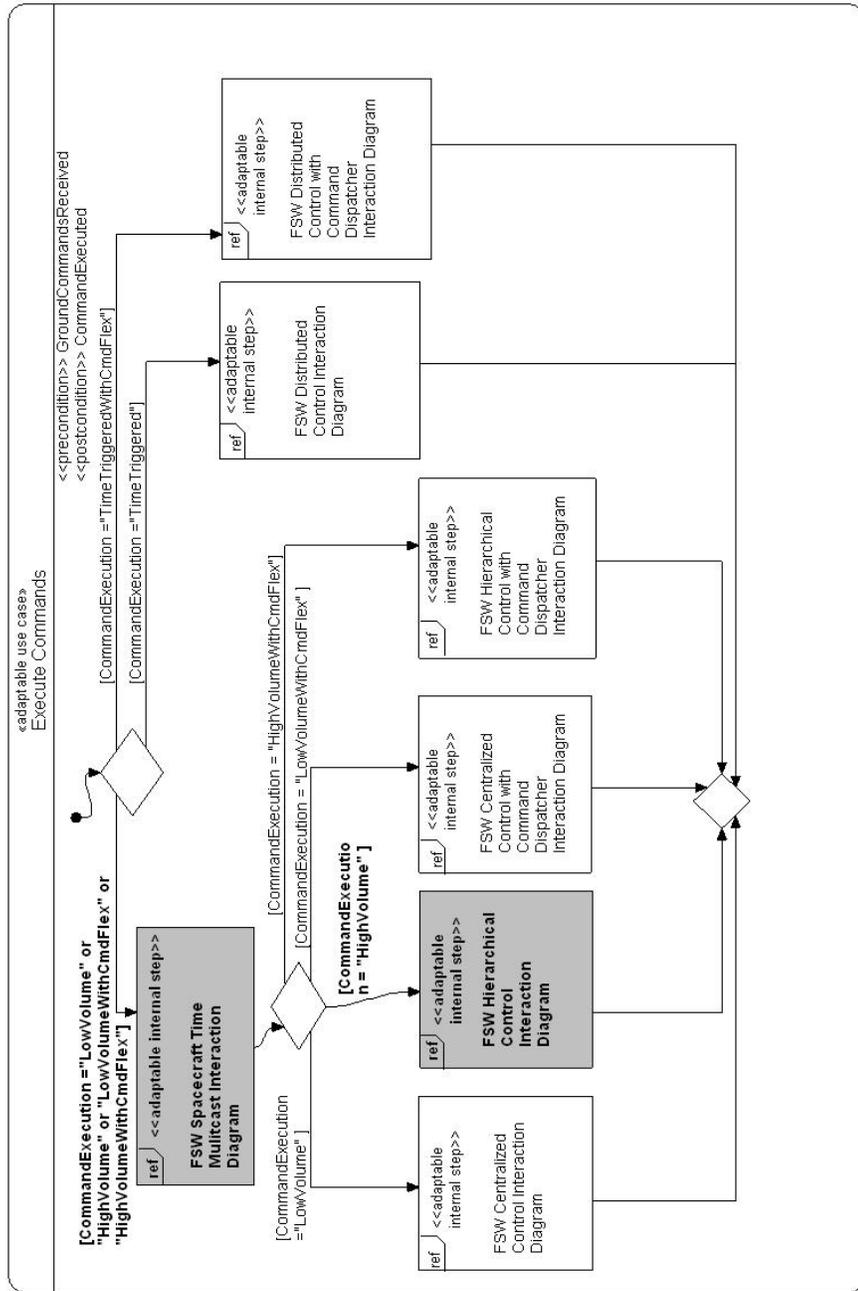


Figure D-12 STEREO's command execution interaction overview diagram

D.3.4 STEREO Design Pattern within Layered Architecture

The last step in the analysis modeling phase is to update the Layered Architecture to include just the STEREO specific components and variants. As discussed in Chapter 5, this involves removing any optional components that are not realized based on the STEREO's feature selection and using just the STEREO specific variants. A view of the STEREO C&DH subsystem architecture with the Hierarchical Control components is depicted in Figure D-13.

