Development of A Software Architecture Method for Software Product Families and its Application to the AubieSat Satellite Program

by

John Ryan O'Farrell

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 18, 2009

Keywords: software architecture, product families, reuse

Approved by

Richard Chapman, Co-chair, Associate Professor of Computer Science and Software Engineering
John A. Hamilton, Jr., Co-chair, Professor of Computer Science and Software Engineering
Saad Biaz, Associate Professor of Computer Science and Software Engineering

Abstract

Software architecture methodologies are very useful in reusing software code from one product to another. In software product families, several similar products are developed by the same organization that share many features and are prime candidates for code reuse. Previous work on software architectures for product lines is leveraged to create a unique architecture methodology called Core Based Architecture for Product Families (CBAPF). This methodology is then applied to the AubieSat satellite program at Auburn University by modifying the design of AubieSat-1 and demonstrating how the methodology assists in reusing the core software in the ground station system and in a hypothetical AubieSat-2 mission. This case study demonstrates that CBAPF framework excels at providing a formal design that allows for code reuse in this satellite family.

Table of Contents

Abstract ii

List of Tables vi

List of Illustrations vii

Chapter 1: Introduction 1

Chapter 2: Literature Review 6

2.1: Current Software Architecture 6

2.1.1 UML Based Architecture Frameworks 7

2.1.2 Architecture Definition Language (ADL) Approaches 8

2.1.3 DODAF and Related Frameworks 9

2.1.4 Service Oriented Architectures 10

2.1.5 C2 Architectural Style 11

2.2 Product Family Specific Architectures 12

2.2.1 Component-Oriented Platform Architecting Method (COPA) 14

2.2.2 Family-Oriented Abstraction, Specification, and Translation (FAST) 15

2.2.3 Feature-Oriented Reuse Method (FORM) 16

2.2.4 KobrA 16

2.2.4 Quality-driven Architecture Design and quality Analysis (QADA) 17

2.3 Space and Satellite Software Architecture 17

2.4 Research Objectives 19

Chapter 3: Software Architecture Methodology Definition 21

3.1 A Core Based Architecture for Product Families 21
3.2 Requirements in CBAPF 24
3.3 Required Architectural Artifacts 25
3.3.1 Core Architecture Description 25
3.3.2 Core Connector Description 28
3.3.3 Product Specific Architecture Description 30
3.3.4 Combined Architecture Description 31
3.4 Field Specific Documentation in CBAPF 34
3.5 Detailed Documentation in CBAPF 35
3.6 Implementation of a CBAPF System 36
Chapter 4: AubieSat-1 Software Architecture 39
4.1 AubieSat-1 Command and Data Handling Software Requirements 40
4.2 Previous Software Architecture Design 42
4.3 AubieSat-1 CBAPF Documentation 43
4.3.1 AubieSat-1 Core Architecture Description 43
4.3.2 AubieSat-1 Core Connector Description 47
4.3.3 AubieSat-1 Product Specific Architecture Description 49
4.3.4 AubieSat-1 Combined Architecture Description 52
4.4 AubieSat-1 Detailed Documentation 55
4.5 AubieSat-1 Implementation 57
Chapter 5: AubieSat-1 Ground Station Software Architecture 59
5.1 AubieSat-1 Ground Station Product Specific Architecture Description 59
5.2 AubieSat-1 Ground Station Combined Architecture Description 60

Chapter 6: AubieSat-2 Software Architecture Design 62
6.1 AubieSat-2 Command and Data Handling Software Requirements 62
6.2 AubieSat-2 CBAPF Documentation 66
6.2.1 AubieSat-2 Core Architecture Description 66
6.2.2 AubieSat-2 Core Connector Description 70
6.2.3 AubieSat-1 Product Specific Architecture Description 71
6.2.4 AubieSat-1 Combined Architecture Description 73
6.3 AubieSat-2 Detailed Documentation and Implementation 76
Chapter 7: Conclusion 79
References 82
Appendix A: Original Software Architecture 86
Appendix B: AubieSat-1 Detailed Software Documentation 94
Appendix C: AubieSat-2 Detailed Software Documentation 101

List of Tables

Table 1: AubieSat-1 System Tasks 53

Table 2: AubieSat-1 Requirement to Component Traceability 55

Table 3: AubieSat-2 Requirement to Component Traceability 76

List of Illustrations

Illustration 1: Generic CBAPF Product Family Design 23

Illustration 2: Generic Core Architecture Description 27

Illustration 3: Generic Core Connector Description 29

Illustration 4: Generic Product Specific Architecture Description 31

Illustration 5: Generic Combined Architecture Description 33

Illustration 6: AubieSat-1 Requirements 41

Illustration 7: AubieSat-1 Core Architecture Description 44

Illustration 8: AubieSat-1 Core Connector Description 48

Illustration 9: AubieSat-1 Product Specific Architecture Description 50

Illustration 10: AubieSat-1 Combined Architecture Description 54

Illustration 11: AubieSat-1 File System Component Detailed Documentation 56

Illustration 12: Ground Station Product Specific Architecture Description 59

Illustration 13: Ground Station Combined Architecture Description 61

Illustration 14: AubieSat-2 Requirements 66

Illustration 15: AubieSat-2 Core Architecture Description 68

Illustration 16: AubieSat-2 Core Connector Description 70

Illustration 17: AubieSat-2 Product Specific Architecture Description 72

Illustration 18: AubieSat-2 Combined Architecture Description 75

Illustration 19: AubieSat-2 Solar Panel Component Detailed Documentation 77

Chapter 1: Introduction

Current software engineering practice is greatly concerned with the possibility of reusing software code in several different software projects. Much of the focus of software engineering techniques is on creating code that is flexible, modular, and easy to adapt and maintain such that it can be reused in different software products. Achieving this goal of creating reusable code is very difficult, and must be considered at each point in the software engineering lifecycle in order to properly design and implement code that can be effectively reused. The software architecture, high-level design, low-level design, and implementation must each carefully analyze how to design and implement the code in a way that will allow the greatest possible reuse.

There is a need for a software architecture methodology that is capable of capturing the relationship between the different software products in a software family [Clements and Weiderman 1998]. The major part of this research has been devoted to developing such a methodology, and then applying that method to a software product family to demonstrate it effectiveness. The method that has been defined is based on defining a core set of components in a group of related products that can then be reused in each product, and is therefore called the Core Based Architecture for Product Families (CBAPF) [O'Farrell and Hamilton 2009].

There are several software product families that could greatly benefit from the use of such a method to reuse software code in each product. One such family of software products is the AubieSat satellite program at Auburn University [Chapman, Wersinger, Wilson 2008]. This is a student program to develop and launch cube satellites designed by engineering and physics students at Auburn University. The program is currently in the development of the first satellite,

called AubieSat-1, and is set to launch in the near future. The program hopes to continue in the future to allow students develop more satellites with increasing complexity. One key to the success of such a program will be the ability to build on the work from previous projects to allow each new satellite to be designed and developed quickly by relying on the hardware and software that have been used successfully in previous missions. Apart from reducing the development time, it is also important to reuse code in order to ensure the success of the mission by using flight-tested software. In addition to developing a methodology, this methodology has been applied to the software architecture of the AubieSat-1 to define a reusable portion of code that can be used by students for future satellite missions. This development could also be used in several other universities that have existing cube satellite programs, or those wishing to start a new cube satellite program [Dabrowski 2005].

The software architecture phase of development can be particularly useful when creating reusable software. Software architecture is the description of the software in a system that bridges the gap between software requirements and high-level software design. Because it is the first step in design of a system, one of the best possibilities for reusing large sections of code is to identify a portion of the software system that may be fit for use elsewhere, and design this component in a manner that facilitates is usage in other software. The current methods of software architecture vary widely in their approach, from strict guidelines of how to create documentation such as the Department of Defense Architecture Framework [DOD 2004], to much more free-form design techniques that involve general usage of the Unified Modeling Language (UML) or Architecture Definition Languages (ADLs) with little to no guidelines for how to perform the actual architecture activities [Clements 1996]. These methods each have advantages and disadvantages with respect to the level of activity required to produce the

documentation, support of reusable designs, and support from tools and the general software engineering community.

With regards to software architecture, the area of software product families is one that can greatly benefit from designing from reusable software components at a high level. Software product families, or software product lines, involve a series of related software products or projects that share much of the same functionality. These products could be embedded systems such as printers, cameras, or satellite systems, but it could also be traditional desktop applications, such as an office application suite, a family of operating systems (Windows Vista has several similar versions), or any other group of software systems that share a common subset of requirements and functionality. Any such group of systems would greatly benefit from using software architecture to identify the set of common software that is needed to support the common requirements for all of the products in the group. Using software architecture, it would then be possible to create a design that allowed the easy reuse of certain software components throughout each of the different products in the family. It is important to use software architecture to achieve this goal because it describes the overall structure of the software system, and therefore will make it easy to separate the portions that are shared and those that are specific to each individual family member. It is also important to develop some type of formal method for performing this identification to make sure both that the most possible reuse can be achieved and that verification and validation can be performed both on the shared portion of software and also on each product individually to be sure that it meets the full requirements of the product through a combination of the shared and product specific software components.

While some work has been done in developing software architecture approaches that can be used specifically for product families, certain improvements can be made to make the process

easier to use and to fit all of the requirements for the method. The work done in this area to date shows the promise of leveraging software architecture in the area of a software product groups [Jazayeri, Ran, and Linden 2000]. The novel methodology described here learns from the research already available in this area and improves upon it by developing a new approach to developing the software architecture of such a system that takes into account the software process challenges associated with such programs and also enhances the technical capability of the views to support high reusability and maintainability in such a system.

By focusing on applying this novel software architecture methodology to the AubieSat system, the method developed has been tuned in a manner that would be impossible without its use in an application. The problem to be solved in this case is to develop a method that will be usable by senior level and graduate students in developing the software for the satellite. In order to meet this requirement, the method will have to be easy to learn and use quickly. The method should provide enough guidance for the students to apply sound engineering principles that allow for reuse without incurring so much overhead as to impede progress or deter its usage. It will also need to use current modeling techniques and tools with which the students will be familiar. Because of these constraints, the methodology will be forced to use standard modeling techniques that will assist both its effectiveness and its acceptance to a wider audience. By designing the methodology for the use of software engineering students, it is hoped that practicing software engineers could also easily use the method.

From a software viewpoint, it is often very difficult to reuse code from one application to another. If the software in question is not originally designed to be reused, there must often be major modifications to that code before it can be used in another system, sometimes so much that the rework of this code requires more effort than developing a new piece of software to perform

the same function. Either of these cases is unacceptable for the AubieSat program. In order to allow for code reuse from one satellite to the next, there must be some high level view of how the code operates, how components communicate and behave, and how to separate the code logically so that the needed portions can be reused in future projects. The same constraints that apply to the AubieSat program also apply in general to the software engineering community, in which there is much pressure to deliver high quality, proven software products quickly. The software architecture method that has been developed will be used to meet the requirements of the AubieSat program, and, in doing so, will prove its general effectiveness in any software product family domain.

Chapter 2: Literature Review

Software architecture has in recent years become recognized as an important discipline and component in the software lifecycle [Hofmeister, Nord, and Soni 2000]. The development and implementation of software architecture has been widely studied, including techniques of how to go about creating an architecture, processes of how an architecture can be followed, and languages for representing the software architecture [Garlan 2000]. One of the benefits of focusing on software architecture early in the software development lifecycle is the ability to design for such desirable characteristics as reusability, modularity, and correctness. Several different approaches to developing software architecture will be examined along with how they are able to effectively define a common architecture for a suite of related products.

2.1 CURRENT SOFTWARE ARCHITECTURE METHODS

Several methodologies exist for representing a software architecture. This section will examine current software architecture frameworks that could be used in the domain of representing a reusable core architecture among a group of closely related systems. Many current software projects use informal methods for defining software architecture, such as general "box-and-line" diagrams. It is important, however to use a more formal methodology for describing the software architecture in order to correctly and completely specify all facets of the architecture and design of the software system [Abowd, Allen, Garland 1995]. The methods mentioned here are examples of these more formal methods.

2.1.1 UML based architecture frameworks

A common approach to developing a software architecture is to use a framework that is based on the Object Management Group (OMG) Unified Modeling Language (UML). UML provides several diagrams as part of its specification that can be used to represent the design of a software architecture. These approaches generally do not provide many architectural standards, as they typically rely only on conformance to the UML diagrams as criteria for a successful design. In this situation, the responsibility for creating a good design falls on the architectural team with little assistance from the framework. This is a problem when creating an ideal methodology, as the framework itself should provide assistance to the software developer that will promote the reusability that will be needed for the system to be designed and created effectively. UML itself is only a modeling tool, and does not qualify as a full architecture framework, but it is often used as one without any accompanying framework [Medvidovic et al. 2002]. This, not the usage of UML itself, is what causes issues when UML is chosen as the basis for a representation of software architecture [Kacem et al. 2006].

The diagrams in UML that can be used to represent a software architecture are the component and deployment diagrams. The component diagram is used to show the software components and connectors that are a part of the system. This diagram does a good job of showing the main parts of a software architecture, but since it is not specifically used for this purpose, there are certain aspects that it cannot capture fully. For instance, architecture description languages typically contain documents and representations for describing the components and connectors of a system, and often include constructs related specifically to the architecture level of design, such as task priorities or deadlines, and are designed specifically to support verification and validation activities. Since UML is a more general modeling language, it does not support these

architecture specific features. The deployment diagram gives a high level view of the software architecture, and includes information about the hardware systems and executables that make up the final deployment of the system under design, but does not provide the same level of detail as an ADL [Krutchen et al. 2001].

UML offers several advantages in the domain of product specific architectures. First, UML is a well-known modeling language standard that is easily understood by almost all practicing software engineers. This allows the architecture information to be shared across several teams very well, which is important in developing product families since separate software groups generally create separate products. Unfortunately, the lack of discipline that is allowed in a purely UML based scheme makes this an unlikely choice for the overall basis of the architectural design. Part of the UML definition allows for extensions to be applied, but this requires a great deal of work by some organization and loses many of the advantages of UML by deviating from the standard. The usage of UML, however, in conjunction with a more formal and rigid framework is an acceptable solution to the issue of representing the software architecture effectively.

2.1.2 Architecture Definition Language (ADL) Approaches

A similar approach to the use of UML in software architecture design is to base the framework around a specific architecture definition language (ADL). An ADL is a modeling language like UML, but one that is designed specifically to provide methods for capturing details specific to architecture design and assisting the architects by providing guidelines as well as assistance in tracing the requirements to the design. This is a more rigid method than simply using UML and is more likely to lead to designs that support reuse of the components developed.

However, simply using an appropriate ADL will not guarantee the creation of reusable parts [Clements 1996].

An ADL-based system would be better for the style of architecture needed for a product family architecture, but it lacks the same support for collaboration that UML provides. It is possible that an ADL could provide adequate communication between the separate groups, but its usage would require a substantial initial investment by the organization in the chosen ADL. This is a problem because there are really no well-established ADLs currently available, and committing to one of the current languages could be problematic because no single language has been widely adopted [Clements 1996]. New employees would have to be trained in the language, and recruiting may be hard because of the use of an obscure technique. The technical challenges are also greater when using an ADL as there may be a lack of tool support and a lack of information available regarding the correct usage of the language and its application to the domain being modeled.

2.1.3 DODAF and Related Frameworks

The United States Department of Defense has established an architectural framework to be used in systems that it develops or purchases [DoD 2004]. This framework, called the DODAF, prescripts step by step exactly how the architecture should be designed, including artifacts that must be created. While it leaves the language that is used to implement the artifacts as a decision for the architect, a complete design must include all of the artifacts specified in the framework description [Leist and Zellnor 2006]. There are several other similar frameworks found around the globe, both military and civilian in nature, such as the MODAF and DNDAF, from the United Kingdom and Canada respectively. By creating a software architecture that conforms to the DODAF standard, you are guaranteed a high level of uniformity in the description across

systems compared the UML and ADL based approaches. While the DODAF does encourage reusability by requiring that certain artifacts be created and meet certain parameters, it does not specifically address the issues that a product family architecture and implementation faces, which include developing a core architecture and defining the way that this core can be used for each product.

The DODAF alone does not provide all of the requirements for the architecture methodology that is needed, but it does provide some key concepts that the methodology should be able to leverage. Like the DODAF, a software architecture specifically tailored to product families should give some type of guidelines that assist the architect with what types of artifacts and documentation should be completed. With this requirement, the core architecture can be developed by a separate team than the one that designs and implements each individual project. The standardization that it provides is essential to making sure that all of the documentation across the organization can be combined. The DODAF also has the advantage of being backed (and in some cases required) by the United States Department of Defense, which gives it a wide user base and gives organizations some assurance that it will continue to be used throughout the industry well into the future.

2.1.4 Service Oriented Architectures

A somewhat different paradigm when designing a software architecture is to base the architecture on defining the set of services that each component provides and make this the focus of the design. The design can then be articulated using any modeling technique that the team chooses, so long as it can portray the services and components correctly. This approach to software architecture is gaining popularity in much of the software industry with the increase of distributed systems and the Internet [Radhakrishnan and Sririman 2007].

Service Oriented Architectures could be applied successfully to the domain of product family architectures. By defining the core architecture in terms of the services that it provides to each of the product specific components, the components and services could easily be analyzed and refined to be certain that the core components provide the services that each product requires [Rafik and Zeid 2004]. The language independence that it provides allows for easy communication between the teams once a standard is chosen. Also, the concepts of SOA promote the modularity that makes reusable components possible in a system that conforms to this paradigm. One key drawback in basing an architectural style on a pure SOA style is that this choice limits the type of architectures that can be created. If a more general approach to describing software architecture is taken, it may still be possible for an SOA compliant software architecture to be designed and described using the method while still allowing for other more general forms of software architectures to be created and represented. This is believed to be a better approach, as there may be systems of product families that do not conform well to the SOA model, and the goal is to create a method that can be used for any software product family that may exist presently or in the future.

2.1.5 C2 Architectural Style

The C2 architectural style was proposed by a group of researchers at the University of California, Irvine. This style is based on message passing as the sole form of communication between components. The style provides strict guidelines for how the components are defined. It also places great emphasis on the use of connectors as a liaison between the components. A unique ADL is provided to describe the architecture, although it could also be expressed in other languages. The strict guidelines that this style provides as well as the principles on which it is based make it an excellent candidate for use in product family architectures [Medvidovic 2002].

C2 supports all of the reuse requirements for the product family architecture. The message-based environment promotes the principles of reuse effectively without ruling out certain solutions such as an SOA approach [Medvidovic 1997]. The main issue with C2 is its usage of a unique ADL that is not widely used in the software industry. However, the ADL does a good job of expressing the architecture and allowing each of the groups to communicate correctly and easily. The style also provides the appropriate amount of guidance to the architects, which is not as little as a UML based approach.