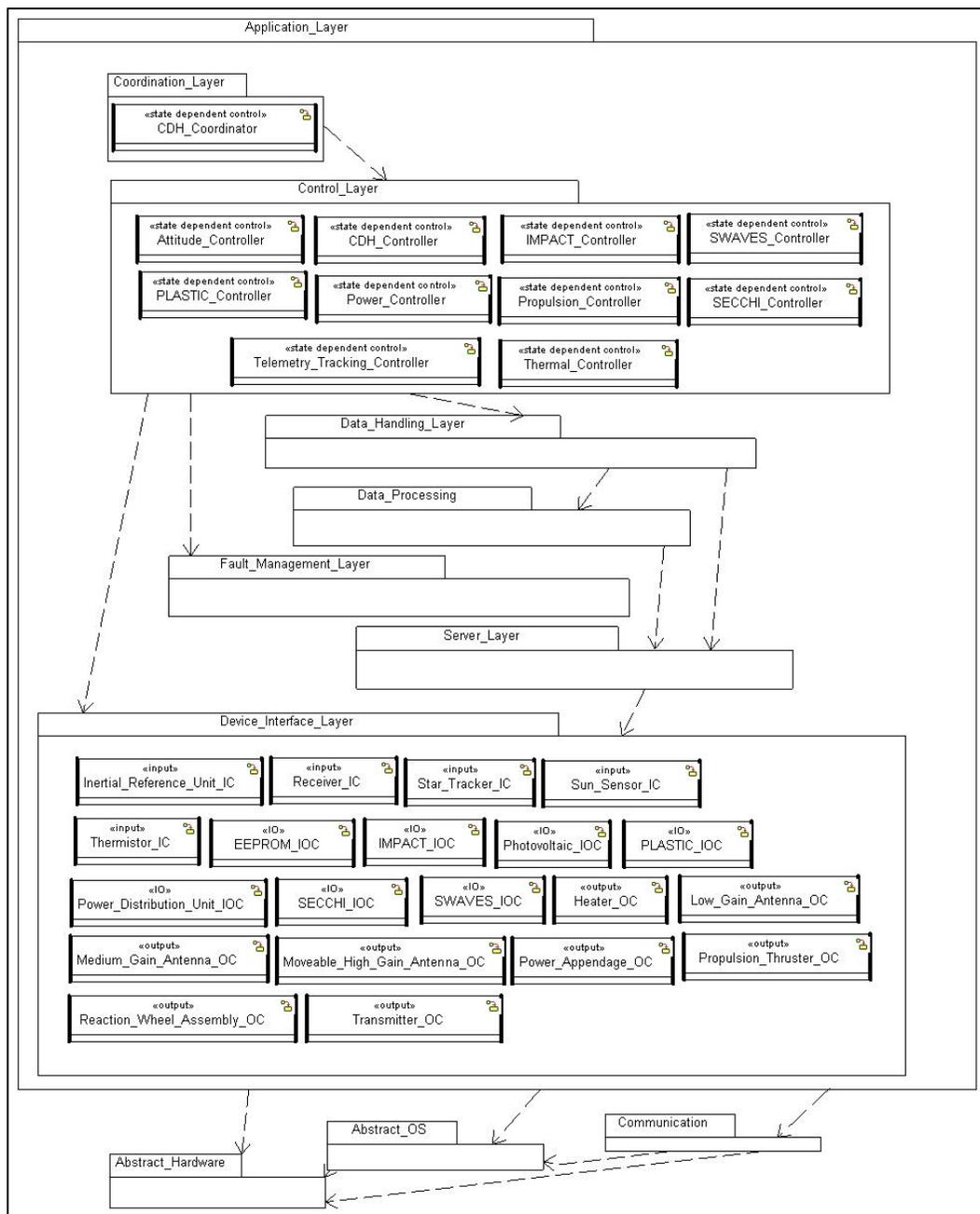


Figure D-13 STEREO Architecture Layered View with Hierarchical Control Components

REFERENCES

REFERENCES

- Albin-Amiot, H.; et al., 2001. Instantiating and detecting design patterns: putting bits and pieces together. In *16th Annual International Conference on Automated Software Engineering, 2001. (ASE 2001)*. IEEE.
- Alencar, P. et al., 1999. A pattern-based approach to structural design composition. In *Computer Software and Applications Conference (COMPSAC '99)*. Phoenix, AZ, USA: IEEE.
- Allen, R. & Garlan, D., 1997. A Formal Basis For Architectural Connection. *ACM Transactions on Software Engineering and Methodology*.
- Allen, R.J., 1997. *A Formal Approach to Software Architecture*. Carnegie Mellon University. Available at: <http://reports-archive.adm.cs.cmu.edu/anon/1997/CMU-CS-97-144.pdf>.
- Alphonse, C., Caspersen, M. & Decker, A., 2007. Killer “Killer Examples” for Design Patterns. In *SIGCSE '07*. Covington, Kentucky, USA: ACM.
- Al-Tahat, K.S.Y. et al., 2006. A Design Pattern Management Tool for Educational Purposes. In *The 2nd Information and Communication Technologies (ICTTA)*. IEEE.
- Anon, 2007. *Science Operations Manual and Instrument Users Manual for the Sun Earth Connection Coronal and Heliospheric Investigation (SECCHI)*, Naval Research Laboratory E.O. Hulburt Center For Space ResearchH. Available at: <http://secchi.nrl.navy.mil/wiki/pmwiki.php?n=Main.FlightSoftware>.
- Anon, Software Product Lines | Case Studies. Available at: <http://www.sei.cmu.edu/productlines/casestudies/> [Accessed April 21, 2011a].
- Anon, Stereo-Waves Instrumentation. Available at: http://swaves.gsfc.nasa.gov/swaves_instr.html [Accessed February 24, 2011b].