2.2 PRODUCT FAMILY SPECIFIC ARCHITECTURES

Examples of product families are found throughout the software development industry [Clements and Weiderman 1998]. There are numerous organizations in the software industry that face the same challenges, and many have implemented similar strategies with varying degrees of structure and with varying degrees of success. These include such products as printers, video games, office tool suites, banking suites, and others. The idea of using software architectures to assist in developing such related products has existed in industry for several years. A number of papers and books have been published regarding how best to apply each approach to easily develop such a group of products [Jazayeri, Ran, van der Linden 2000].

Nokia is one company that develops software for a variety of different cellular phones. Each phone offers some distinct functionality, but each shares the common functions that a cellular phone must always perform. These functions include making calls, a phone book function, and remembering calls that have been made and received. As part of their effort to make the development of each product fast, efficient, and uniform, the research team for the organization developed an architecture framework, complete with a custom developed toolset

based on the Rational Rose toolset that allows each product to share information and components with one another [Riva et al. 2004].

For the purposes of this project, the types of software product families can be divided into two major categories, embedded and non-embedded software products. These two categories have different issues when considering architecture and design, as well as all other phases of the software process, from requirements to performance characteristics [Gomaa 2000]. While an embedded program may have to focus on delivering certain time guarantees, such as in a real-time system, non-embedded software may have to serve thousands or even millions of users. It is important that the requirements for these systems be dealt with accordingly, because software architecture is based on the production of sound requirements [Kuusela and Savolainen 2000]. Work has been done in developing frameworks in diverse areas such as web [Balzerani 2005], operating systems, desktop applications, and embedded applications.

The engineering of software product families has made great progress in several European countries. The ARES project was a collaboration of several parties in the European engineering community, including Philips Research, Nokia Research, the Polytechnic University of Madrid, Technical University of Vienna, and others. This partnership of industry and academia was able to produce several sound ideas regarding the engineering and software architecture of product families [Jazayeri et. al 2000]. The project focused on important goals that align well with this study, including software reuse and software process. The ARES project was centered on software architecture, and proposed some important methods for capturing architecture in these types of systems [Jazayeri et. al 2000]. The project included case studies at Nokia and is related to the development of the custom support software there. This particular project made use of the ADL Darwin to enumerate the architecture views and chose not to use a

more standard language such as UML partially because it is intended for low-level design and does not easily provide the level of abstraction desired [Magee et. al 1995]. This is a valid concern that must be evaluated and accounted for in software architecture systems based on UML.

Another key area of software architecture and product families is the area of software architecture recovery. This area consists of reverse engineering a piece of code that has already been completed in order to represent the architecture of the system. This is applicable to this project as much of the code and a basic software architecture design are already in place. There have been several efforts to achieve this, including as part of the ARES project [Eixelsberger et. al 1997]. Similar works focus on using common reverse engineering techniques and applying them to the field of software architecture [Eixelsberger et. al 1998]. Many of the techniques discussed in this literature will be used when developing the software architecture for AubieSat-1, for which much of the code had already been written before the application of the new software architecture methodology.

Matinlassi gives an excellent survey of state of the art product line architectures [Matinlassi 2004]. The methods are compared according to several criteria, which include goals, inputs, outputs, required skills, users, tool support, and many others. These techniques are most closely related to this work, and their examination can provide great insight into how the method can best achieve its goals by building on the best aspects from these methods while improving upon some of their shortcomings. Each of these methods have separate goals and can be used for developing a software architecture under different circumstances, given the application domain, culture of the organization, and experience of the engineers.

2.2.1 Component-Oriented Platform Architecting Method (COPA)

COPA was developed by engineers at the Philips Research Labs, and is mainly used in the telecommunication infrastructure systems domain, but has also been applied in the medical domain. As its name states, COPA is a component-oriented method that hopes to achieve reuse through the creation of standard software components that can be easily applied to different systems. The method is unique in that it attempts to combine a top-down approach based around software architecture with a bottom-up approach based on software components. This approach allows for the successful use of several components in different products of the same line, by using software architecture as a tool to combine these components. The method is based around the BAPO product family approach [van Ommering 2002]. It covers all aspects of the product line lifecycle, including requirements, process, and organizational aspects. This method is not specifically targeted towards any particular part of an organization, but is used by all members.

2.2.2 Family-Oriented Abstraction, Specification, and Translation (FAST)

Developed at Lucent Technologies around 1999, the FAST architecture method is meant to make the software engineering process for product lines more efficient by reducing costs and shortening time to market [Weiss, Lai, and Tau 1999]. The process is aimed at software engineers and hopes to reduce the amount of redundant work that must be performed when creating similar products, and has been successfully applied to telecommunications and real-time systems. This is a highly practical method that was developed in industry, and its benefits are easily recognizable. It divides the process for developing a product line into three separate parts: domain qualification, domain engineering, and application engineering. While it does require the use of tools in order to perform the required activities, it does not explicitly define the set of tools used to support these activities.

2.2.3 Feature-Oriented Reuse Method (FORM)

This method begins with applying domain analysis of the commonality and variability between the separate products in a given line to an engineering process to create reusable and adaptable components. The method is focused on software "features", which are product characteristics, and provides a nice abstraction that can be easily understood by customers, domain engineers, and software engineers [Kang et al. 1998]. The method includes feature modeling, domain engineering, and application engineering. Created at Pohang University of Science and Technology in Korea, it is an extension of the Feature-Oriented Domain Analysis (FODA) method [Kang et al. 1990]. One unique aspect of this method is that it provides automatic transformation from requirements to a domain architecture.

2.2.4 KobrA

KobrA stands for a German phrase that translates to "component-based application development". It is unique in several facets from other product line architecture methods [Atkinson et al. 2002]. This method can actually be applied to systems that involve only a single software product, as well as to software product lines. This method supports a Model Driven Architecture (MDA) approach that allows it to be platform independent [Frankel 2003]. It defines activities to be performed as well as artifacts to be produced, and even goes as far as to include directions for the implementation and testing phases of development. Although originally developed for the information systems domain, it is easily adaptable to other domains by allowing explicit customization of the method. The two main activities performed in this method are framework engineering and application engineering. This is a very practical method aimed at practicing software engineers, and is the only method that deals with the practical aspect of configuration management in the method. By supporting industry standards, such as

MDA and using UML for artifact representation, this method eases the transition and special knowledge needed for its successful implementation.

2.2.5 Quality-driven Architecture Design and quality Analysis (QADA)

QADA is unique among product line architecture methods in its focus on quality requirements. All of the views in this method feature quality attributes, and the major factor in designing the architecture of a system is to support the quality requirements of that system [Matinlassi et al. 2002]. This approach presents a different paradigm compared to the traditional software architecture methodology, and although it may not be the most appropriate fit for all software product lines, there is much to be learned from the focus placed on quality in order to develop a fully functional, reliable, and maintainable software system, which are attributes that are magnified in the case of software product lines since the same design will be used in several different products. Two abstraction levels are used in the design in this method, which are conceptual and concrete. The conceptual focuses more on domain related terminology and components, while concrete follows a more traditional software centered architecture approach. This method is intended for use in the middleware and service architecture domains. Although created as an academic project, it has been used successfully in several industrial systems. This method also relies on UML for artifact description and conforms to IEEE standards for viewpoints in the definition of its artifacts, which makes it more practical to be used by practicing software engineers and software architects.

2.3 SPACE AND SATELLITE SOFTWARE ARCHITECTURE

There are similarities in the software of members of any group related systems, and space and satellite systems are no exception. The design and architecture of such systems are often closely related, making it beneficial to examine the design of the software on current and past

satellites around the country and world. Related particularly to this program are other cube satellite programs. The AubieSat-1 program at Auburn University is not the first of its type [Chapman, Wersinger, and Wilson 2008]. Several other universities and organizations have also developed similar cube satellites, many of which have been deployed successfully [Dabrowski 2005]. It is important to learn from these efforts in order to best achieve a design that allows for the success of this and future missions by providing a safe, reliable, and extendable design.

Although similar, space systems and satellites offer a good deal of variety with regards to design and implementation. They use several different paradigms, design techniques, programming languages, and operating systems. For instance, several of these systems operate in the structured programming paradigm and use a language such as C, ADA, or a related language as the main implementation language. Some systems, however, follow object-oriented methodology and use an object-oriented language such as C++ for implementation [Murray and Shahabuddin 2006]. From an architecture standpoint, it is important to understand the different methods that are used for low-level design and implementation, as a software architecture framework intended for this domain should be agnostic with regard to the approach taken at these lower levels and should be capable of easily accommodating any of these techniques.

Some efforts have been made to date to allow software to be reused in satellite systems [Pasetti and Pree 1999]. It has been recognized that these systems are an area that could apply software reuse principles effectively. Most systems, however, still do not apply many of these principles and often develop most of the software for each new satellite from scratch. Most current work focuses on the use of object-oriented design principles to allow for reuse [Pasetti and Pree 1999]. It may be possible to use software architecture to encourage reuse at a higher

level of abstraction and to allow any programming paradigm to be included, even legacy functional software systems.

A similar approach that is often taken in this field but that is able to reuse components at a higher level is to combine several consumer off the shelf (COTS) products into a single system [Ferguson and Thompson 2005]. This allows for the quick development of a software system that uses reliable, well-tested components that often have a long record of successful use in related applications. By using software architecture techniques to support this type of effort, it is possible to capture the design of such a system in a way that will provide traceability between the requirements and implementation of the system and allow for other similar systems to use the same components and design when needed.

2.4 RESEARCH OBJECTIVES

Much like the current software architecture methods, this method will offer an alternative that may be selected under the correct circumstances for certain software product families. The new architecture methodology will offer several key contributions not present in the current literature. By allowing for the variability and commonality of the products to be easily captured through the selection of core components, the method will assist in easing the process of developing the software architecture from the requirements for a system. The method will also be able to leverage the use of only standard UML in all of the artifacts that it requires.

The core-based architecture approach used in the Core Based Architecture for Product Families (CBAPF) is unique and provides an entirely different paradigm for approaching the software architecture of a group of related products. A major issue with all of the current methods is that they require either a unique ADL or some modification to UML, which in turn requires special knowledge and tools to support [Matinlassi 2004]. Although it does not restrict

the language used, CBAPF is able to describe the software architecture completely using only standard UML diagrams, and the case study of AubieSat-1 will be used to provide an example and prove the practicality of the method.

Another unique contribution of this work is the application of this method to the space and satellite system domain. While several methods have been applied to real-time systems, none have been applied to this domain specifically, and the advances gained will be beneficial to all organizations in the industry. This method also hopes to remain as domain independent as possible, in order to support the widest possible variety of product families for modeling, whether embedded software or not, and allow for adaptation of the method in several domains so that the greatest benefits can be achieved.

Chapter 3: Software Architecture Methodology Definition

3.1 A CORE BASED ARCHITECTURE FOR PRODUCT FAMILIES

In order to describe the software architecture for a software product family in the best possible way, a new software architecture methodology is proposed here called Core Based Architecture for Product Families (CBAPF). This architectural style makes use of a central set of components, called the core, which is reused in each of the products within the family, and a set of external components that are used in each family, as seen in figure 1. This set of components must be defined by a team that is familiar with the requirements of each product in order to determine the best common subset of functionality that should be included in the core component. Each product then uses this set of software components with no modification, only adding the product specific parts to the system in order to add the specific functionality that is required. The CBAPF model dictates certain artifacts that should be created in order to enforce this paradigm. This model is not meant to be a generic style that can be applied to several different problem types, but it is flexible enough to be used in any type of software-based system that includes a set of related products. It is instead considered more beneficial to provide specific guidance in how the architecture is designed, at least to a certain extent, in order to guarantee that the principles are followed correctly. Although it would be considered advantageous to reuse some of the product specific components in products that share features not common to the core, this is a secondary goal of the architecture and is not addressed specifically by this method.

The C2 style is chosen as the basis for the CBAPF system because of its properties that support the main goal of reuse [Medvidovic et al. 1997]. C2 focuses on describing the components and connectors found in a system, and it relies on message passing as the sole forms of communication. The CBAPF architecture shares these same characteristics. In this implementation of the C2 style, there are two major components at the top level, which are the core component and the product component. These components can be divided into the many smaller components by the individual organization, but this breakdown is the key to the architecture. C2 is chosen because it adequately represents the architecture and its key concepts and is an exceptional way of articulating the design while forcing the architect to conform to the chosen style. The CBAPF also takes some guidance from styles such as the DODAF by requiring specific architectural artifacts to be created.

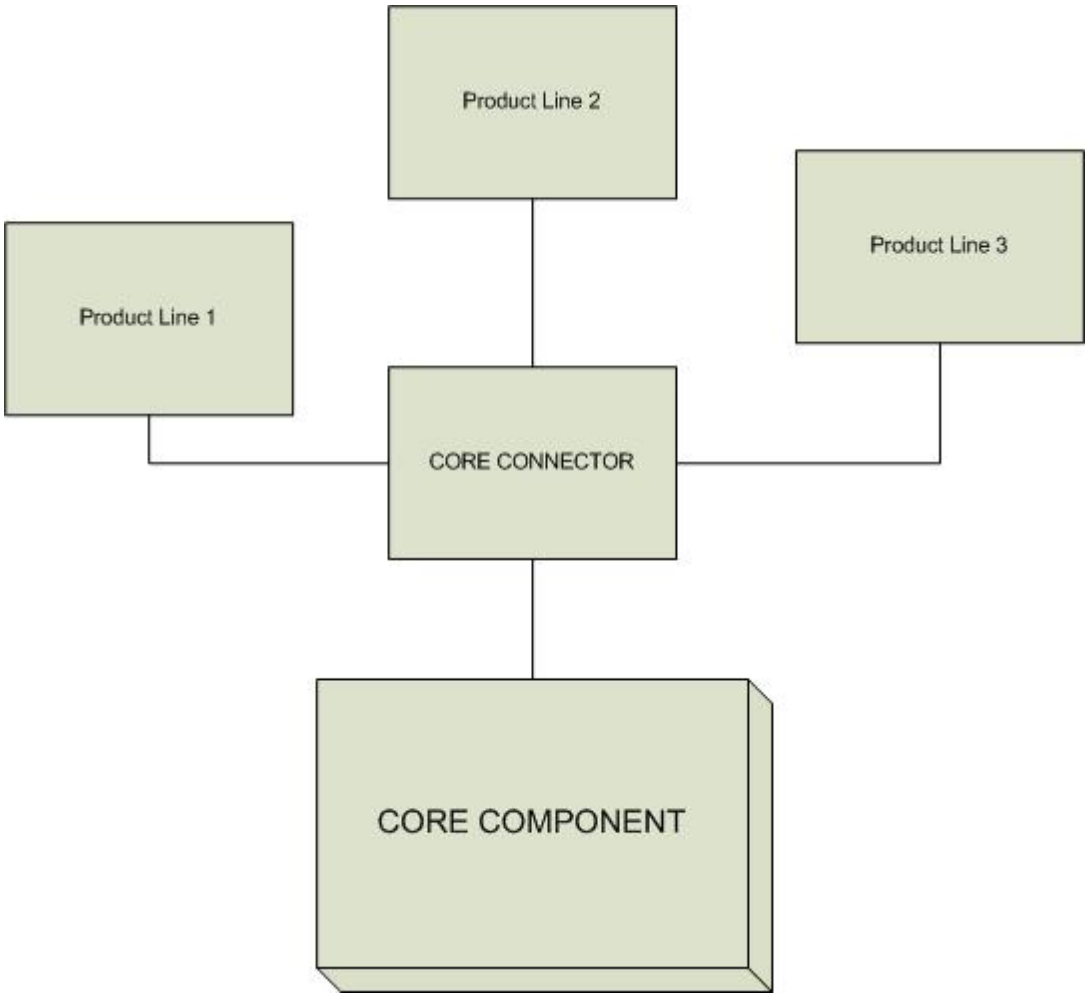


Figure 1

This method differs from the C2 style by using UML diagrams to represent the required artifacts. These diagrams are chosen because they can be used to effectively display the style. Because UML is well known and has a large user community and tool support base, it is currently the best medium for sharing documentation among large organizations. The UML component diagram is used to implement the required artifacts, without modification to the diagram definition. This will allow the technique to leverage existing tools and knowledge effectively. However, the usage of UML is not essential for the style to be applied correctly. The same artifacts could be produced in a different modeling language, and all of the same

concepts would apply. Only the modeling semantics would change, but the artifacts that are required could be represented using any common ADL or even another modeling language. This decision is left entirely to the organization.

Although UML is used to model the software architecture, an object-oriented programming language is not required. Ideally, a software architecture methodology should be independent of the programming language or paradigm used to implement the software architecture that it produces. This is true of CBAPF, which, although it uses UML for its diagram, does not use UML in the commonly accepted way. By viewing software components as classes in the architecture realm, it is easy to represent the software architecture with standard UML diagrams, but these are still software components and connectors that can be object-oriented or not, as the software architect and project team sees fit. It is important to notice that the architecture does not depend on object-oriented design or principles. Although it is possible to represent object-oriented principles, such as inheritance, within the diagrams, this is simply a property of the flexibility of the method and does not in any way require the usage of such principles in order to fully use CBAPF correctly.

3.2 REQUIREMENTS IN CABPF

Requirements engineering is an irreplaceable part of the software development lifecycle that allows the software developers to analyze and understand how to correctly implement a solution to the problem that they are facing. This holds true for software engineering of product families, in which it is important to recognize the similarities and differences in the requirements for different products, in order to comprehend how the implementations of the products will differ. Many of the current software product line architectures include some direction on how to perform the requirements for a project. In CBAPF, requirement analysis is an essential step that

must occur before the CBAPF method can be applied. In order to correctly design and create a software architecture for a software product family, the requirements must first be known for the family of products and for each product independently. The method used to define and analyze these requirements is independent of the CBAPF method, and a discussion of how to perform these steps is beyond the scope of the method. CBAPF chooses instead to focus on how to correctly design and model the software architecture after the requirements have been defined. It is often very hard if not impossible to define the full requirements before the architecture is created, but some general knowledge must be used in this case and the architecture will adjust as the requirements become more concrete.

3.3 REQUIRED ARCHITECTURAL ARTIFACTS

3.3.1 Core Architecture Description

The first required document in this architectural framework is the core architecture description. The software development team that develops the core component that is used in each of the products develops this document. This method assumes as a precondition that the requirements for what should be included in the core have already been determined. A team that includes both the core developers and representatives from each of the product teams should cooperate to define the set of core components so that the common functionality and common startup functions that the core should be performed can be defined correctly. Once this is accomplished, the core architecture should be designed in the core architecture description.

In accordance with the C2 style, this document must define certain key parts of the architecture. First, any components that make up the core architecture must be described here. Only the name of the component need be included on this diagram, as its details are not

necessary or appropriate at this level of abstraction. Next, the connectors that allow the components to communicate via message passing should be defined. One connector is particularly important in this view, as it allows interaction with the product specific components. This connector, which will be called the core connector, is special in that it will interact with components outside of the scope of the architecture of the core, and it therefore requires more information than the other components. These details will be covered in the core connector description explained in the next section. For this document, it is only necessary to denote which connector will be the core connector, either by a naming convention or a note. A key advantage of using a style based on C2 is that there is some level of decoupling between the components, which in this case are the core and the product specific components. This attribute allows the team of developers responsible for developing the core components to modify and improve the core while the products themselves are being developed concurrently. The team need only ensure that the components created meet the same requirements and conform to the core connector interface.