- Arcelli, F., Masiero, S. & Raibulet, C., 2005. Elemental Design Patterns Recognition In Java. In *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*. IEEE.
- Arcelli, F., Masiero, S., Raibulet, C., et al., 2005. A Comparison of Reverse Engineering Tools based on Design Pattern Decomposition. In *2005 Australian Software Engineering Conference (ASWEC'05)*. IEEE.
- Aversano, L., Canfora, G., et al., 2007. An Empirical Study on the Evolution of Design Patterns. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*. Cavtat near Dubrovnik, Croatia: ACM.
- Aversano, L., Cerulo, L. & Di Penta, M., 2007. Relating the Evolution of Design Patterns and Crosscutting Concerns. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE.
- Balanyi, Z. & Ferenc, R., 2003. Mining Design Patterns from C++ Source Code. In *International Conference on Software Maintenance (ICSM'03)*. IEEE.
- Baniassad, E.L.A., Murphy, G.C. & Schwanninger, C., 2003. Design Pattern Rationale Graphs: Linking Design to Source. In *25th International Conference on Software Engineering (ICSE 03)*. IEEE.
- Bass, L., Clements, P. & Kazman, R., 2003. *Software Architecture in Practice*, Addison-Wesley Professional.
- Basu, N., Chatterjee, S. & Chaki, N., 2005. Design Pattern Mining from Source Code for Reverse Engineering. In *TENCON 2005*. IEEE.
- Bayley, I. & Zhu, H., 2007. Formalising Design Patterns in Predicate Logic. In *Fifth IEEE International Conference on Software Engineering and Formal Methods*. IEEE.
- Bayley, I. & Zhu, H., 2008. On the Composition of Design Patterns. In *The Eighth International Conference on Quality Software (QSIC '08)*.
- Bellebia, D. & Douin, J.-M., 2006. Applying patterns to build a lightweight middleware for embedded systems. In *2006 conference on Pattern languages of programs*. Portland, Oregon, USA.
- Bennett, M. et al., 2008. An Architectural Pattern for Goal-Based Control. In *IEEE Aerospace Conference*. IEEE Computer Society.

- Bennett, M. et al., 2005. State-Based Models for Planning and Execution. In *15th International Conference on Planning and Scheduling (ICAPS 2005)*. Jet Propulsion Laboratory, National Aeronautics and Space Administration.
- Benowitz, Ed, Clark, K. & Watney, G., 2006. Auto-coding UML Statecharts for Flight Software. In *Proceedings of the 2nd IEEE International Conference on Space Mission Challenges for Information Technology*. IEEE Computer Society.
- Benowitz, E.G. & Niessner, A.E., 2003. A Patterns Catalog for RTSJ Software Designs. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*.
- Bieman, J.M. et al., 2003. Design Patterns and Change Proneness: An Examination of Five Evolving Systems. In *Ninth International Software Metrics Symposium (METRICS'03)*. IEEE.
- Briand, L.C., Labiche, Y. & Sauvé, A., 2006. Guiding the Application of Design Patterns Based on UML Models. In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*. IEEE.
- Bulka, A., 2002. Design Pattern Automation. In 2002 conference on Pattern languages of programs (CRPIT '02). Melbourne, Australia: Australian Computer Society, Inc.
- Buschmann, F., Henney, K. & Schmidt, D.C., 2007. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*, Hoboken, NJ: John Wiley & Sons, LTD.
- Buschmann, F. et al., 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Hoboken, NJ: John Wiley & Sons, LTD.
- Cinneide, M.O. & Nixon, P., 2001. Automated Software Evolution Towards Design Patterns. In *IWPSE*. Vienna Austria: ACM.
- Clements, P. & Northrop, L., 2002. *Software Product Lines: Practices and Patterns*, Addison-Wesley Professional.
- Clements, P. et al., 2002. *Documenting Software Architectures: Views and Beyond*, Addison-Wesley Professional.
- Cornils, A. & Hedin, G., 2000. Statically Checked Documentation with Design Patterns. In *33rd International Conference on Technology of Object-Oriented Languages (TOOLS)*.