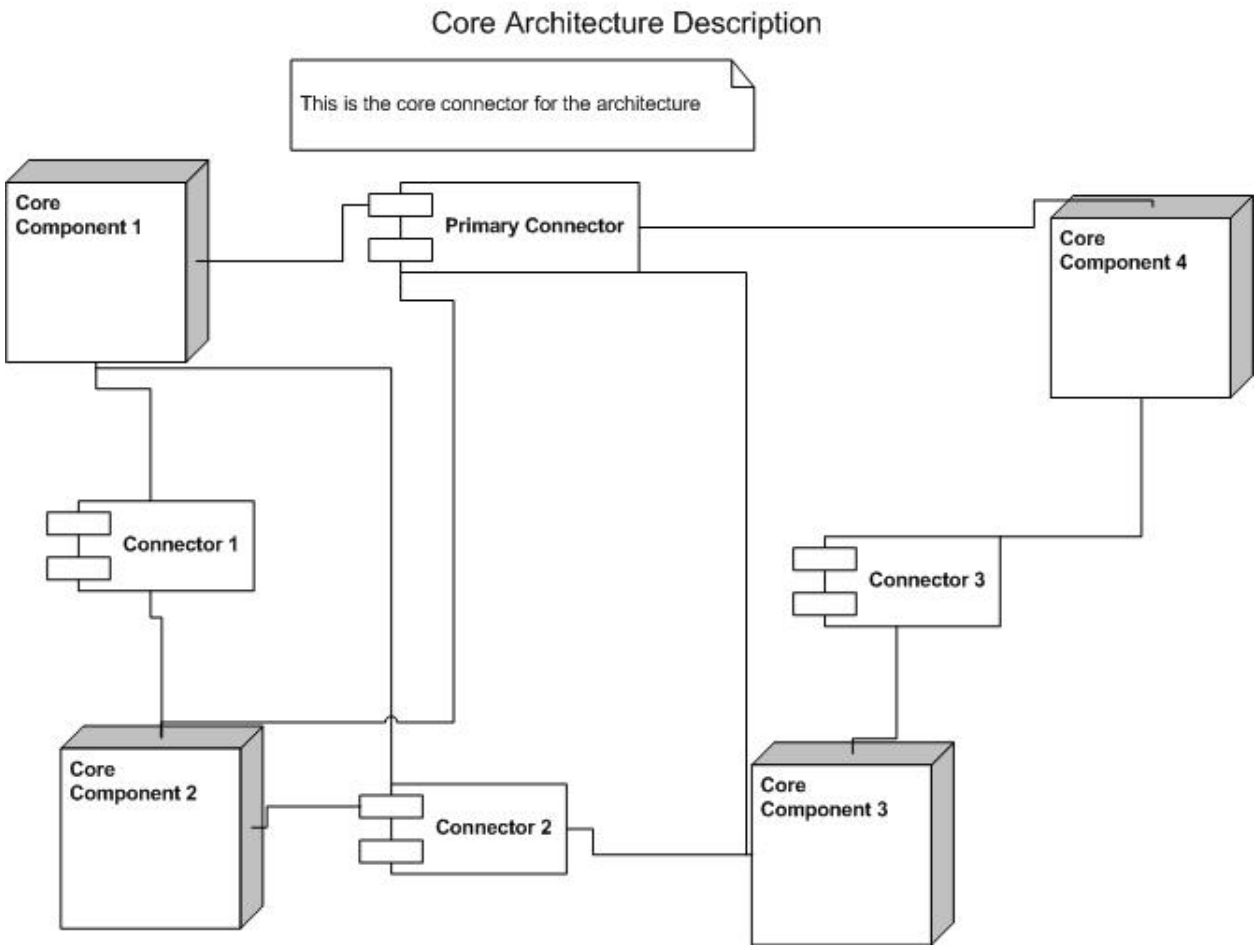


Figure 2

The core architecture description in CBAPF is implemented using a UML component diagram. Each component should be defined as a node, and each connector should be defined as a component. While this may seem counterintuitive, connectors are in fact first class components in the C2 architecture, and therefore this representation is valid when converting to the UML domain. The architect should denote in some manner the core connector. An example core architecture description is shown in figure 2.

While a top-level core architecture description is mandated in this approach, this does not rule out the creation of multiple documents to describe the architecture. These documents can be used to represent several levels of abstraction in more complex systems. This feature is

important for the scalability of CBAPF when it comes to designing large software systems. It is recommended that these documents be linked to each other for traceability purposes. The use of UML notes would suffice for such a link, although tool supported links would be a better solution.

3.3.2 Core Connector Description

The core connector is responsible for handling all of the interaction between the set of core components and the set of product specific components. This forces the core connector to interact with many different components in the architecture. For this reason, special care must be taken in defining and agreeing upon the interface that the core connector provides. These requirements merit the creation of a special artifact, the core connector description. This artifact must portray certain properties of the core connector. First, each of the functions or features that the connector provides to the product family should be described. In order for this connector to be applied successfully, full function signatures should be provided for each message that can be called, including return types, parameter types, and names. This can require a great deal of initial design to take place, but is a necessary part of the architecture, as without this level of detail the interface is of little use. Next, the artifact should include the set of messages that can be sent by the components that are included as part of the core architecture. Again, these should include full signatures. The core connector should be defined in coordination with the core architecture, and the core connector description and core architecture description should be created concurrently.

In CBAPF, a UML class diagram is used to represent the core connector description. This is a non-traditional usage of a class diagram, as the core connector is a software connector and not a single class. This allows the same UML document and methodology to be applied

regardless of the software language used to implement the system and refrains from restricting the system and architect to solely object-oriented languages. The same usage is found for all UML diagrams in CBAPF, which uses the UML framework but at a higher level of abstraction than its original purpose that is not concerned with the programming paradigm used to implement the architecture design. The core connector diagram is a special class diagram that only has one class. This class should have no attributes, only method descriptions. The methods described are actually the signatures of the functions that the core connector provides. In order to differentiate between functions used by product components and those used by core components, the functions used by the product components are marked as public, while those used by core components are marked as private. This distinction is logical when the architecture as a whole is considered, as the functions used by the core are not available to the products and therefore not visible outside of the core, while the product functions are available as the primary interface for using the core. An example is shown in figure 3.

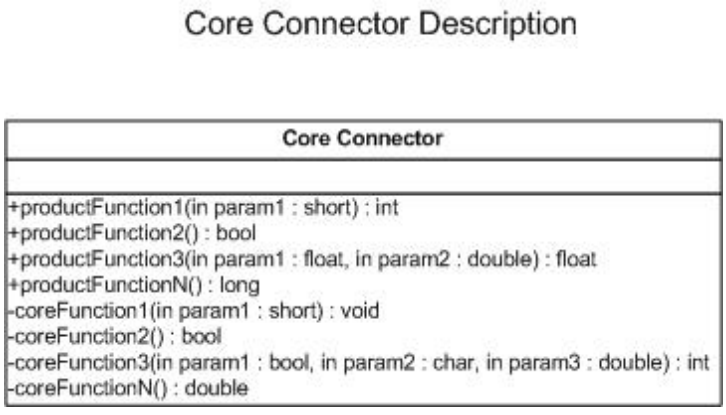


Figure 3

Each of the other connectors that are defined in the core architecture description and product architecture description needs to be defined with the same level of precision as the core connector. This method does not rule out the use of diagrams similar to the core connector to

represent these components. The core connector description is required because it is the only connector that must be shared across development teams and architectures. It serves as the bridge between the core and product architectures, and so its details are of great importance to both of these systems. Connectors found only in one or the other, however, need not share the details of their implementation with the other parts of the system.

3.3.3 Product Specific Architecture Description

The counterpart to the core architecture description for each separately developed product is the product specific architecture description. This artifact is essentially the same as the core architecture description, but captures the architecture used in the parts of the product that are specific to that product alone. The components and connectors that make up this architecture must be defined in this artifact. The core connector must be displayed on this document and must be specially defined in some way. This should be the only portion of the core architecture that is included in the diagram.

Just all of the other artifacts described here, this artifact requires as a precondition that the requirements for the product have already been determined. Unlike the core architecture description, this document need not be shared across the different teams in the organization, either the other product teams or the team responsible for the core, and is instead intended solely for use in by the team that is developing the product that it describes.

Like the core architecture description, a UML component diagram is used in CBAPF to represent this artifact. Components are represented using nodes, and connectors are represented using components. Links between the components and connectors are shown using dependency lines that are bidirectional. The core component should be explicitly articulated using either a note or including this property in the name. A generic example is shown in figure 4. This

approach does not limit the possibility of including several diagrams to represent different of levels of abstraction in more complex architectures, but it does require that a top level view be defined and be notated as the product specific architecture description.

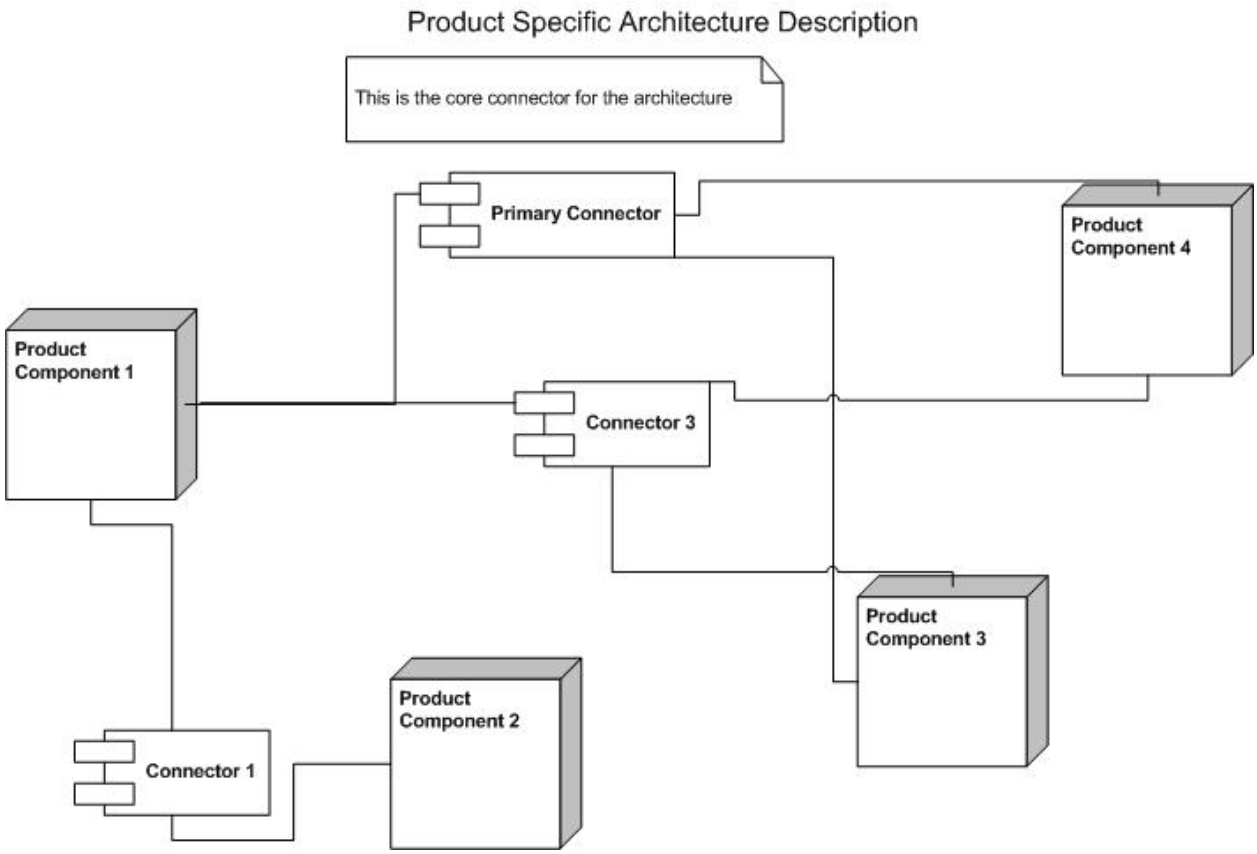


Figure 4

3.3.4 Combined Architecture Description

An overall view of each product is required in order to fully understand how the system as a whole functions. This specification is included in CBAPF in the form of the combined architecture description. The need for this artifact is one of the key factors in requiring that the core architecture description be shared with each product team. This artifact is required for a

few reasons. It is important that each product have a full architecture description that includes each of the components that exist in that system, regardless of the distinction between the core and product specific parts that make it up, because it gives the team responsible for the product something that can be analyzed. This analysis can be for non-functional requirements, such as usability or performance. It could also be used for traceability between the requirements for the product and its implementation in architecture. One of the key purposes of a software architecture is to bridge the gap between requirements and design, and so this traceability is very important. The creation of this artifact also allows for automatic analysis tools or architecture simulation to be applied to this framework. This artifact need not be shared across teams, but is intended for use by the product team that develops the product that it describes.

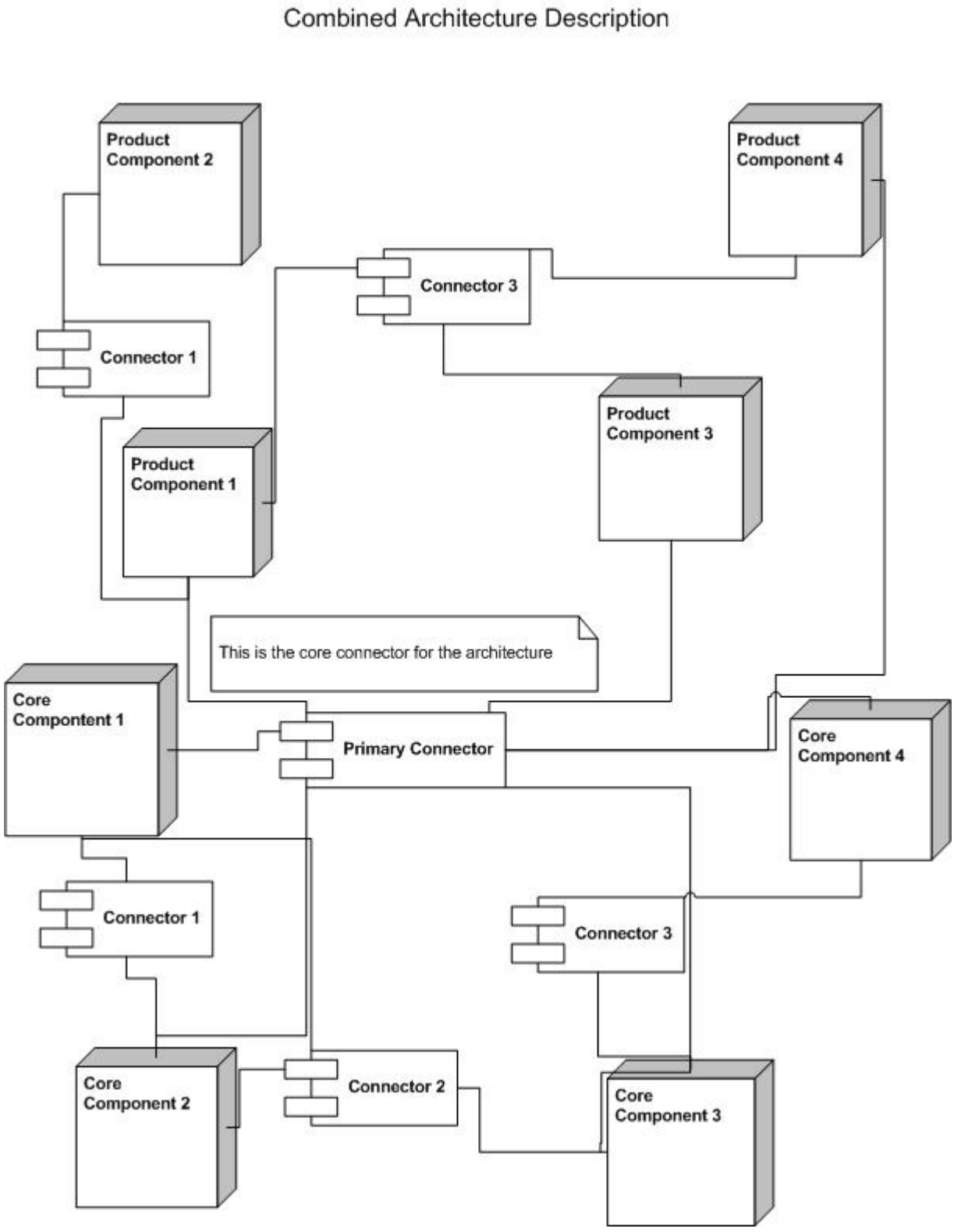

Combined Architecture Description
Product Component 2
Product Component 4
Connector 3
Connector 1
Product Component 3
Product Component 1
This is the core connector for the architecture
Core Compontent 1
Primary Connector
Core Component 4
Connector 1
Connector 3
Core Component 2
Connector 2
Core Component 3

Figure 5

This artifact should be very easy for the team to develop because it is basically a combination of two previously created diagrams. In this diagram, all of the components and connectors from both the core architecture description and product specific architecture description should be displayed. The core connector should be clearly identified and should be centrally located in the diagram. This connector should serve as a clear separation between the product specific and core components such that it is apparent from the diagram the components that belong to each category.

Once again, a UML component diagram is employed in CBAPF to represent the combined architecture description. An example of a generic implementation is shown in figure 5. The components should be represented using nodes, and the connectors should be represented with components. The core connector is also a component, but it is marked with a note to show the difference between the other available connectors. Whether or not UML is used as the modeling language, this diagram should completely agree with the form used in the other diagrams to preserve consistency and allow for greater traceability.

3.4 FIELD SPECIFIC DOCUMENTATION IN CBAPF

The CBAPF method has been designed to capture the software architecture of a product line regardless of the domain being modeled. There may, however, be some important information that could easily and fittingly be captured using this method. For instance, a real-time system may need to include information on the priority of different modules or tasks as part of the software architecture. Any such additional information should be added within the constraints of the other required artifacts, by using the existing UML, at the discretion of the software architect. The use of UML notes is one common method of adding such information. Another common technique may be to make a note linking the document to another document,

which should be a UML document or a standard file such as word processing document, spreadsheet, or something similar.

A major goal of the CBAPF method is to avoid situations in which this additional information must be supplied. Only information relevant to the software architecture itself should be included in the documentation, and any information that might fall into the requirements or detailed documentation should be included in those documents. It is believed that simply following the artifacts provided will adequately capture the software architecture of any product family line in a manner that will establish a core set of components for reuse and achieve all of the goals of a correct and complete software architecture, and any domain-specific additions should be kept to a minimum.