- Costagliola, G. et al., 2005. Design Pattern Recovery by Visual Language Parsing. In *Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*. IEEE.
- Dascalu, S., Hao, N. & Debnath, N., 2005. Design Patterns Automation with Template Library. In *IEEE International Symposium on Signal Processing and Information Technology*. IEEE.
- Dasiewicz, P., 2005. Design Patterns and Object-Oriented Software Testing. In *CCECE/CCGEI*. Saskatoon: IEEE.
- Dong, J. & Zhao, Y., 2007. Experiments on Design Pattern Discovery. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07)*. IEEE.
- Dong, J., Lad, D.S. & Zhao, Y., 2007. DP-Miner: Design Pattern Discovery Using Matrix. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. IEEE.
- Dong, J., Yang, S. & Zhang, K., 2006. A Model Transformation Approach for Design Pattern Evolutions. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*. IEEE.
- Dong, J., Yang, S. & Zhang, K., 2005. VisDP: A Web Service for Visualizing Design Patterns on Demand. In *International Conference on Information Technology: Coding and Computing (ITCC'05)*. IEEE Computer Society.
- Douglass, B., 2003. *Real-Time Design Patterns*, Addison-Wesley.
- Dupire, B. & Fernandez, E.B., 2001. The Command Dispatcher Pattern. In 8th Conference on Pattern Languages of Programs. Monticello, Illinois, USA.
- Dvorak (editor), D., 2009. *NASA Study on Flight Software Complexity*, NASA Office of Chief Engineer. Available at:
http://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf.
- El Boussaidi, G. & Mili, H., 2007. A model-driven framework for representing and applying design patterns. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*. IEEE.
- Esmailzadeh, H. et al., 2007. NNEP, Design Pattern for Neural-Network-Based Embedded Systems. In *MIXDES 2007*. Ciechocinek, POLAND: Department of Microelectronics & Computer Science, Technical University of Lodz.

- Ferenc, R. et al., 2005. Design Pattern Mining Enhanced by Machine Learning. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE.
- Fliege, I. et al., 2005. Developing safety-critical real-time systems with SDL design patterns and components. *Computer Networks*, 49.
- Galvin, A. et al., 2008. The Plasma and Suprathermal Ion Composition (PLASTIC) Investigation on the STEREO Observatories. *Space Science Reviews*, 136(1), pp.437-486.
- Gamma, E. et al., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series.
- Garlan, D., Monroe, R.T. & Wile, D., 1997. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*. CASCON'97. Toronto, Ontario.
- Gay, D., Levis, P. & Culler, D., 2007. Software Design Patterns for TinyOS. *ACM Transactions on Embedded Computing Systems*, 6, Number 4.
- Gestwicki, P.V., 2007. Computer Games as Motivation for Design Patterns. In *SIGCSE'07*. Covington, Kentucky, USA: ACM.
- Goddard Space Flight Center, 2009. *Core Flight Software System: Reducing Flight Software Development Costs*, National Aeronautics and Space Administration Goddard Space Flight Center. Available at: gsfctechnology.gsfc.nasa.gov/TechSheets/CFSS_Goddard_09.pdf [Accessed October 15, 2010].
- Goddard Space Flight Center, 2010. Driving Down Mission Costs : New Flight Software Package Delivered to Lunar Mission. *Driving Down Mission Costs New Flight Software Package Delivered to Lunar Mission*. Available at: <http://gsfctechnology.gsfc.nasa.gov/MissionCost.html> [Accessed October 15, 2010].
- Gomaa, H., 2000. *Designing Concurrent, Distributed, and Real-Time Applications with UML Third.*, Boston: Addison-Wesley Object Technology Series.
- Gomaa, H., 2005. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison-Wesley Object Technology Series,.

- Gomaa, H., 2011. *Software Modeling and Design: UML, Use Cases, Architecture, and Patterns*, Cambridge University Press.
- Gomaa, H. & Olimpiew, E.M., 2010. Multiple View Requirement Models for Software Product Line Engineering. In K. C. Chang, V. Sugumaran, & S. Park, eds. *Applied Software Product Line Engineering*. CRC Press.
- Gomaa, H. & Shin, M.E., 2004. Multiple-View Meta-Modeling of Software Product Lines. In *Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'02)*. Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'02). IEEE Computer Society Press.
- Graf, S., 2004. UML based modeling of real-time and embedded systems and formal validation of timed systems with the IF environment. In *International Symposium on Formal Methods for Components and Objects (FMCO)*. Leiden, The Netherlands.
- Gueheneuc, Yam-Gael & Albin-Amioti, H., 2001. Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-class Design Defects. In *39th International Conference and Technology of Object-Oriented Languages and Systems (TOOLS)*. IEEE.
- Gwaltney, D.A. & Briscoe, J.M., 2006. *Comparison of Communication Architectures for Spacecraft Modular Avionics Systems*, NASA Marshall Space Flight Center. Available at: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20060050129_2006251309.pdf.
- Gwaltney, D.A. & Briscoe, J.M., 2005. The Integrated Safety-Critical Advanced Avionics Communication & Control (ISAACC) System Concept: Infrastructure for ISHM. In *Integrated Systems Health Management Conference*. Integrated Systems Health Management Conference. Ohio.
- Hakansson, J. et al., 2004. An Analysis Tool for UML Models with SPT Annotations. In *Workshop on SVERTS: Specification and Validation of UML models for Real Time and Embedded Systems*. Lisbon, Portugal.
- Harel, D., 1997. Executable object modeling with statecharts. In *18th International Conference on Software Engineering*. 18th International Conference on Software Engineering. IEEE Computer Society.

- Hecht, M. & Buettner, D., 2005. Software Testing in Space Programs. *Crosslink*, 6(Number 3). Available at:
<http://www.aero.org/publications/crosslink/fall2005/06.html>.
- Herrmann, A. & Schöning, T., 2000. Standard Telemetry Processing – an object oriented approach using Software Design Patterns. *Aerospace Science and Technology*, 4(4), pp.289-297.
- Howard, R. et al., 2008. Sun Earth Connection Coronal and Heliospheric Investigation (SECCHI). *Space Science Reviews*, 136(1), pp.67-115.
- Hsueh, N.-L. et al., 2007. A Quality Verification Model for Design Pattern. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*. IEEE.
- IBM Corporation, 2009. *Rational Rhapsody User Guide*, IBM Corporation. Available at:
<http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/UserGuide.pdf>.
- ISO/IEEE, 2011. *Systems and software engineering — Architecture description (DRAFT)*, ISO/IEEE. Available at: <http://www.iso-architecture.org/ieee-1471/docs/ISO-IEC-IEEE-latest-draft-42010.pdf>.
- Izurieta, C. & Bieman, J., 2007. How Software Designs Decay: A Pilot Study of Pattern Evolution. In *First International Symposium on Empirical Software Engineering and Measurement*. IEEE.
- Jalil, M., Noah, S.A. & Idris, S., 2010. Evaluating the Effectiveness of a Pattern Application Support Tool for Novices. In *15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2010)*. Bilkent, Ankara, Turkey: ACM.
- Jeon, S.-U., Lee, J.-S. & Bae, D.-H., 2002. An Automated Refactoring Approach to Design Pattern Based Program Transformations in Java Programming. In *9th Asia Pacific Software Engineering Conference (APSEC 02)*. IEEE.
- John Galloway, 1998. STEREO Mission Design. Available at:
http://stereo.nrl.navy.mil/orig_stereo/stereo_galloway.pdf [Accessed April 26, 2010].

- Johns Hopkins University Applied Physics Laboratory, JHU/APL Status Reports/Viewgraphs. Available at: http://stereo.nrl.navy.mil/orig_stereo/pre-phaseA.html [Accessed April 26, 2010a].
- Johns Hopkins University Applied Physics Laboratory, STEREO Web Site. Available at: <http://stereo.jhuapl.edu/index.php> [Accessed April 26, 2010b].
- Kaczor, O., Gueheneuc, Yann-Gael & Hamel, S., 2006. Efficient Identification of Design Patterns with Bit-vector Algorithm. In *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE.
- Kalinsky, D., 2002. Design Patterns for High Availability. *Embedded Systems Programming*.
- Katwijk, J. van, Schwarz, J.-J. & Zalewski, J., 2001. PRACTICE OF REAL-TIME SOFTWARE ARCHITECTURES: COLLIDER, SATELLITES AND TANKS COMBINED. In IFAC Conference on New Technologies for Computer Control. Hong Kong.
- Kim, D.-K. & Lu, L., 2006. Inference of Design Pattern Instances in UML models via Logic Programming. In *11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)*. IEEE.
- Kircher, M. & Jain, P., 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*, Hoboken, NJ: John Wiley & Sons, LTD.
- Laboratory For Atmospheric and Space Physics at the University of Colorado at Boulder, Student Nitric Oxide Explorer Homepage. <http://lasp.colorado.edu/snoe/>. Available at: <http://lasp.colorado.edu/snoe/> [Accessed April 21, 2010].
- Landauer, C., 2000. Wrappings as Design Patterns. In *33rd Hawaii International Conference on System Sciences*. IEEE.
- Lee, H., Youn, H. & Lee, E., 2007c. Automatic Detection of Design Pattern for Reverse Engineering. In *Fifth International Conference on Software Engineering Research, Management and Applications*. IEEE.
- Luhmann, J. et al., 2008. STEREO IMPACT Investigation Goals, Measurements, and Data Products Overview. *Space Science Reviews*, 136(1), pp.117-184.
- M.L. Kaiser et al., 2008. The STEREO Mission: An Introduction. *Space Science Reviews*, 136, pp.5-16.