3.5 DETAILED DOCUMENTATION IN CBAPF

The architectural views that are defined in the core-based architecture are meant to be the basis for a successful system-wide design in an organization for related software. This architecture definition alone, however, will not be sufficient to support the environment for developing these products. The definition of the architecture in this manner implies the need for several detailed design documents. In conjunction with the normal detailed design that the team will undertake for the project, this technique suggests a few specific documents that will assist in further defining the architecture and will ease its implementation. The architecture focuses mainly on allowing a uniform design in many different teams, but the detailed documents allow this to be implemented at the code level.

A very important part of the architecture is the core connector and the views that support it. In the detailed design stage, each of the connectors in the system should be fully described as the core connector is. In addition, more focus should be placed on the core connector. On top of

any dynamic views that may be created, an additional static view is suggested to provide even more detail into the interface that this component provides. These documents need not follow any particular format; instead they need only to include the pertinent information described here.

There should be an artifact that describes in detail the pre-conditions and post-conditions for each of the functions that the core connector provides. This will allow the connector to be easily integrated at the code level with the other components. A common technique for performing this type of task is to extract this information from comments that are contained in the source code for the connector. This approach is recommended because it is very practical and causes a low overhead to the developers. It also promotes good coding technique along with the concepts of decoupling and reuse. An example of a tool that can perform this task is the Javadoc tool, which extracts comments formatted in a certain manner into HTML files that can be easily shared, searched, and accessed throughout the organization [Leslie 2002]. Similar tools exist for other programming languages as well, such as Doxygen and others.

In addition to this, there should also be an artifact that describes the details of the functionality that is provided by each component. This should be done at both the architecture and product levels. By explicitly identifying where in the system each feature is implemented, it is much easier to trace the features from the architecture to the code. This is especially important in the core-based system because the implementation is spread between two separate groups of components developed by entirely different groups. It is very important to track where each feature is implemented to be sure that all requirements are met and to prevent redundancy between the two sets of components.

3.6 IMPLEMENTION OF A CBAPF SYSTEM

The proposed architectural framework is of little use if it imposes conditions that make it impossible to actually create any systems when it is used. Here we examine several issues with the implementation of a system using this method and propose solutions. By considering past and current examples, which tools can be used to assist the process, and the management structure and frame of mind needed to implement the system, we are able to provide guidance for how this system can be used in a real organization.

It is important to note that similar methodologies have been used successfully in software projects for many years. Several approaches that are based mainly on software architecture have seen great success [Barrow 2005]. The Nokia solution mentioned earlier, which is very similar to the approach taken here besides the definition of the core, has been used in their organization with great success. These successes provide reason to believe this method could be successful in industrial grade software projects.

In order to fully implement this system in a real project, as much as possible must be done to ease the workload that the framework forces on the development teams. The main way to achieve this is through the use of software tools. The primary motivation for choosing UML as the medium for CBAPF is that there is already a wide range of commercial tools available for creating these documents [Whitehead 2007]. When coupled with project management software and team collaboration tools, such as Microsoft Office Groove, this method will allow documents to be created and shared across different teams easily and quickly [Fox and Spence 2005]. The system also allows for several other possible areas of automation, including parts of the detailed documentation.

Perhaps the key component in the framework is the development of the core components. These components serve as more than just a reusable library to be called by the product specific

components. Instead, they perform basic functionality that the product uses. For example, the core may include initializing and starting the system, system shutdown sequences, and key functions that the system must perform. When implementing the core, all of these functions must be provided. Special care must be taken when integrating the core into the system for each product, which should include both regression testing on the core components and system-wide testing for the entire product.

Another important aspect of the implementation of the software system is the people who create it. This architecture is designed to be helpful to both the system architects and developers. Before using this system, its benefits should be explained to all members of the team. It is essential to the success of this framework that the teams included are dedicated to the method and the concepts it proposes so that it will be implemented and lead to a correct system. As with any method that may be chosen, it is important that all of the teams work together effectively and smoothly. The focus that this method places on ease of communication proves its dedication to the developers and hopes to make the method easy to apply.

Chapter 4: AubieSat-1 Software Architecture

In order to validate the CBAPF method, a case study was required that would allow for the implementation of a software architecture using the method in a setting that would test its effectiveness. The AubieSat satellite program at Auburn University provided a prime opportunity for this effort as it was just developing the software for its first of several student-produced satellites, AubieSat-1. This system represents a software product line, in which each of the software products is a separate satellite created by high level undergraduate and graduate engineering students. It is easy to see that there will be several software functions that will be common between the different satellites that will make up the core components in the CBAPF method, while the software used to support the different missions of each satellite will be represented with product specific components. By determining the most common components that will be needed in future satellites, the AubieSat-1 software was modified to incorporate the software architecture design produced by its application to CBAPF. This will allow future satellite software products to rely on the core that has been created and will allow for the effective reuse of software code.

The AubieSat program also uses a piece of similar software to run on the ground station and communicate with the satellite during its mission. While the ground station does not belong to the AubieSat satellite product family in a strict sense, it does share many of the same components and source code. Because of this, the ground station will be used as an example of how to model different products within the family using the CBAPF methodology. The product specific and combined architectures for the ground station system will be modeled to show how

multiple products can interact with the same core components through the core connector and how the methodology can easily support the modeling of multiple products with reuse of large pieces of code. A hypothetical AubieSat-2 mission is also described in chapter six that uses the same core architecture as the AubieSat-1 mission.

4.1 AUBIESAT-1 COMMAND AND DATA HANDLING SOFTWARE REQUIREMENTS

The CBAPF method requires that the requirements for the system be determined prior to beginning the process. These requirements must be used as inputs when developing the software architecture for the product family. No formal method of requirements was used in order to fulfill this obligation. Rather, the requirements for AubieSat-1 were fully defined using the program wide requirements capturing technique. Analysis was then performed on these requirements to determine which represented functionality that would most likely be required in future mission programs. This is slightly different from what might be done in other circumstances, in which several software products for a certain family are initially planned and the requirements for each product are easier to determine. However, the need to develop the core without having all of the requirements for future software products is a common one [Mannion and Kandal 2001], and so the methodology used here reflects the common practice of industry software engineers who must determine what may be needed for future software products in the product line.

AubieSat-1 C&DH Software Requirements

Core Requirements

These requirements should be common for any satellite produced in the AubieSat program.

1. CDH01.1 (From SYS01.5.2 CDH01.1) – CDH shall control antenna deployment such that the antennas may be deployed 15 minutes **after** ejection from the P-POD (as detected by CubeSat deployment switches).
2. CDH02.1 (From SYS02.1.5) – CDH must be able to receive commands from COMP in accordance with specifications in AubieSat CDH onboard software architecture.
3. CDH02.2 (From SYS02.1.6) – CDH must be able to receive commands from COMS in accordance with specifications in AubieSat CDH onboard software architecture.
4. CDH03.1 (From SYS02.2.1) – CDH must be able to execute commands from COMP in accordance with specifications in AubieSat CDH onboard software architecture.
5. CDH03.2 (From SYS02.2.2 CDH03.2) – CDH must be able to execute commands from COMS in accordance with specifications AubieSat CDH onboard software architecture.
6. CDH05.2 (From SYS02.4.3) – CDH shall be able to store measured sensor data in accordance with specifications in the Aubie Sat CDH onboard software architecture. (This data must be time stamped)
7. CDH07.1 (From SYS02.6.1) – CDH shall be capable of controlling the signal generated for the beacon in accordance with specifications in CDH requirements document
8. CDH07.2 (From SYS02.6.2) – CDH shall be capable of generating a signal for the beacon in accordance with specifications in CDH requirements document
9. CDH08.1 (From SYS02.10.1) – CDH shall be able to control restarts of itself
10. CDH08.2 (From SYS02.10.2) – CDH shall be able to restart upon command from COMP in accordance with specifications in the primary commands document.
11. CDH08.3 (From SYS02.10.3 CDH08.3) – CDH shall be able to restart upon command from COMS in accordance with specifications in the primary commands document.
12. CDH10.1 (From SYS01.1.2) – NASA approved materials should be used whenever possible to prevent contamination of other spacecraft during integration, testing, and launch.
13. CDH11.1 (From SE10.1) – AS-1 must pass functional testing in accordance with the testing requirements document
14. CDH11.2 (From SE10.2) – All hardware shall be able to operate during environmental testing in accordance with specifications in test requirements document
15. CDH11.3 (From SE10.3) – All hardware shall be able to operate after environmental testing in accordance with specifications in test requirements document

Mission Specific Requirements

These requirements are specific to AubieSat-1.

1. CDH01.2 (From SYS01.5.3) – CDH shall control the fact that CubeSats may enter low power transmit mode (LPTM), or enter high power transmit mode (HPTM), and may activate all primary transmitters 30 minutes **after** ejection from the P-POD. LPTM is defined as short, periodic beacons from the CubeSat.
2. CDH04.1 (From SYS02.3.4) – CDH must be able to pass data to COMP in same format that can be read by GS in accordance with specifications in Aubie Sat CDH onboard software architecture.
3. CDH05.1 (From SYS02.4.2) – CDH shall be able to measure sensor data in accordance with specifications in analog signals list.
4. CDH06.1 (From SYS02.5.1) – CDH shall be capable of generating a signal for the science experiment in accordance with specifications in the primary commands document.
5. CDH06.2 (From SYS02.5.2) – CDH shall be capable of controlling the signal generated for the science experiment in accordance with specifications in the primary commands document.
6. CDH09.1 (From SYS02.16) – AS-1 shall be able to operate autonomously from the ground in accordance with specifications CDH requirements
7. CDH12.1 (From EPS04.1) – All hardware must operate off a voltage of 5V (with in a margin of #TBD)
8. CDH13.1 (From SE12.1.1.1) – All master signals generated must be connected to all pins on the ICD with that name

   CDH13.2 (From SE12.1.1.2) – All slave connections can only be connected to one pin with that name on the ICD (with the exception of 5V, 3.7V, and GND pins)

Figure 6

In order to make reliable predictions about such future systems, the main factor that must be relied on is domain knowledge about the systems being created. This comes from experience in the field, research about the field in which the system operates, and learning from similar systems, both past and current. In determining which requirements fit a typical satellite system in the AubieSat program, engineers from several fields, including systems and electrical engineering, as well as project managers and leaders were consulted in order to determine what was most likely to represent the software requirements of future missions. It was also possible to rely on general software engineering knowledge in order to determine which requirements were part of a general software command and data handling system, such as an embedded operating system, some type of file system and networking, and other common infrastructure related components. The requirements for AubieSat-1 can be found in figure 6. These requirements were developed by students in the AubieSat program, and organized to differentiate between those requirements common to all satellites in the program and those specific to the AubieSat-1 mission.

4.2 PREVIOUS SOFTWARE ARCHITECTURE DESIGN

The previous software engineers working with the AubieSat-1 system software developed a foundation for the software architecture design of the system in order to reflect the state of the software. The original software architecture document can be found in Appendix A. This document describes each of the main software components in the system by describing the software tasks that are found in the software. This software architecture is used as the foundation for producing a software architecture that supports the goal of reusability through the CBAPF method.

The AubieSat-1 software consists of several different components that allow it to fulfill its requirements and achieve mission success. The software architecture is divided into six different tasks: the control task, housekeeping task, AX.25 networking task, secondary command task, beacon transmit task, and the data logging task. According to the requirements needed for the future AubieSat systems, all of these tasks will be included in the core portion of the architecture except for the secondary command task. There will be some reorganization of the tasks in order to separate the portions that are more likely to be reused. It is, however, expected, that most of tasks and component found in the AubieSat-1 software will be included as part of the core component in the CBAPF architecture, because this is the first mission and is mainly meant to establish a successful and reliable product with all of the basic necessary features.

4.3 AUBIESAT-1 CBAPF IMPLEMENTATION

Having determined the requirements for the software of the AubieSat system, the necessary inputs to begin the CBAPF documentation were complete. The first artifact that had to be created to begin the CBAPF process is the core architecture description, which identifies the architectural components of the system that will make up the core component to be reused in each product in the product line, which in this case will be each separate satellite in the AubieSat program.

4.3.1 AubieSat-1 Core Architecture Description

Figure 7 shows the core architecture description as described in AubieSat 1.

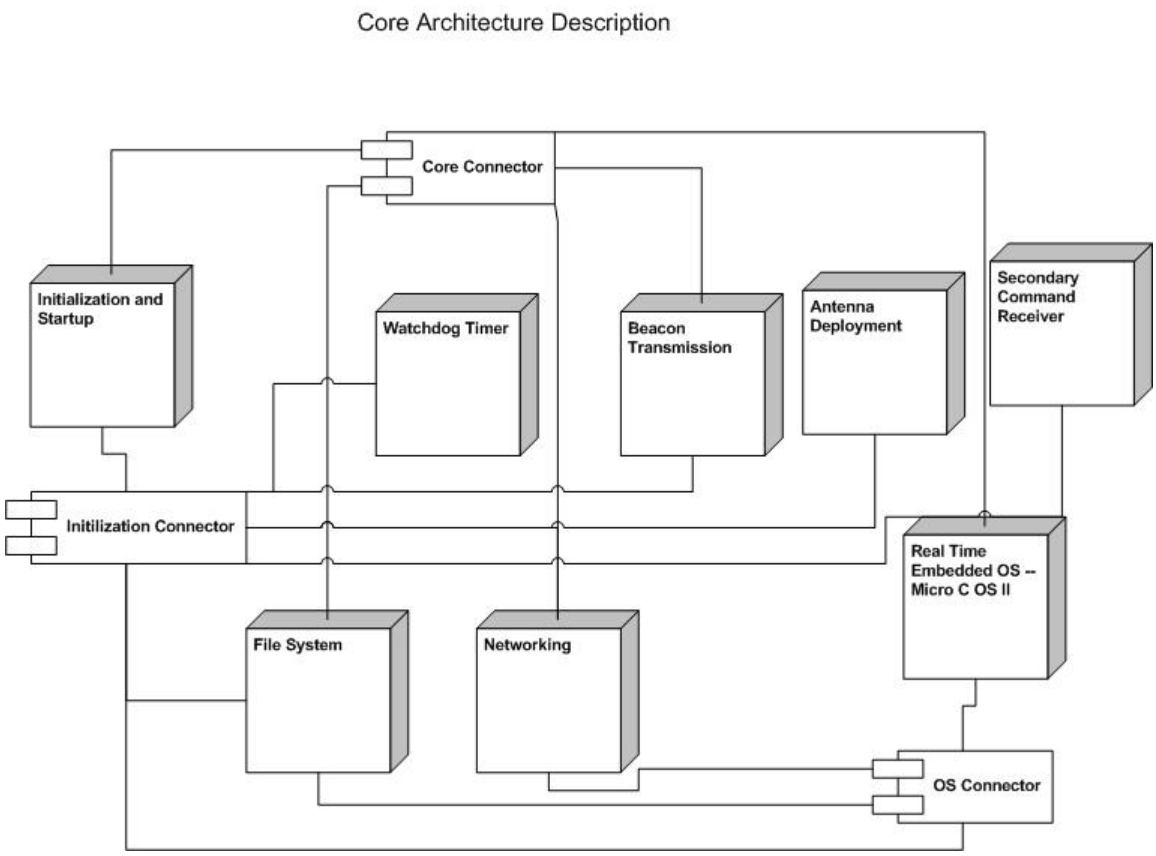


Figure 7

These components combine to perform all of the necessary functions of the AubieSat core requirements. The initialization and startup component is responsible for initializing all parts of the core, from the operating system to the initial connection of the networking component and file system startup. The initialization connector is an interface to the component that interacts with the several different components. The file system component is made up of a file system task and several file system operations that allow other modules to read, write, and create files.

This file system component is only a basic file system. This is different from the original implementation and design of AubieSat-1, in which there was no separation between the basic file system and the portion specific to the mission for storing and retrieving scientific data samples. The latter has now been separated into a separate component in the product specific design called the scientific data-logging component, described in detail later. In the core

component, the file system component is a custom file system with custom file and directory structures that communicate with flash data over an serial peripheral interface (SPI) bus. As seen here in the CBAPF architecture, this component is a generic file system component, and the actual implementation is left to the latter stages of the software lifecycle. This could be a custom file system, such as the one actually used in the AubieSat-1 code, or a popular commercial file system such as NTFS or NFS. The implementation of the component could even change in later AubieSat systems and still conform perfectly to the CBAPF core architecture definition. This flexibility is very important to achieving the reusability of code and design that is the goal of CBAPF.

While it is important that an implementation be chosen that will support all missions as best as possible, it may be the case that one implementation is used for a mission, while a different implementation is used later to improve the overall quality of the system, such as reliability and interoperability or to support standards, or simply to adapt to certain file requirements of the data of future systems. The important note from a software architecture standpoint is that the core architecture consists of some file system component and that file system component is capable of achieving the requirements of the core component and conforms to the core connector interface established for file system usage. By allowing all interaction between product specific and core components to take place through the core connector, it is much easier to allow single components to be replaced within one of these domains so long as it performs the expected functionality and conforms to the interface as required.

The networking component is responsible for establishing a network connection with the base station to send and receive messages. This is part of the core component as there will always be a need to communicate with a ground station in order for the satellites in the program

to operate successfully. While there may be some modifications to the messages sent, the basic need for some form of communication will not change. Much like the file system, the actual network implementation used could be changed in future missions, but the current implementation will be able to provide networking successfully, if only at a minimal level. AubieSat-1 currently uses a custom AX.25 protocol networking implementation. This supports several messages such as stop transmission, resume transmission, restart system, and request science mission data. Also like the file system, the portion of the software that deals with mission specific messages is included in a separate component, the networking protocol component, which is explained in the product specific documentation.

The antenna deployment component is also included as part of the core architecture design and is responsible for deploying the antenna hardware of the satellite to allow a communications link with the ground station to be established. This operation will be required in other satellites, which makes it an ideal candidate for the core. This component must interact with the hardware by sending a signal to deploy the antenna.

Two more components that provide basic core functions are included in the AubieSat core component. These are the beacon transmitter and secondary command receiver components, and the two are similar in the fact that they are required by the United States government for any space satellite, which means they will be required as part of all future AubieSat missions. The beacon transmitter automatically transmits a beacon signal to the ground station at a given interval and is contained in its own task. The secondary command receiver component is a backup to the normal networking component. It is responsible for operating in the case of a normal networking failure, and is also able to implement certain important commands in emergency situations, such as a forced shutdown, but the messages can

only be received and not sent. While it is generally good to attempt to reuse the code for similar pieces of software, in this case it is important that a different strategy is used including the design and code of this component in order to allow for redundancy and decrease the probability that the same or similar bugs will appear in both components. These components should be very nearly fully functional as needed in all future missions as implemented for AubieSat-1, and so the reuse value of these components should be very high as they can be included in the core and reused in future missions with basically no modifications to the source code.