- Magee, J. et al., 1995. Specifying Distributed Software Architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC 95)*. Sitges, Spain.
- Mark A. Salada & Randal Davis, 1995. SNOE Telemetry and Flight Software Design. *American Institute of Aeronautics and Astronautics*.
- Mark A. Salada et al., 1998. Software Lessons from the University of Colorado's Student Nitric Oxide Explorer. *American Institute of Aeronautics and Astronautics*.
- Masuda, G., Sakamoto, N. & Ushijima, K., 2000. Redesigning of an Existing Software using Design Patterns. In *International Symposium on Principles of Software Evolution*. IEEE.
- Meffert, K., 2006. Supporting Design Patterns with Annotations. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*. IEEE.
- Mens, T. & Tourwe, T., 2001. A Declarative Evolution Framework for Object-Oriented Design Patterns. In *IEEE International Conference on Software Maintenance*. IEEE.
- NASA Goddard Space Flight Center, NASA - STEREO. Available at: http://www.nasa.gov/mission_pages/stereo/main/index.html [Accessed April 26, 2010].
- NASA Jet Propulsion Laboratory, 2011. Basics of Space Flight. Available at: <http://www2.jpl.nasa.gov/basics/index.php> [Accessed February 11, 2011].
- NASA Office of Chief Engineer, NASA - NASA Study on Flight Software Complexity. Available at: http://www.nasa.gov/offices/oce/documents/FSWC_study.html [Accessed October 26, 2010].
- Ng, T.H. et al., 2007. Do Maintainers Utilize Deployed Design Patterns Effectively? In *29th International Conference on Software Engineering (ICSE '07)*. IEEE.
- Noble, J., 1998. Classifying Relationships Between Object-Oriented Design Patterns. In *Australian Software Engineering Conference*. IEEE.
- Nuseibeh, B., Kramer, Jeff & Finkelstein, A., 1993. Expressing the relationships between multiple views in requirements specification. In *Proceedings of the 15th international conference on Software Engineering*. ICSE '93. IEEE Computer Society Press.

- O'Hara-Schettino, E. & Gomaa, H., 1998. Dynamic navigation in multiple view software specifications and designs. *Journal of Systems and Software*, 41(2), pp.93-103.
- Ohtsuki, M., Yoshida, N. & Makinouchi, A., 1999. A Source Code Generation Support System Using Design Pattern Documents Based on SGML. In *Sixth Asia Pacific Software Engineering Conference*. IEEE.
- Olimpiew, E.M., 2008. *Model-based testing for software product lines*. George Mason University. Available at:
http://digilib.gmu.edu:8080/bitstream/1920/3039/1/Olimpiew_Erika.pdf.
- Olimpiew, E.M. & Gomaa, H., 2009. Reusable Model-Based Testing. In *11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering*. 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering. Springer-Verlag Berlin, Heidelberg.
- OMG, 2005. *UML 2.0 Specification*, Object Management Group (OMG) Inc. Available at: <http://www.omg.org/spec/UML/2.0/> [Accessed October 19, 2008].
- Ouyang, Y., 2002. Explaining design patterns through one application. In *32nd Annual Frontiers in Education*. IEEE.
- Pappalardo, G. & Tramontana, E., 2006. Automatically Discovering Design Patterns and Assessing Concern Separations for Applications. In *Symposium on Applied Computing (SAC)*. Dijon, France: ACM.
- Petriu, D.C., 2005. Performance Analysis with the SPT Profile. In *Model-Driven Engineering for Distributed and Embedded Systems*. London, England: Hermes Science Publishing Ltd, pp. 205-224.
- Pettit IV, R. & Gomaa, H., 2007. Analyzing Behavior of Concurrent Software Designs for Embedded Systems. In *ISORC*. IEEE.
- Pohl, K., Böckle, G. & Linden, F. van der, 2005. *Software Product Line Engineering Foundations, Principles, and Techniques*, Springer.
- Pont, M.J. & Banner, M.P., 2004. Designing embedded systems using patterns: A case study. *The Journal of Systems and Software*, 71.
- Prechelt, L. et al., 2001. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions On Software Engineering*, 27(12).

- Prechelt, L. et al., 2002. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Transactions On Software Engineering*, 28(6).
- Ram, D.J., Reddy, P.J.K. & Rajasree, M.S., 2004. Pattern Hybridization: Breeding New Designs Out of Pattern Interactions. *ACM Software Engineering Notes*, 29 Number 4.
- Riehle, D., 1997. Composite design patterns. In *12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Atlanta, Georgia, United States: ACM.
- Rumbaugh, J., Jacobson, I. & Booch, G., 2004. *The Unified Modeling Language Reference Manual Second.*, Boston: Addison-Wesley.
- Schmidt, D. et al., 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Hoboken, NJ: John Wiley & Sons, LTD.
- Schulz, B. et al., 1998. On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems. In *Technology of Object-Oriented Languages and Systems (TOOLS)*. IEEE.
- Scott Bailey et al., 1996. Science Instrumentation for the Student Nitric Oxide Explorer. In *Optical Spectroscopic Techniques and Instrumentation for Atmospheric and Space Research II (Proceedings Volume)*. SPIE Press.
- Selic, B., 1996a. An Architectural Pattern for Real-Time Control Software. In *Pattern Languages of Program Design 2*. Addison-Wesley.
- Selic, B., 2004. Architectural Patterns for Real-Time Systems: Using UML as an Architectural Description Language. In *UML for Real*. Springer, pp. 171-188.
- Selic, B., 1996b. Recursive Control: An Architectural Pattern for Large Real-Time Software. In *International Conference on Pattern Languages of Programming (PLoP)*. Addison-Wesley.
- Shaw, M. & Garlan, D., 1996a. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.
- Shaw, M. & Garlan, D., 1996b. *Software Architecture: Perspectives on an Emerging Discipline*, Upper Saddle River, NJ: Prentice Hall.

- Shi, N. & Olsson, R.A., 2006. Reverse Engineering of Design Patterns from Java Source Code. In *21st IEEE International Conference on Automated Software Engineering (ASE'06)*. IEEE.
- Singh, P.B. & Chaudhary, B.D., 2009a. A formal model of design-patterns based design. In 2nd India software engineering conference.
- Singh, P.B. & Chaudhary, B.D., 2009b. Structural Formalization of Design-Pattern Based Software Design. In 2009 Third International Conference on Digital Society. IEEE Computer Society.
- Smith, C. & Williams, L., 2002. *Performance Solutions A Practical Guide to Creating Responsive, Scalable Software*, Boston: Addison-Wesley.
- Stanley Solomon et al., 1996. The Student Nitric Oxide Explorer. In *Space Sciencecraft Control and Tracking in the New Millennium*. SPIE Press.
- Stanley Solomon et al., 1998. The SNOE Spacecraft: Integration, Test, Launch, Operation, and On-orbit Performance. *American Institute of Aeronautics and Astronautics*.
- Street, J. & Gomaa, H., 2006. An Approach to Performance Modeling of Software Product Lines. In *9th International Conference on Model Driven Engineering Languages and Systems' Modeling and Analysis of Real-Time and Embedded Systems (MARTES) Workshop*. Genova.
- Strom, S.R., 2002. Charting a Course Toward Global Navigation. *Crosslink*, 3(2). Available at: <http://www.aero.org/publications/crosslink/summer2002/01.html>.
- Tahvildari, L. & Kontogiannis, K., 2002a. A Software Transformation Framework for Quality-Driven Object-Oriented Re-engineering. In *International Conference on Software Maintenance (ICSM'02)*. IEEE.
- Tahvildari, L. & Kontogiannis, K., 2002b. On the Role of Design Patterns in Quality-Driven Re-engineering. In *Sixth European Conference on Software Maintenance and Reengineering (CSMR 02)*. IEEE.
- Tawhid, R. & Petriu, D., 2011. Product Model Derivation by Model Transformation in Software Product Lines. In *MoBE-RTES Workshop*. Newport Beach, California: IEEE Computer Society.
- Tawhid, R. & Petriu, D., 2008. Towards automatic derivation of a product performance model from a UML software product line model. In *Proceedings of the 7th*

- international workshop on Software and performance*. WOSP '08. New York, NY, USA: ACM.
- Taylor, R.N., Medvidovic, N. & Dashofy, E.M., 2009. *Software Architecture: Foundations, Theory, and Practice*, Wiley.
- Tonella, P. & Antoniol, G., 1999. Object Oriented Design Pattern Inference. In *IEEE International Conference on Software Maintenance*. IEEE.
- Tsai, W.-T. et al., 1999. Testing Extensible Design Patterns in Object-Oriented Frameworks through Scenario Templates. In *The Twenty-Third Computer Software and Applications Conference (COMPSAC)*. IEEE.
- TTech, *NASA's Orion Crew Exploration Vehicle Systems Integration with TTEthernet*, Available at: <http://www.ttech.com/fileadmin/content/pdf/TTTech-NASA-Casestudy-Orion.pdf>.
- Wagstaff, K.L. et al., 2008. Automatic Code Generation For Instrument Flight Software. In *9th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*. 9th International Symposium on Artificial Intelligence, Robotics, and Automation in Space.
- Walko, J., 2009. TTEch, NASA team on TTEthernet for space apps. *EETimes*. Available at: <http://www.eetimes.com/electronics-news/4195051/TTTech-NASA-team-on-TTEthernet-for-space-apps> [Accessed March 13, 2011].
- Wang, J., Song, Y.-T. & Chung, L., 2005d. From Software Architecture to Design Patterns: A Case Study of an NFR Approach. In *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks (SNPD/SAWN'05)*. IEEE.
- Wang, W. & Tzerpos, V., 2005. Design Pattern Detection in Eiffel Systems. In *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE.
- Wilmot, J., 2005. A core flight software system. In 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis. Jersey City, NJ, USA: ACM.
- Wilmot, J., 2006. Implications of ResponsiveSpace on the FlightSoftware Architecture. In 4th Responsive Space Conference. Los Angles, CA: AIAA.