These components combine to form the complete basis for the AubieSat core component. This component will be used in each of the AubieSat systems, and should comprise the vast majority of common requirements and functionality between all of the systems. Although slight implementation modifications may be necessary, the design shown here should allow the code to be reused easily without major changes to the source code. Partitioning the design correctly and using message passing through interfaces for communication between the separate components can do much to allow for the easy reuse of relatively large software components in a significantly complex system, as this example shows.

4.3.2 AubieSat-1 Core Connector Description

The core connector for the AubieSat system serves as an interface to all of the components contained in the core component. As described in CBAPF, the core connector must be well defined to ensure correct interoperability between the product specific components and those found in the core. Figure 8 represents the core connector description for the AubieSat program. The same core connector is used in each product in the product family. As such, great care must be taken when defining the public operations that the connector provides. Although new operations can be added to the connector with little trouble, it is extremely important not to

remove or modify any existing operations for the use of a single product in the product line, as this may have far reaching effects in the components of other products. While invalidating previous products is not a major concern in a program such as AubieSat, in which all products are made at separate times, it is still important not to modify the connector if at all possible, as several components on both sides of the interface may rely on a certain operation or behavior by the core connector.

Core Connector Description

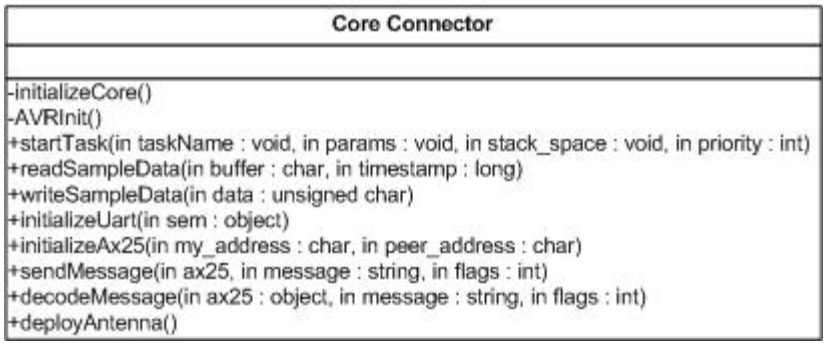


Figure 8

The private functions in the core connector represent the functions contained in the core connector that are used only by components in the core, but are still related to several components inside the core. Among these functions are the operating system initialization and the control board initialization. These functions are performed within the core and set up the system so that it is ready to operate correctly, including such tasks as initializing mailboxes and semaphores as well as ports on the control board.

The public functions are those that are provided to the product specific components to access the core component. Specifically, these operations correspond to the functionality found in the core components. For example, the create, read, and write file operations provide access to the file system found in the core. The networking component provides a send message

operation, and a general start task operation allows the product to launch any necessary tasks using the embedded operating system. The usage of the core connector can be thought of in the manner of the façade design pattern [Gamma et al 1994], in which a single interface is used to call the software in several different pieces in order to separate the implementation of components. Some components, such as the beacon transmission and secondary command receiver, do not provide any public operations through the core connector, but are contained entirely within the core. This is to be expected, as these particular components do not provide basic operations that might be used by satellites and do not require much communication with other components in the system.

4.3.3 AubieSat-1 Product Specific Architecture Description

In order to completely represent the architecture of each satellite in the product family, some portion of the architecture must be dedicated to the components found only in each product. These components are designed in the same manner as the core component, and there interaction with the core is carefully modeled through the core connector. The product specific architecture for AubieSat-1 is shown in figure 9.

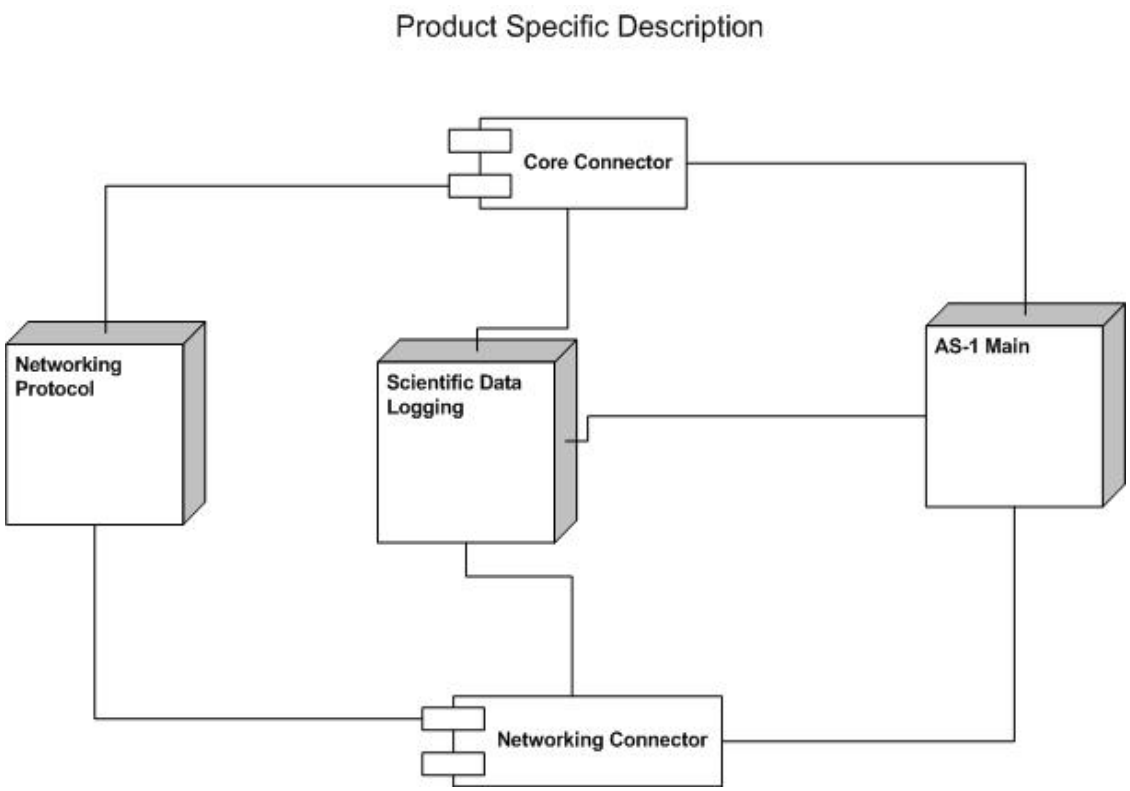


Figure 9

AubieSat-1 is the first satellite in the AubieSat product line. As described in the requirements for this mission, the main goal of this satellite and mission is to successfully achieve a satellite flight performing only the most basic flight operations, such as communicating with the ground station and recording and storing measurements. Because of this, the product specific architecture for AubieSat-1 is not as complex as future satellites may have. However, future missions can still benefit from the design and implementation of AubieSat-1. The most important reason for designing the components in the core and the product in the same manner is to allow for the most possible reuse. If it is later determined that a product specific component from one mission is required for another, that component can easily be reused in that mission without modification. The same method can be used to migrate product specific components into the core if future mission requirements seem to make this reasonable, or even to move a

component from the core to the product specific architecture in the case that it is no longer required by all products in the line.

The networking protocol component contains the portion of the network stack that applies only to AubieSat-1. This software module contains the actual messages transferred between the satellite and ground station. This module interacts with the core to send the messages that it creates. Although the messages may need to be changed for future products, it is expected that the basic design and code of this module may be reused in future missions if the messages themselves can easily be modified without changing how the networking stack as a whole operates. Because this portion of the network stack is contained in this product specific module, other products in the line can use this module as a starting block, add, modify, or remove messages as needed, and replace it with the new, changed module in order to easily and quickly create an entire functioning network communication relying mainly on the code contained in the architecture core. This component is also responsible for processing received messages from the network component in the core, since it contains the application layer message information needed to execute the received commands.

The component most specific to this mission is the scientific data-logging component. This mission is required to obtain data from several analog to digital converters located on the satellite, store the data, and send the data to ground station when requested. While the core component provides basic access to the file system, the data-logging component provides a high-level application programmer interface for use by the main component to read and store the sensor values. It also provides access to the analog to digital converters that take the data samples to be stored. This component interfaces with the file system through the core connector and also the network protocol component to send data and respond to commands. When a data

request is processed by the networking and network protocol components, the data-logging component is responsible for retrieving this data from the file system component and sending it back to the ground station using the network.

The final component required for the product specific architecture is the AubieSat-1 main component, which provides the main mission requirements of AubieSat-1. This component requires communication with several of the other components in the architecture, along with extensive use of the core component through the core connector. It constitutes the main thread of control from the mission perspective, while all other components perform the support or secondary operations required to make the primary task successful. The main component holds the main function used for application startup and handles initialization of the system as well as coordinating execution and communication between several of the other components. This component will not be reused in other products, although there will be a similar module in each satellite that will feature much of the same design, only handling the different mission requirements for that satellite.

4.3.4 AubieSat-1 Combined Architecture Description

Table 1 shows the tasks found in the system, the file in which each was located prior to the CBAPF design, and the new component and file location of each of the tasks after the new architecture design and implementation.

Task Name	Original File	New Containing Component	New File
Network Control Task	Test.c	Network Protocol (Product)	network_control.c
Watchdog Task	N/A (not implemented)	Watchdog Timer (Core)	Housekeeping.c
AX.25 SendInfo Task	Test.c	Network Protocol (Product)	network_control.c
Audio Beacon Task	Test.c	Beacon Transmission	AudioBeacon.c
Data Logging Task	Test.c	Data Logging	science_data.c

Table 1

Figure 10 shows the AubieSat-1 combined architecture description. This shows the entire software architecture of the AubieSat-1 satellite, including the communication and dependencies between the software components. The core connector in the middle of the diagram separates the core components at the bottom from the product components at the top, just as the connector separates the component in the software implementation. Each component in the core also includes a version number, which shows which version of this module currently exists. This allows modules to be interchanged with the same name and behavior but still be annotated as different implementations in the software architecture of each product. Since this is the first product in the family, each version number matches (1.0), but other products may choose to use a mixture of different versions of the core components. Each version must be compatible with all other versions of the other components as the core software evolves. As discussed, the use of message passing and interfaces allows this to be achieved.

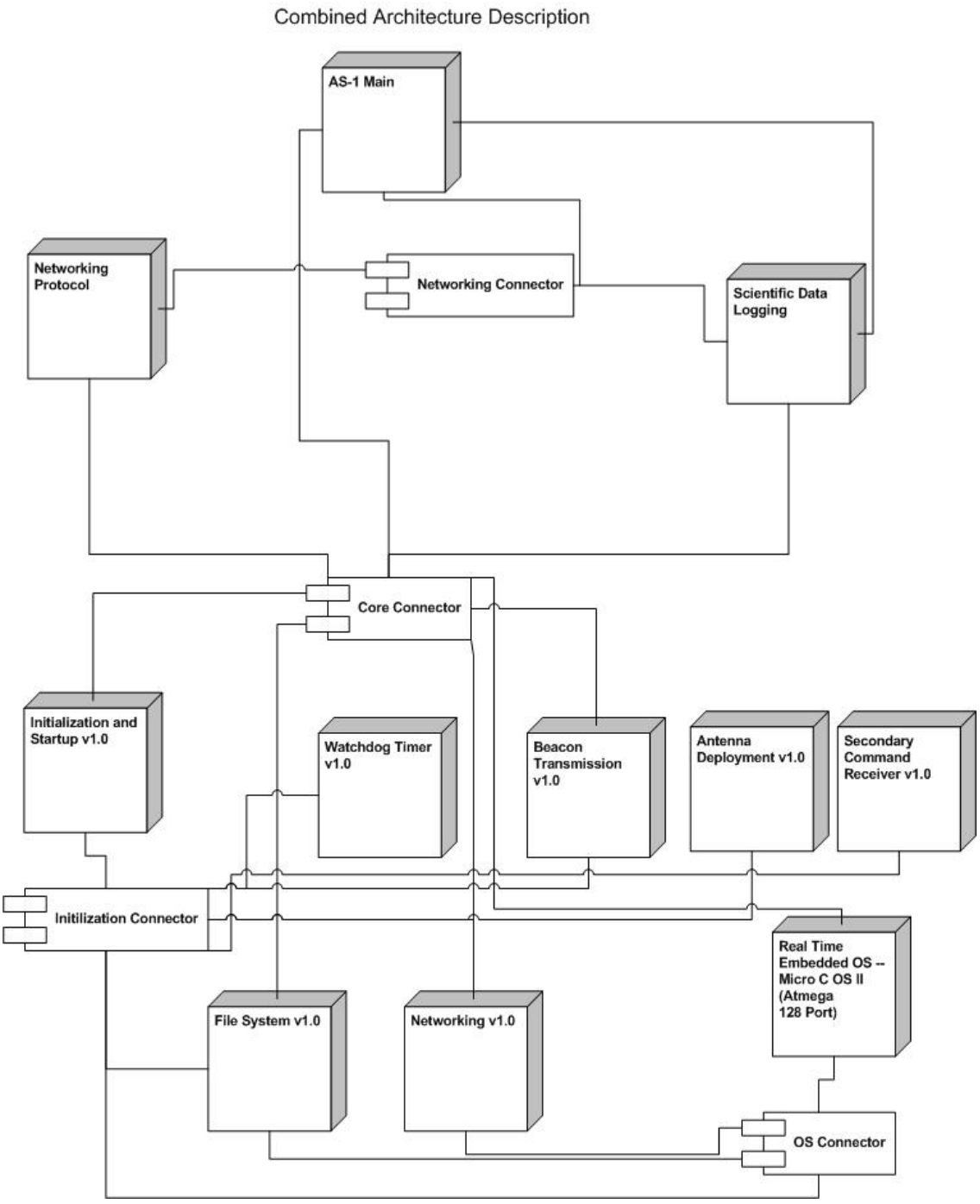


Figure 10

One advantage of the combined architecture description is to allow for architecture verification and validation of the software design. This document allows traceability to the implementation and to the requirements of the system. While the detailed documentation shows

the mapping to the architecture, the architecture can also be mapped to the requirements. Table 2 shows such a mapping for a subset of the AubieSat-1 requirements, as an example of how such a mapping can be performed using this part of the CBAPF product family architecture. This activity can take place for each satellite separately in order to assure that the core and product specific components have been combined in an appropriate way to allow the satellite to correctly achieve mission success.

Requirement	Component Implementing
CDH 2.1, 3.1	Networking
CDH 2.2, 3.2	Secondary Command Receiver
CDH 7.1, 7.2	Beacon Transmission
CDH 8.1, 8.2, 8.3	Watchdog Timer, Networking
CDH 1.1	Antenna Deployment
CDH 1.2, 9.1	AubieSat-1 Main
CDH 4.1	Networking Protocol
CDH 5.1, 6.1, 6.2	Scientific Data Logging

Table 2

4.4 AUBIESAT-1 DETAILED DOCUMENTATION

There is not an extensive amount of detailed documentation based on the AubieSat-1 code. Because the implementation does not follow an object-oriented programming style, the documentation artifacts typically found in those projects do not apply, such as UML based class, sequence, and state diagrams. However, there are still several diagrams that can show the design at an implementation level. Included in this section is part of the detailed documentation of one

component from the CBAPF architecture for AubieSat-1. This represents one way of creating detailed documentation in accordance with the CBAPF process. Although no particular style or methodology for software implementation design is required, this method is useful because it provides excellent traceability between the software architecture and the detailed design. In this method, each component has been broken down into the several files that it contains, with each file showing its operations, using a UML class diagram.

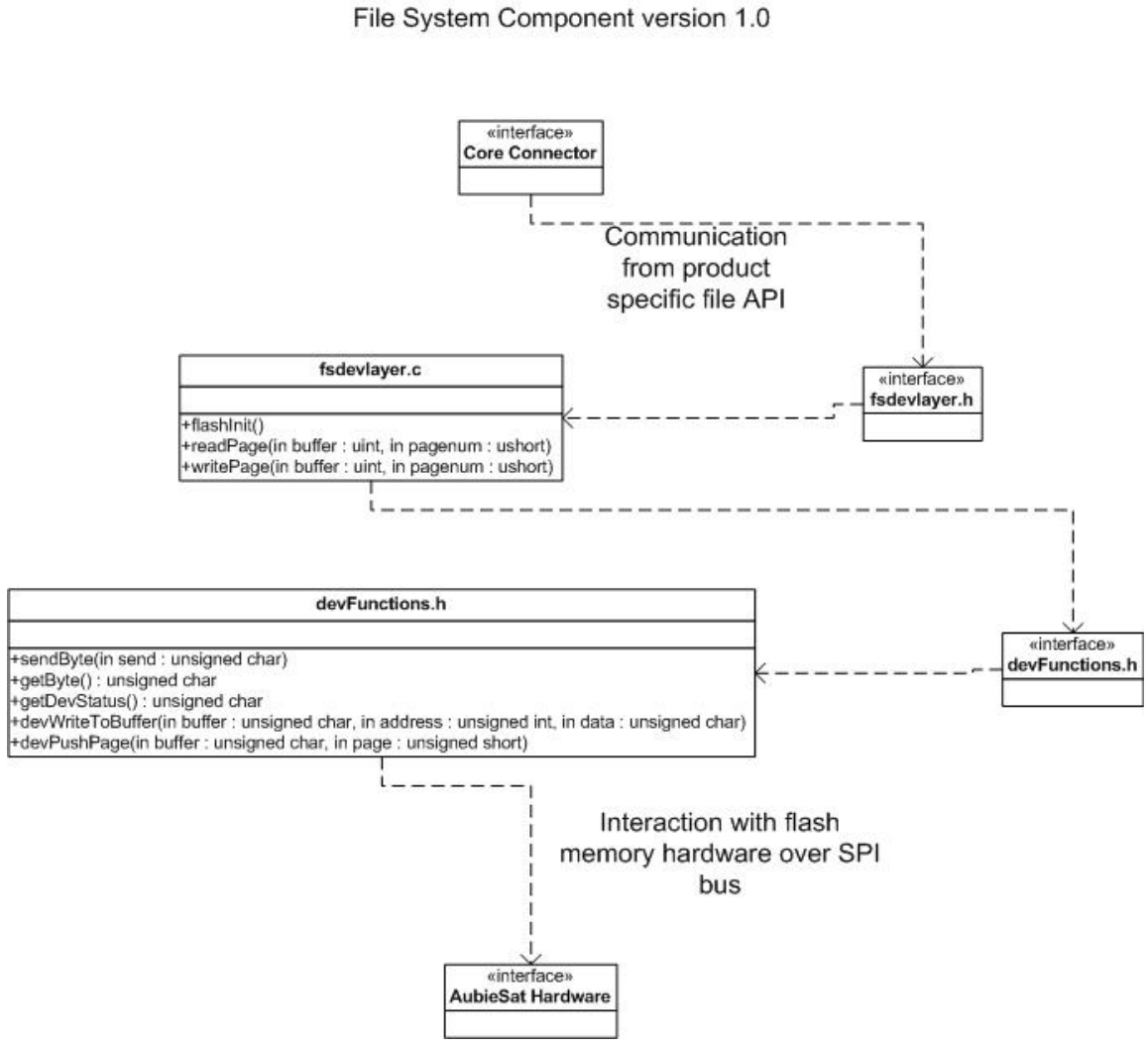


Figure 11

Figure 11 shows the static detailed programming diagram for the file system component found in the AubieSat core component. This can be used in conjunction with the requirements

mapping for this component in order to establish a high amount of traceability from the detailed design through the architecture to the software requirements. In this case, we can see that the requirement for storing measured sensor data is fulfilled by the file system component, inside which the writePage function of fsdevlayer.c and the sendByte function of devFunctions.c are used to fulfill this requirement. From here, all that must be done is to trace this design to the actual code in order to perform a thorough form of verification that the requirements are correctly implemented in the system. The detailed documentation for each of the components from the AubieSat-1 combined architecture can be found in Appendix B.