- Woodside, M. et al., 2005. Performance by Unified Model Analysis (PUMA). In *Fifth International Workshop on Software and Performance (WOSP 05)*. Palma, Illes Balears, Spain.
- Wright, D.R., 2007. The Decision Pattern: Capturing and Communicating Design Intent. In *SIGDOC'07*. El Paso, Texas, USA: ACM.
- Xue-Bin, W. et al., 2007. Research and Implementation of Design Pattern-Oriented Model Transformation. In *International Multi-Conference on Computing in the Global Information Technology (ICCGI'07)*. IEEE.
- Yacoub, S.M., Xue, H. & Ammar, H.H., 2000. POD: a composition environment for pattern-oriented design. In *Technology of Object-Oriented Languages and Systems (TOOLS 34)*. Technology of Object-Oriented Languages and Systems (TOOLS 34). Santa Barbara, CA , USA: IEEE.
- Yacoub, S.M., Xue,, H. & Ammar, H.H., 2000. Automating the Development of Pattern-Oriented Designs for Application Specific Software Systems. In *3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'00)*. 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'00).
- Yuan, J., Miao, H. & Cai, L., 2004. A Design Pattern Verifier in Two-tier Programming Environment. In *Fourth International Conference on Computer and Information Technology (CIT'04)*. Fourth International Conference on Computer and Information Technology (CIT'04). IEEE.
- Zalewski, J., 1999. Real-Time Software Architectures and Design Patterns: Fundamental Concepts and Their Consequences. In *Annual Reviews in Control 2001*. pp. 133-146.
- Zalewski, J., 2001. Real-Time Software Architectures and Design Patterns: Fundamental Concepts and Their Consequences. *Annual Reviews in Control*, 25.
- Zalewski, J., 2002. *Real-Time Software Design Patterns*,
- Zhao, C., Kong, J. & Zhang, K., 2007. Design Pattern Evolution and Verification Using Graph Transformation. In *40th Hawaii International Conference on System Sciences*. IEEE.

CURRICULUM VITA

Julie Street Fant was born and raised in the District of Columbia metropolitan area. After graduating high school, she attended James Madison University and graduated in 2001 with a Bachelor of Science in Integrated Science and Technology and a minor in Computer Science. After college she began working in the software industry and attended graduate school. In 2004 she graduated from George Mason University with a Master of Science in Information Systems and received the Information and Software Engineering Department's 2004/2005 Academic Excellence Award in Information Systems.

Ms. Fant has 10 years of experience in the software industry. She is currently employed as an Engineering Specialist at The Aerospace Corporation. Her areas of expertise include software design, model-driven software analysis, UML, and applying research to industrial applications.

Publications

- “Building Domain Specific Software Architectures from Software Architectural Design Patterns,” Julie S. Fant, 33rd International Conference on Software Engineering (ICSE) ACM Student Research Competition, Honolulu, Hawaii USA, May 2011.
- “Architectural Design Patterns for Flight Software,” Julie S. Fant, Hassan Gomaa, and Robert G. Pettit *2nd IEEE Workshop on Model-based Engineering for Real-Time Embedded Systems*, Newport Beach, California, March 2011.
- “Modeling and Prototyping of Concurrent Software Architectural Designs with Colored Petri Nets”, Robert G. Pettit, Hassan Gomaa, and Julie S. Fant, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'09)*, Paris, France, 2009.

- "Cost-Performance Tradeoff Analysis for Embedded Systems," Julie S. Fant and Robert G. Pettit, *Proceedings of the 6th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2008)*, Capri, Italy, October 2008.
- "Software Architectural Reuse Issues in a Service-oriented Architectures (SOA)," Julie A. Street and Hassan Gomaa, *Proceedings Hawaii International Conference on System Sciences (HICSS 2008)*, Waikoloa, Big Island, Hawaii January 2008.
- "An Experimental Evaluation of Software Performance Modeling and Analysis Software Performance Assessments of UML Designs," Julie A. Street and Robert G. Pettit, *Proceedings of the International Conference on Software and Data Technologies (ICSOFT 2007)*, Barcelona, Spain July 2007.
- "Independent Model-Driven Software Performance Assessments of UML Designs," Julie A. Street, Robert G. Pettit and Hassan Gomaa, *Proceedings of the 10th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2007)*, Santorini, Greece May 2007.
- "An Approach to Performance Modeling of Software Product Lines," Julie A. Street and Hassan Gomaa, *Proceedings of the International Workshop on Modeling and Analysis of Real Time Embedded Systems (MARTES 2006)*, Genova, Italy, October 2006.
- "Lessons Learned Applying Performance Modeling Analysis Techniques," Julie A. Street and Robert G. Pettit IV, *Proceedings of the 9th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2006)*, Gyeongju, Korea, April 2006.
- "A Host-Based Approach to Network Attack Chaining Analysis," Paul Ammann, Joseph Pamula, Ronald Ritchey, and Julie Street, *Proceedings of the 2005 Annual Computer Security Applications Conference (ACSAC)*, Tucson, AZ, December 2005.
- "The Impact of UML 2.0 on Existing UML 1.4 Models," Julie A. Street and Robert G. Pettit IV, *Proceedings of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Conference (MoDELS 2005)*, Montego Bay, Jamaica, October 2005.
- "Lessons Learned in the Development of Critical Systems Using UML," Robert G. Pettit IV and Julie A. Street, *Proceedings of the 7th ACM/IEEE International Conference on the Unified Modeling Language (UML 2004) Conference*, Lisbon, Portugal, October 2004.