4.5 AUBIESAT-1 IMPLEMENTATION

The implementation of the AubieSat design was done in a reverse engineering fashion. This is a very important way to use the CBAPF method, as there may be several product lines with several products already developed that could benefit greatly from the formality of design that the method offers in order to increase the traceability of the documentation and to allow for the reuse of large portions of code between the different products. Because of the nature of this approach, the design was completed individually without communicating with an entire software development team, which affects the implementation of the method from a software process standpoint. This strategy can, however, be used equally as well when designing a new product line from scratch. As stated in the CBAPF definition, one of the goals of the method is to successfully support the communication of several teams during the software development, and the implementation for AubieSat could easily extend to a team of designers. In fact, this design allows several teams to modify the documents in the future. It is believed that the design and implementation of the software core and satellite software to separate student led development teams for future missions due to the separation provided by the software architecture.

For the architectural and detailed design, Microsoft Visio was used to create and express the documentation. Although no sharing of the documents was required for this effort, any number of standard available team document sharing technologies could easily be employed to assist in the automation of documentation and source code version control. No custom software was required in order to implement the design of the system, and the approach provides high consistency between documentation easily with simple and common configuration management in place. Although not part of the author's research for AubieSat, it is believed that the approach used here would scale well to larger organization because of its reliance on standard tools and modeling languages.

Chapter 5: AubieSat-1 Ground Station Software Architecture

Although there is currently only one AubieSat satellite in production, there are two separate products in the program. In addition to the software for the satellite itself, there is a separate software product that runs on the ground station that sends commands to and receives data from the satellite during its mission. Since both must communicate using the AX.25 networking protocol, there is a significant amount of code reuse between the two. Although this does not fit the strict definition of a product family in the CBAPF since because it does not make use of an identical set of core components, the software architecture for the ground station can be modeled using the same core based methodology. While the core is a little different, the main differences come in the product specific documentation.

5.1 AUBIESAT GROUND STATION PRODUCT SPECIFIC ARCHITECTURE DESCRIPTION

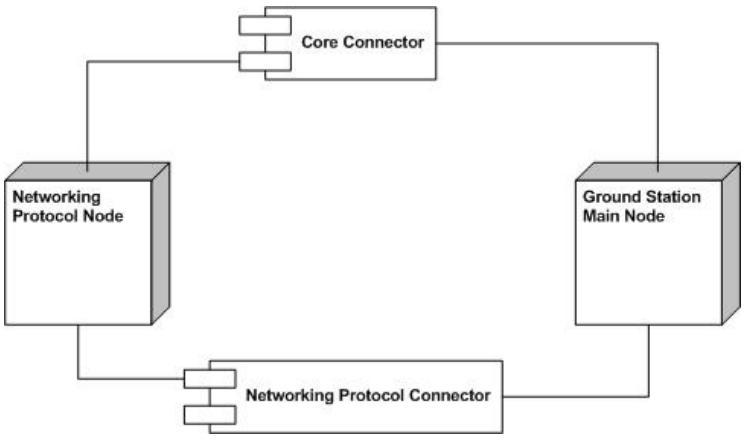


Figure 12

Figure 12 above shows the product specific architecture for the AubieSat ground station. There are similarities between this architecture and the AubieSat satellite architecture, since both are responsible for communication with the other. The networking protocol node in this diagram is the same component used in the satellite, which is simply configured differently based on which application it is being used for. This allows the most code reuse between the two and all the benefits that accompany this reuse, namely less testing and debugging as well as more confidence in the code operating correctly in both products. The ground station does not need to access the entire core software, since much of this is based on the satellite family and the embedded environment in which it operates. The portions of the core used are explained in more detail in the combined architecture description for the ground station.

5.2 AUBIESAT GROUND STATION COMBINED ARCHITECTURE DESCRIPTION

The AubieSat ground station source code is a subset of the full satellite command and data handling source code. As such, it contains portions of the core component and a separate product specific design from the satellite software architecture. Figure 12 displays the combined architecture description for the ground station software for the AubieSat-1 mission. This document shows how different products can interact with the same core component using the core connector. This is the same core connector and components from the AubieSat core architecture description. The only difference here is that several components have been removed from the core that are not required and do not apply to a desktop computing environment for the ground station, such as the Micro-C OS II operating system and the watchdog timer.

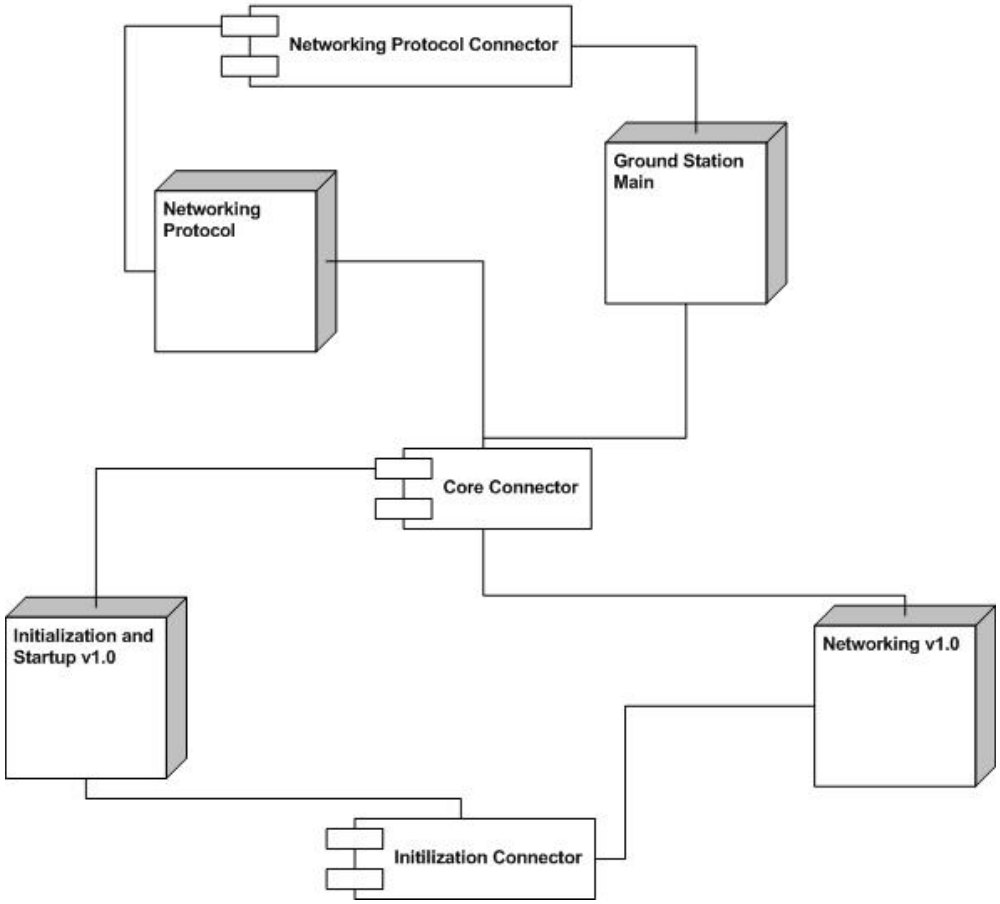


Figure 13

In this software architecture, the networking protocol component is specific to this version of the ground station and contains the protocol and messages needed to communicate with the command and data handling software running on the satellite in space. In order to send and receive the messages, the networking component found in the core is used. Because identical hardware is used in both the satellite and the ground station (serial port through a TNC, which broadcasts the signal using amateur radios), the same low level networking code is used in both with only very small differences between the two that are resolved using simple constant definitions. The configurable header files that determine which portion of the code to use from

the core and networking protocol are found in the ground station main module, along with the main function and thread of control for the system.

This architecture shows how the CBAPF architecture can assist in allowing for large amounts of code to be reused in a safe and correct manner. In this case, even when the two products are not strictly in the same line but are closely related, we are able to leverage the methodology to reuse the same code across both products. Since the ground station required the same AX.25 networking stack to be implemented for the ground station as the satellite, the author was able to recognize the similarity and realize that the ground station could benefit from using the same core modules found in the satellite by simply removing the components specific to embedded software from the core component. While ideally the core component would not be modified in any way from one product to the next, this is a valid use of the CBAPF method. As long as components are removed from the core and not added to it for the sake of a single product, the core can still be used in its original form in other products in the family. Often in real applications of a methodology, certain compromises must be made so that the method can be used in ways not originally anticipated and applied to a wide range of systems.

The combined architecture description also shows the benefit of using this method to practice code reuse when the products are closely related. If an ad hoc design approach is taken, it would still be possible to reuse the code in much the same manner. With the formality provided by the combined architecture design, however, the architect can be confident in the correctness of the design and therefore the system itself. As demonstrated in the command and data handling combined architecture, it is much easier to trace the requirements to the architecture and then to the design and source code when a formal method such as this is used. While other formal methods might provide this same benefit, CBAPF gives the formality while

still focusing on the partitioning of components with an emphasis on reuse that is tailored specifically for application to these similar applications. The use of this method provides the best of both worlds, allowing the architect to easily identify which components should be included for reuse and then helping the architect to validate and verify the architecture according to the requirements for each product in the family independently.

Chapter 6: AubieSat-2 Software Architecture Design

Although the AubieSat program at Auburn University is still developing its first project in the family, a hypothetical design for a second satellite is included here. This project is designated as AubieSat-2. This design will demonstrate how the CBAPF process and methodology can be applied in order to effectively leverage the core architecture from the AubieSat-1 project. The requirements and design for AubieSat-2 have been invented to represent possible actual requirements for future AubieSat missions.

5.1 AUBIESAT-2 COMMAND AND DATA HANDLING SOFTWARE REQUIREMENTS

For the hypothetical second mission for the AubieSat program, several pieces of new functionality will be required by the command and data handling software. First, a camera will be added to the satellite. The onboard software must be capable of powering on and off the camera, taking a picture in a certain direction, and storing and retrieving the pictures based on timestamp or location of the picture. The hardware will be responsible for supplying the power to the system and making sure that the camera operates properly in space, while the software will be responsible for controlling its operation.

Another additional requirement will be to use solar panels to help supply power for the satellite. The concept here is to develop a system that will deploy solar panels in space then measure the amount of power captured during different points of the flight in order to determine how much power is gained during different points of orbit. After its integration into AubieSat-2, solar panels will be used in all future missions to provide extra power to the satellite as it grows

in complexity and the power needed increases. This mission will demonstrate the correct operation of the solar panels from a hardware and software perspective to prove that this can be relied on as a main form of energy for future missions.

This mission will also add two separate thermistors to the satellite. The first will be an external thermistor connected to the hardware. The software is responsible for measuring the temperature once every five minutes and storing this data in the file system to be transferred to the ground station upon request. Another thermistor will be integrated into the satellite itself to measure the temperature of key components in the system. This hardware is actually already included in the AubieSat-1 hardware, but are not monitored or controlled at the software level, which makes for a good approximation of actual requirements of a second mission. The software is then responsible for monitoring these temperatures and applying controls in real time in order to correct overheating conditions in order to preserve the hardware in optimal working condition for the longest possible time period.

In addition to these new requirements, the satellite must also be able to perform all of the requirements from AubieSat-1 that are required for the satellite to operate correctly and support the explicit requirements for this mission. The satellite is no longer required to record and store measurements from analog-to-digital converters, which was the main mission goal from AubieSat-1. The mission will be required to communicate with the ground station, to store files in a file system, to beacon its position at a certain time interval, and to provide a secondary form of communication as was required in the original mission. Figure 14 shows the AubieSat-2 command and data handling software requirements, using the same approach from the previous mission.

AubieSat-2 C&DH Software Requirements

Core Requirements

These requirements should be common for any satellite produced in the AubieSat program.

1. CDH2-00 – CDH shall implement all core requirements from AubieSat-1 mission.
2. CDH2-01.1 -- CDH shall deploy solar panels 10 minutes after entering orbit, but not before antenna deployment is complete.
3. CDH2-01.2 – CDH shall have the capability to determine energy gained from solar sensors.
4. CDH2-01.3 – CDH shall have the capability to store energy from solar panels for use by other systems. CDH shall support determining power level remaining and activating and disabling solar panels as a power source.

Mission Specific Requirements

These requirements are specific to AubieSat-2.

1. CDH2-01.4 – CDH shall record the amount of energy absorbed by the solar panels once every two minutes and store the recorded information for later transfer to the ground station.
2. CDH2-02.1 – CDH shall control a digital camera through software. Allowed operations include power on, power off, rotate camera, and take picture.
3. CDH2-03.1 – CDH shall support the measuring of external temperatures through the external thermistor.
4. CDH2-03.2 – CDH shall record the external temperature once every 5 minutes and store this information for later transfer to the ground station.
5. CDH2-04.1 – CDH shall support the measuring of onboard temperatures through the internal thermistor.
6. CDH2-04.2 – CDH shall record the internal temperature in several different regions of the satellite once every 30 seconds.
7. CDH2-04.3 – CDH shall alter the operation of the satellite to reduce the internal temperature in the case that it is beyond safe operating conditions. This may include temporarily stopping a task from execution or disabling hardware.
8. CDH2-05.1 – CDH must be able to pass data to COMP in same format that can be read by GS in accordance with specifications in Aubie Sat CDH onboard software architecture.

Figure 14

6.2 AubieSat-2 CBAPF Documentation

6.2.1 AubieSat-2 Core Architecture Description

The core architecture description is the heart of the CBAPF methodology, and we see from this example how useful it can be in establishing commonality between similar products in the same line in order to quickly provide overlapping functionality to new products. As stated in the AubieSat-2 requirements, the second mission is required to perform all of the operations performed by the first test mission in addition to several new pieces of functionality. The best-case scenario would be to use the same code from the original mission again to perform this functionality for the second mission, and this is exactly what we attempt to do here by leveraging the CBAPF architecture method. We have already seen from the AubieSat-1 architecture which

requirements are performed by each component within the architecture. By using this methodology, we have already separated the components and included those that would be required for all products in the line in the core architecture. By including the core architecture from AubieSat-1, we are able to support several of the key requirements of the system, such as storing data files with timestamps, providing an audio beacon, implementing a watchdog timer to avoid runaway tasks, and several others.

It is important at this point to review the core architecture to be certain that the design is correct for the product being designed. In this case, we must include all of the components from the core architecture in order to support the requirements for this product. There may, however, be a case in which the product does not require some of the components from the core architecture. It is generally advised in this situation to include these components unless there is some compelling case to remove the component from the system, e.g. memory requirements prevent proper functioning with the inclusion of these components, or to reevaluate why these components are included as part of the core architecture and if they should be removed from the core altogether. This situation should be very unlikely in the domain that this method is generally applied if the core component is designed correctly. This example shows how important it is to use CBAPF and to focus carefully on correctly designing the core architecture such that it can be included into several different products in the line without modification. Since most product families have several products that share these same characteristics, just as the AubieSat case study shows, a well-designed core architecture should be able to be implemented in several products without modification to the architecture or code of the components contained in the core.

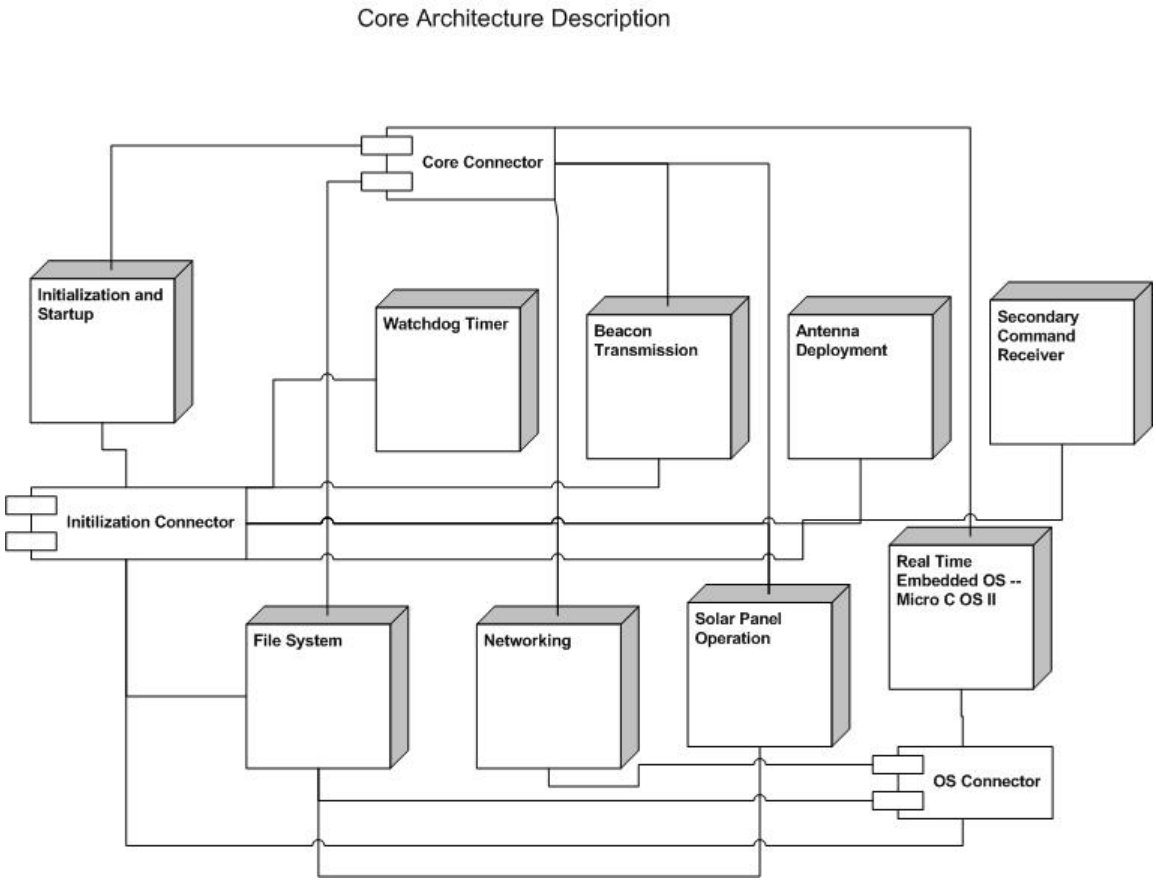


Figure 15

Figure 15 shows the core architecture description for AubieSat-2. This core architecture is the same as the core architecture from AubieSat-1, with the addition of one new module. The solar panel operation module has been added in this core architecture, due to the fact that it can be identified as responsible for part of the core requirements for all products in the system. The solar panel module interacts with the file system module and initialization connector within the core itself. It also interacts with the core connector in order to provide the ability to perform product specific actions using the solar panels, such as recording the current amount of energy being absorbed by the panels, or accessing the total power available.

Although not included in the first mission, the product line planning states that solar panel operation will be required for all future missions, and so this will be needed as part of the

core architecture that can be reused from one mission to the next. This shows how the core itself can evolve over time in certain product families that employ a staggered release of separate products. In other families, there will be a plan for several products to be released simultaneously with different requirements and functionality. If for instance, the AubieSat-1, AubieSat-2, and AubieSat-3 missions were developed simultaneously, this module would need to be included as a product specific module in the AubieSat-2 and AubieSat-3 design and code structure. Similar situations may occur in other product families as discussed in the introduction section, such as cellular phones developed simultaneously or over time, or a professional and standard edition of an application suite or operating system.

Another important thing to note here is the ability of CBAPF to support changes to the core without affecting the product specific portions of the design and code. If for instance, the software team decided that the file system from AubieSat-1, although adequate and performing to specifications, should be upgraded to a more robust file system with better handling for errors and to support more storage space, the implementation could easily be changed without changing the components that rely on the file system. This can be achieved due to the modular black box design that is used in the core architecture. As long as the new implementation conforms to the specification and interface, the way that it is implemented will be invisible to those components that must store and retrieve files. By basing the other components on the high level interface represented by the core connector, the design is flexible enough to allow several different modules to be used for the file system for the system. By defining the core architecture, however, we are able to recognize that some type of file storage and retrieval is needed, and must be covered by some component in the core. In this way, CBAPF can support both code reuse

from one product to another and modification and upgrading core components at the discretion of the software architect, software engineers, and developers.

6.2.2 AubieSat-2 Core Connector Description

Much like the core architecture, the core connector will remain largely unchanged from the AubieSat-1 design to the AubieSat-2 design. Because the connector provides an interface to the core component, it is very closely tied to the components that make up the core architecture. Just as with the core architecture, it is crucial that the core connector remain stable and the interface unchanged for each of several products in the product family. In order to guarantee that objects will interact with the core in a predictable and expected manner, the core must have well defined functionality that can be easily accessed using the core connector. For this reason, all of the functionality provided by the original core connector is left unchanged in the core connector of AubieSat-2.

Core Connector Description

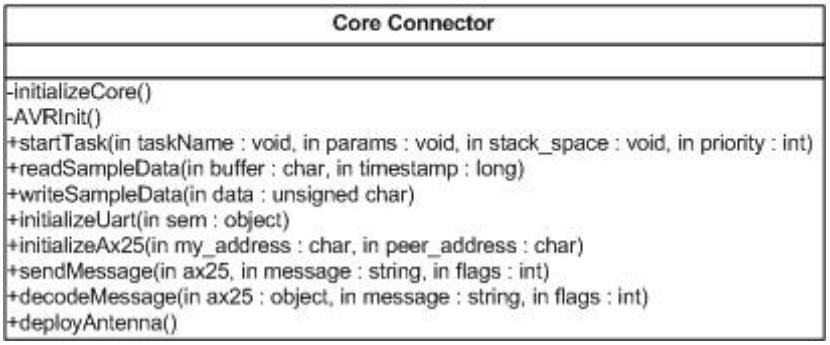


Figure 16

Figure 16 shows the core connector description for the new mission. The new functionality provided by the solar panel operation module can be accessed through the interface operations record_solar_measurement, get_total_power, and get_remaining_power. These operations may be required for different missions. For instance, the AubieSat-2 mission

objective is to record the amount of power gained from the solar panels at different times, while future missions will require the total and remaining power in order to appropriately use the power collected to operate the satellite components.

6.2.3 AubieSat-2 Product Specific Architecture Description

From the requirements for AubieSat-2, we are able to discern several requirements that are specific to this mission alone and therefore will only be needed in the current satellite and not others in the program. These are a mechanism for recording internal temperatures, a way to record external temperatures, and a way to control and operate a digital camera connected to the satellite. As part of the software architecture process for CBAPF, we must design modules in the product specific portion of the design that will be able to perform these functions. Figure 17 shows the product specific design for this AubieSat-2 mission. It is obvious that the camera control operations should be separated into one component that will interact mainly with the core connector and the camera hardware. This module is shown in the diagram and can be used to power the camera on or off, move the camera to point in a different direction, and take a still picture. The design of the thermistor reading portions is less obvious. There may be several shared code operations between the two since both are thermistors, however, there are also key differences in that the internal temperature must be used to control certain aspects of the satellite software and hardware and sample more frequently than the external temperature, while the external temperature is simply stored in the file system and transmitted to the ground station when requested. For this reason, the design shown here breaks the temperature recording into three separate modules, one for recording and accessing general thermistor properties, one for controlling the monitoring and storing of external temperatures, and one for monitoring the internal temperature and modifying the satellite operation accordingly.

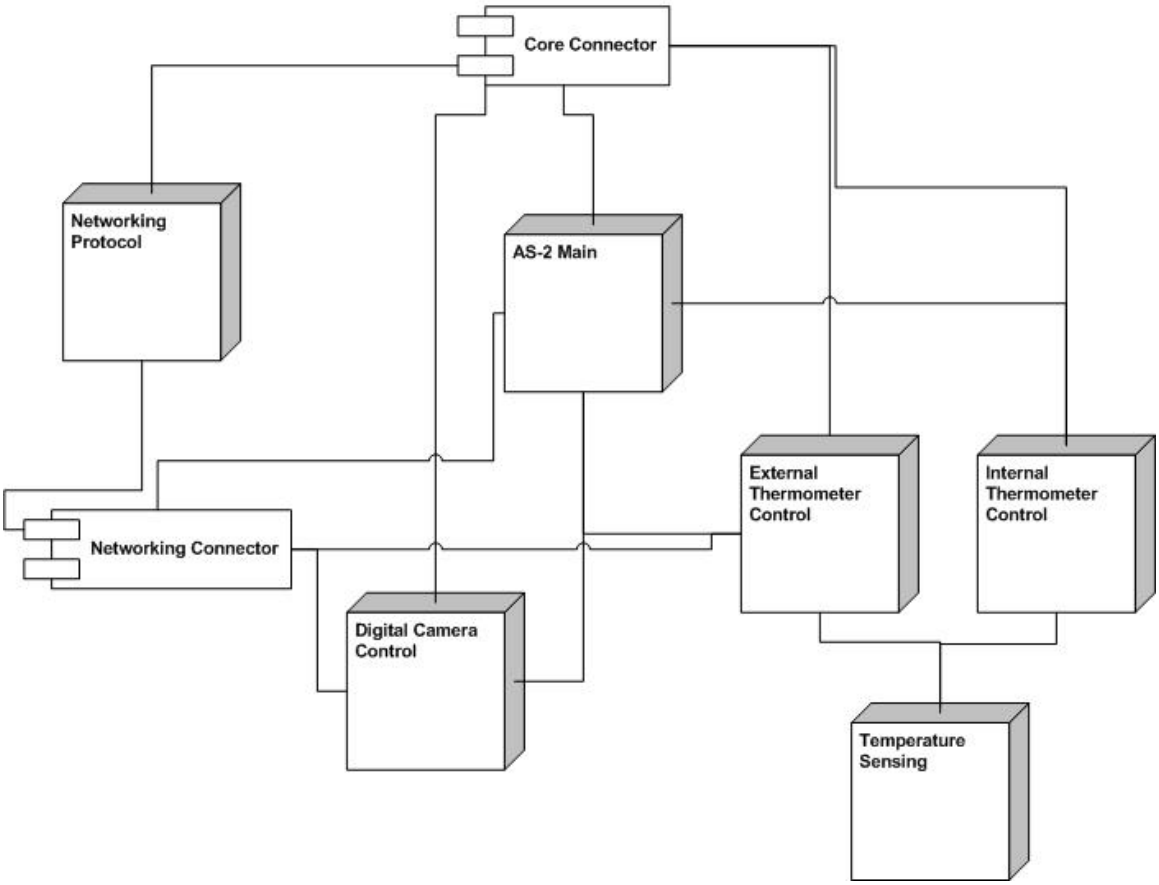


Figure 17

In addition to these explicitly listed requirements, a few other product modules are required to support the correct operation of the satellite. First, the main module is created, which will act as a controller for the other components and coordinate the activity of the satellite, including setting the priorities of the tasks in the system and assisting with inter-task communication. Next, the network protocol module is included to provide the protocol for sending and receiving messages from the ground station. This module will be similar to the corresponding module in the AubieSat-1 design, but the implementation will be different because of the need for different messages, such as retrieve picture, take picture, retrieve

temperature, or other operations needed for this mission. The module design and code however will only need to be modified from the original component instead of being created anew. This shows the motivation for keeping the product specific architecture in a modular design in that it will allow the parts specific to the protocol for AubieSat-1 to be easily identified and modified for inclusion in the new satellite while having confidence that no incorrect messages will be found in other parts of the system.

6.2.4 AubieSat-2 Combined Architecture Description

The combined architecture description allows the architect to verify the architecture by tracing each of the requirements to a specific component within the design. The combined architecture description for the AubieSat-2 mission is included in figure 18. Here we can see the combination of the core and product along with their interaction through the core component. The components from each part work together in much the same way as in the AubieSat-1 design. The main component is responsible for controlling the tasks and initializing the system through the core connector. When it calls the initialize operation using the core connector, the initialization and startup component will manage the components in the core to initialize and start the operating system, deploy the antenna, set up network communications, initialize the file system, and deploy the solar panels. The initialization module shown here is listed as version 2.0, as it has been updated to include the control of the solar panels as part of the initialization process.

When the system is correctly initialized and has entered full operating mode, the main component will initiate the mission tasks, such as controlling the digital camera and interacting with the temperature measurements. These components interact with the core component through the connector to fulfill the requirements of the system. In order to send messages

through the network, they interact with the networking protocol module, which will format the message correctly and pass it to the core to be sent to the hardware using the AX.25 networking stack. Inversely, received messages will first travel to the protocol module to be decoded, then passed to the main module where they are identified and passed to the appropriate module and task to handle the command. This document makes it easy to recognize such information flow within the system and be able to identify the cohesion and coupling the system to make improvements on the design and to trace each requirement to a certain part of the system at a high level of abstraction.

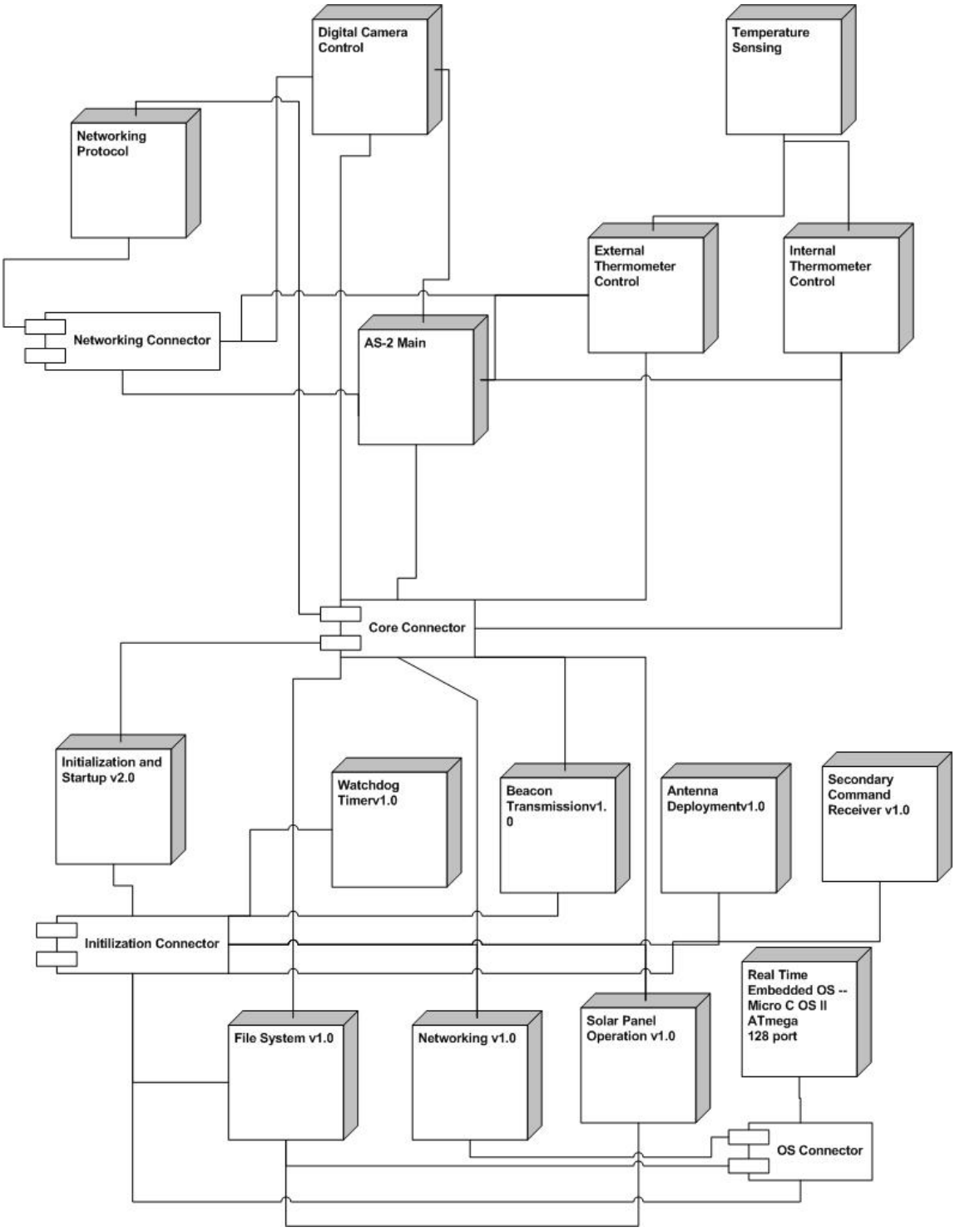


Figure 17

Using the same method as the AubieSat-1 design, we have included a traceability matrix in Table 3 that defines where each software requirement is implemented according to

components. The AubieSat core implements all of the requirements from AubieSat-1, as can be proven from the original design and traceability matrix. All other requirements are linked to specific modules added for the purposes of this mission.

Requirement	Component Implementing
CDH2 00	AubieSat-1 Core
CDH2 01.1-01.5	Solar Panel Operation
CDH2 02.1	Digital Camera Control
CDH2 03.1-04.1	Temperature Sensing
CDH2 03.2	External Thermistor Control
CDH2 04.2-04.3	Internal Thermistor Control
CDH2 05.1	Networking Protocol

Table 3

6.3 AubieSat-2 Detailed Documentation and Implementation

The detailed documentation for the majority of the modules used in the total design for AubieSat-2 will be the same as that from AubieSat-1. Since we are simply reusing these modules without modification, none of the detailed design or source code will have to change. This is important because it allows the product specific design and code to be created very early in the software development lifecycle of this satellite. Instead of beginning with the basics of how the modules will interact at the source code level, the product specific team can simply identify the detailed documentation for the already completed modules to see exactly how each will function and what will be required in order to use these modules. In this section, we will

show the detailed design for the new components found in the AubieSat-2 design. As in the previous design, we show one detailed class diagram for each module in the architecture.

Figure 19 shows the detailed design for the solar panel operation component, which is part of the AubieSat core component. This module provides certain operations to the product specific portions, such as the main module, that allow the module to control and interact with the solar panels. The deploy and retract operations are controlled by the initialization component in the core, while the main module is able to monitor and record the power absorbed by the panels. This design is consistent both with the requirements and core based architecture design.

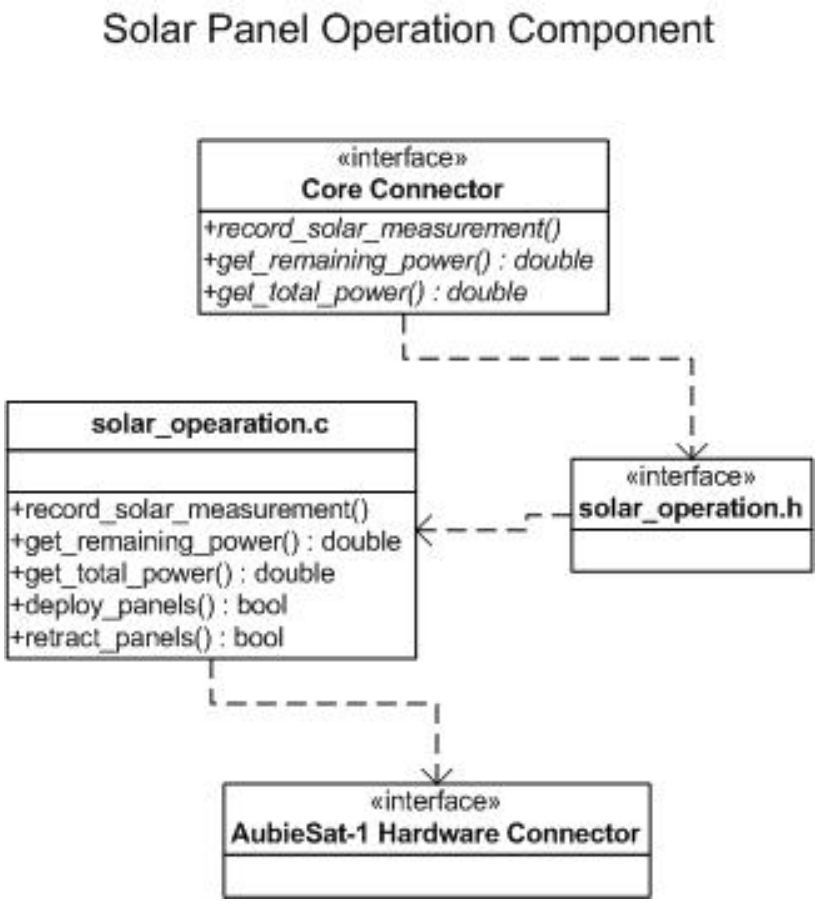


Figure 19

On the product side of the design, the digital camera control component resides in the product specific portion and communicates with the core component to accomplish its goals. While it provides operations for rotating the camera and taking a picture, it relies on the file system through the core to store those files. It also accepts commands from the networking protocol module and sends commands back to the ground station using the core component. Figure 19 shows how the design is broken down into functions and files in the C code. More detailed design for the components from the AubieSat-2 design can be found in Appendix C.

Just as with the AubieSat-1 design, this design was implemented using standard well-used tools. Microsoft Visio was used to create the documents. Since there was only one software architect, no configuration management or collaboration tools were required that generally would be used on this type of project. However, the choice of tools would be left to the development team, as the methodology would support any type that might be selected. This is part of the flexibility demonstrated by CBAPF that allows it to be applied to several different software organizations without requiring any proprietary tools. From a business and software process perspective, using standard tools and techniques eases the strain that the process places on a software development organization, which is very important for many organizations and will increase the chances that the methodology is adopted by a wide range of organizations.

Chapter 7: Conclusion

The ability to reuse software at a high level is a goal of many facets of the discipline of software engineering. With each software failure that occurs in industry and government, the need for a disciplined engineering-based approach in the entire software development lifecycle is apparent [Lyu 2007]. Because of its use early during the engineering process, software architecture offers a valuable way of reusing source code in different software products. The focus in this work has been on establishing a software architecture methodology for groups of related software products, called software product families. Although several methods exist for performing this task, each either lack some important qualities that would allow the method to perform best or are tailored to a specific type of software products [Matinlassi 2004]. In developing the Core Based Architecture for Product Families (CBAPF), the main goals were to allow for as much software reuse as possible between software products in the same family, to create a method that could be applied to any software product line, and to allow the method to be easily used in a typical software development environment featuring many software developers and designers working in coordination to implement a set of software systems.

In order to demonstrate the ability to support reuse, the methodology was applied to the AubieSat satellite program at Auburn University. The embedded command and data handling software that runs on the satellite was modeled using the technique, and the effort was successful in formally defining high-level software components that will be reused in each of the several planned AubieSat missions in the future. The focus of the method on identifying and modeling a core set components that will be the basis of each product proved to be the main strength of the

methodology. By modeling the ground station software as a separate product in the family using the same core component and connector, the method proved its benefits in easily representing the source code in multiple products using the software architecture domain. In order to better demonstrate how the design could be used in future missions, a hypothetical AubieSat-2 mission was used to show how the CBAPF design would be implemented and how the code from the core could be reused in this mission.

The entire design was traced from the requirements through the architecture to the detailed software design and implementation using this method to show the value of using a formal design for the software architecture. Even in the case where software is already reused between the different products, applying the CBAPF methodology could still be beneficial by providing a formal framework in an easy to implement manner. The traceability and formalism that the method provides helps software architects and engineers to validate the software product. The method is also flexible in its ability to capture the design of a software product line. The AubieSat case study shows its application to an embedded software system, but the flexible methodology definition allows its application to any software product family.

From the software process and implementation perspective, the methodology achieved its goal of using only standard modeling languages and tools, as UML and Microsoft Visio were used to create the design. The methodology stresses the importance of easy application of the design across several software engineers and teams. By leveraging standard documentation techniques, the CBAPF methodology is able to support a wide variety of software tools for implementation of the design, coding, and configuration management. This approach also reduces the amount of training required for software engineers that work on the products.

The CBAPF methodology meets all of the goals that it was designed to achieve. Its focus on the core of a software product family differentiates it from current software product family architecture types and allows it to produce a design that facilitates the reuse of code without modification in each of the several products. By identifying the strengths and weaknesses of each of the current software architecture methodologies, a software architecture method was created that was tailored specifically for this area and yet still uses standard documentation techniques and tools. The result is a software architecture methodology that allows for maximal reuse, traceability, and ease of implementation for software product families.

References

Abowd, Gregory D.; Allan, Robert; Garland, David. 1995. Formalizing style to understand descriptions of Software Architecture. *ACM Transactions of Software Engineering Methodology.* Volume 4. Issue 4. Pgs. 319 – 364.

Atkinson, C. et al. 2002. Component-based product line engineering with UML. Addison-Wesley, London, New York.

Balzerani, L.; Di Ruscio, D.; Pierantonio, A.; De Angelis, G. 2005. A product line architecture for web applications. *In the Proceedings of the 2005 ACM Symposium on Applied Computing.* Pages 1689 – 1693.

Chapman, Richard; Wersinger, J.M.; Wilson, Thor. 2008. Aubiesat-1: A Student-Designed Cubesat at Auburn University. *2008 AMSAT Space Symposium.*

Clements, Paul C. 1996. A Survey of Architecture Description Languages. *In the Proceedings of the 8th International Workshop on Software Specification and Design.* Pg. 16.

Clements, Paul C. and Weiderman, Nelson. 1998. Notes on the second international workshop on development and evolution of software architectures for product families. *ACM Sigsoft Software Engineering Notes.* Volume 23. Issue 3. Pages 39 – 43.

Dabrowski, Michael. 2005. The Design of a Software System for a Small Space Satellite. *University of Illinois- Urbana Champagne. Master's Thesis.*

DoD Architecture Framework Working Group. 2004. DoD Architecture Framework Version 1.0 - Volume II: Product Descriptions.

Eixelsberger, W.; Ogris, M.; Hall, G.; Bellay, B. 1998. Software Architecture Recovery of a Program Family. *In the Proceedings of the 20th International Conference on Software Engineering (ISCE '98) Lessons and Status Reports*. Pages 508-511.

Eixelsberger, W.; Warholm, L.; Klosch, R.; Hall, G. 1997. Software architecture recovery of embedded software. *In the Proceedings of the 19th International Software Engineering Conference (ISCE '97), Lessons in Organizations.* Pages 558-559.

Frankel, D. 2003. Model Driven Architecture, Applying MDA to Enterprise Computing. Wiley Publishing Inc., Indianapolis, Indiana.

Ferguson, R.C., and Thompson, H.C. 2005. Case study of the space shuttle cockpit avionics

upgrade software. *In the Proceedings of the 24th Digital Avionics System Conference.* Volume 2. Page 11.

Gamma, Erich; Helm, Richard; Johnson, Ralph; and Vlissides, John. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Upper Saddle River, NJ.

Garland, David. 2000. Software architecture: a roadmap. *In the Proceedings of the Conference on the Future of Software Engineering.* Pgs. 91- 101.

Gomaa, Hassan. 2000. Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley, Upper Saddle River, NJ.

Hofmeister, Christine; Nord, Robert; Soni, Dilip. 2000. Applied Software Architecture. Addison-Wesley, Reading, MA.

Jazayeri, Medhi; Ran, Alexander; van der Linden, Frank. 2000. Software Architecture for Product Families. Addison-Wesley, Upper Saddle River, NJ.

Kacem, Mohamed Hadj; Kacem, Ahmed Hadj; Jmaiel, Mohamed; and Drira, Khalil. 2006. Describing Dynamic Software Architectures Using an Extended UML model. *In the Proceedings of the 2006 ACM symposium of Applied Computing.* Pgs. 1245-1249.

K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. 1998. FORM: A Feature-Oriented Reuse Method with Domain- Specific Reference Architectures. *Annals of Software Engineering*, vol. 5. pp. 143 - 168.

K. C. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. 1990. Feature-Oriented Domain Analysis. Feasibility study. *Software Engineering Institute, Pittsburgh CMU/SEI-90-TR-21.*

Krutchen, Phillippe; Selic, Brian; and Kozacynski, Wojtek. 2001. Describing software architecture with UML. *In the Proceedings of the 23rd International Conference on Software Engineering.* Pgs. 715-716.

Kuusela, Juha and Savolainen, Juha. 2000. Requirements Engineering for Product Families. *Proceedings of the 22nd International Conference on Software Engineering.* Pages. 61 -69.

Leist, Sussane and Zellnor, Gregor. 2006. Evaluation of current architecture frameworks. *In the Proceedings of the 2006 ACM symposium on Applied computing.* Pgs. 1546-1553.

Lyu, M. R. 2007. Software Reliability Engineering: A Roadmap. In *2007 Future of Software Engineering* (May 23 - 25, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 153-170.

Magee, Jeff; Dulay, Naranker; Eisenbach, Susan; Kramer, Jeff. 1995. Specifying Distributed Software Architectures. *In the Proceedings of the 5th European Software Engineering Conference.* Pages 137 – 153.

Mannion, Mike and Kaindl, Hermann. 2001. Requirements-based product line engineering. *Proceedings of the 8th European software engineering conference.* Pages 322-323.

Matinlassi, Mari. 2004. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. *In the Proceedings of the 26th International Conference on Software Engineering.* Pages 127 -136.

M. Matinlassi, E. Niemelä, and L. Dobrica. 2002. Quality-driven architecture design and quality analysis method, A revolutionary initiation approach to a product line architecture. *VTT Technical Research Centre of Finland, Espoo.*

Medvidovic, Nenad; Oreizy, Peyman; and Taylor, Richard N. 1997. Reuse of off-the-shelf components in C2-style architectures. *In the Proceedings of the 1997 symposium of Software reusability.* Pgs. 190-198.

Medvidovic, Nenad; Rosenblum, David S.; Redmiles, David F. Robbins, Jason E. 2002. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions of Software and Engineering Methodology (TOSEM).* Volume 11. Issue 1. Pgs. 2 – 57.

Medvidovic, Nenad. 2002. On the Role of middleware in architecture-based software development. *In the Proceedings of the 14th international conference on Software engineering and knowledge engineering.* Pgs. 299-306.

Murray, Alexander T. and Shahabuddin, Mohammad. 2006. OO Techniques applied to a real-time, embedded, spaceborne application. *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented programming systems, languages, and applications.* Pages 830-838.

O'Farrell, Ryan and Hamilton, John A., Jr. 2009. Using Software Architecture to Facilitate Reuse in a Product Family. In *Proceedings of the 42nd Annual Simulation Symposium*. Annual Simulation Symposium.

Obbink, H., Müller, J.K., America, P., van Ommering, R., Muller, G., van der Sterren, W., Wijnstra, J.G., 2000. COPA: a component-oriented platform architecting method for families of software-intensive electronic products (Tutorial). *In: Proceedings of SPLC1, the First Software Product Line Conference*.

Pasetti, A. and Pree, W. 1999. A component framework for satellite on board software. *In the Proceedings of the 18th Digital Avionics Systems Conference.* Volume 2. Pages 7C.1-1 – 7C.1-10.

Radhakrishnan, R. and Sririman, B. 2007. Aligning Architectural Approaches towards an SOA-Based Enterprise Architecture. *In the Proceedings of the Working Conference on Software Architecture, 2007.* Pg. 38.

Riva, C.; Selonen, P.; Systa, T.; Tuovinen, A.-P.; Jianli Xu; Yaojin Yang. 2004. Establishing a software architecting environment. *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on ,* vol., no., pp. 188-197.

van Ommering, R. 2002. Building Product Populations with Software Components. *In the Proc. of the ICSE'02.* pp. 255 - 265.

Weiss, D.; Lai, C.; and Tau.; R. 1999. Software product-line engineering: a family-based software development process. Addison-Wesley, Reading, MA.

Appendix A – Original Software Architecture

AUBIESAT C&DH ONBOARD SOFTWARE ARCHITECTURE
Version 1.1
20 Nov 2008
Richard Chapman

I. Overview

OPERATING SYTEM
Micrium MicroC/OS-II v2.56, Ports for Atmega 128L by and Atmega 2561 by ???

TASKS DEFINED
- Control task
- Housekeeping task
- AX.25 send/receive task
- Secondary Command Receiver (SCR) processor
- CW data/beacon transmit task
- Data logging task

API's DEFINED
- Initialization
- Filesystem
- Analog to Digital Converter (ADC) reading
- Real time Clock read/write (RTC)
- Watchdog timer
- Electric Power System Control/monitoring (EPS)
- AX.25 stack
- Primary Comm radio control

HARDWARE INTERRUPTS PROCESSED
- Primary command received (UART Serial data)
- Watch dog timer (?)
- Reset
- SCR command received

SEMAPHORES DEFINED
- I^2C bus access (RTC, ADC, GPIO)

- SPI bus access (filesystem)
- Primary comm xmitter access
- Shared memory regions – currently none

II. Tasks

Each of the following is a task in MicroC/OS-II. These are cooperatively scheduled independent threads of control. Static variables can be shared, and semaphores and mailboxes are provided for mutual exclusion and communication. The general idea is that the **control task** schedules and coordinates all other tasks – all intertask communication should probably be via the control task. The **housekeeping task** should be alive at all times. The **AX.25** and **SCR tasks** wait for commands from their respective receivers. The **AX.25 task** can also respond to commands. The beacon task sends CW messages, either a predefined ID sequence, or a set of data values built on the fly. The **data logging task** reads the ADC values every few seconds (adjustable rate) and saves them to the filesystem.

Control Task

Description: Handles inter-task communication and coordination, schedules other tasks, processes primary or secondary commands, including forming the payload for AX.25 packets to be sent, and deconstructing the payload of AX.25 packets received.

API's accessed:

- filesys,
- AX.25,
- RTC,
- radio

Sends messages to tasks:

- beacon (change rate, beacon on/off, send CW data)
- data logging (change rate)
- AX.25 (send packet, either connected or UI)

Receives messages from tasks or handlers:

- AX.25 (msg received)
- SCR (send UI, CW beacon on/off, CW data beacon, change beacon rate, xmit enable/inhibit, reset)

Controls hardware

- Transmitter power level
- Transmitter source selector
- CPU reset
- EPS restart pin
- Transmitter input

Interfaces to

- Primary xmitter
- Flash/SPI
- I^2C

Data Logging Task

Description: Every few seconds (adjustable rate), read data from all channels of Analog to Digital Conversion (4 ADC chips on I^2C bus, 8 channels each, and 8 channels of internal ADC on the CPU, and store these to a file on the flash.

API's accessed:

- Filesystem
- ADC reading
- RTC

Receives messages from Tasks or handlers

- Control (adjust rate of logging)

Sends messages to tasks

- Control (error)

Interfaces to

- ADC's (I^2C)
- Flash/SPI

AX.25 Task

Description: Receives packets from ground station (when the serial data incoming interrupt is triggered, and forwards the payload to the control task), sends data via either connected or unconnected AX.25 packet when directed by control task. Relies on control task to build/deconstruct payload**.** The AX.25 API provides the Control Task with access to the AX.25 stack. The AX.25 Task facilitates, maintains, or is the AX.25 stack, but doesn't provide external access to it - it is awoken by the Control Task making API calls: ax25_read, ax25_write, etc The AX.25 Task does, though, call kiss_read/kiss_write which calls serial_readb/serial_write(b), respectively.

API's accessed:

- AX.25

Receives messages from tasks or handlers

- Control (send UI, send connected, connect)
- Primary command interrupt handler

Send messages to tasks

- Control (connection state, received msg)

Interfaces to

- UART<->PIC<->Pri comm.

Beacon Task

Description: To periodically (variable rate, default 1/minute) send a CW beacon identifying the satellite. Also to send, on SCR or primary command single CW transmissions consisting of critical variables

API's accessed:

- none

Receives messages from tasks

- control (beacon on/off, adjust beacon rate, CW data transmission)

Send messages to tasks

- none

Interfaces to

- Primary transmitter (via direct pins from CPU)

SCR Task

Description: Detect valid SCR command via external interrupt pin, pass message on to the control task. NOTE: this functionality may be migrated from an independent task into the interrupt handler that processes a pin from the SCR

API's accessed:

- none

Receives messages from tasks or handlers

- SCR valid signal received pin interrupt handler

Send messages to tasks

- control (all valid SCR commands)

Interfaces to

- 4 data lines from SCR (direct pins to CPU), SCR valid signal received pin

Housekeeping Task

Description: This is the first task spawned by control, and it handles the initialization/ADM deployment and confirmation. It pings the watchdog timer periodically, monitors the EPS, and keeps track of stuff that needs to be done any time the system resets.

API's accessed:

- filesystem
- RTC
- EPS
- Initialization
- watchdog

Receives messages from tasks

- none

Send messages to tasks

- control ? (notification of dangerous condition)

Interfaces to

- ADM (GPIO via I^2C)
- RTC
- flash/SPI
- watchdog (direct pins to CPU)
- EPS (direct pins to CPU)

III. Application Programmer Interfaces (API's)

Initialization

- 15 minute wait on first launch
- check for successful antenna deployment by reading flash status page
- attempt 5 (?) times to deploy antenna if not deployed
- write success/no success of antenna deployment to flash status page
- update flash status page by incrementing variable that stores number of power-ups encountered

Used by: housekeeping task
Hardware accessed: ADM (I2C), flash (SPI)

AX.25

- process received message and send payload to control when interrupt handler runs
- transmit message on command from control

Used by: AX.25 task, control task
Hardware accessed: Serial ports

Filesystem

- read a file of sensor values (find most recent or find by name)
- write a file of sensor values
- read/write the status page
- keep track of bad pages

Used by: data logging, control, housekeeping
Hardware accessed: flash/SPI

Pri Comm Radio Control

- change transmitter power level (1-4)
- do a science experiment transmission
- control transmitter data source (atmega or PIC)

Used by: control
Hardware accessed: Pri comm. Transmitter (via direct pins to CPU)

RTC

- Set the real time clock
- Read the value of the real time clock

Used by: logging, control, housekeeping
Hardware accessed: RTC (I2C)

ADC

- Read values from onboard or offboard ADC's return as 16-bit value.

Used by: data logging
Hardware accessed: internal ADC (ATmega CPU), external ADC (I2C)

IV. Primary Comms Payload Data Formats

Uplink (Ground station to Satellite)

Length is variable. The payload consists of a 1-byte command, followed by a number of bytes that is dependent on the command value.

Command types (byte 1 of payload)

Value	Meaning	Arguments
0x01	stop transmit	none
0x02	allow transmit	none
0x03	power system restart	none
0x04	c&dh restart	none
0x05	I2c restart	none
0x06	request connected data	none
0x07	request unconnected data	none
0x08	CW beacon	message number, on/off, interval new message
0x09	science experiment xmit	none
0x0a	write internal variable	variable index, length, new value
0x0b	upload new code	page address, 256-bytes of data
0x0c	transmitter power	power level, 1 to 4

Argument formats (bytes 2 to end of payload)

BEACON

Byte 2: index number of pre-stored message, or zero for new custom message
Bytes 3-4: 2-byte integer number of seconds between CW beacon
Byte 5: 0 = regular CW id beacon enable, 1 = regular CW beacon disable, 2= transmit CW data message, Bytes 6-n = new message, only used if Byte 2 is zero

WRITE INTERNAL VARIABLE

Byte 2: unsigned int index of variable to be rewritten, from look-up table in control task
Bytes 3-4: unsigned integer length of the variable's value
Bytes 5-n: value of the variable to be changed

UPLOAD NEW CODE

Bytes 2-3: start address of page to be rewritten in program memory
Bytes 5-261: bytes to be written into that page

TRANSMITTER POWER

Byte 2: new power level (unsigned integer 1-4)

Downlink (Satellite to Ground Station)

The satellite sends down AX.25 from the filesystem in response to the "request connected data" or "request unconnected data" commands, or to acknowledge the receipt of any other Primary commands.

Response Type (byte 1 of payload)

Value	Meaning	Arguments
0x80	connected data	RTC, status, sensor data
0x81	unconnected data	same as connected data
0x82	acknowledge	type of command being ack'ed

Argument formats(bytes 2-n of payload)

SENSOR DATA

The ADC sensors are 12-bit sensors, but we put the data into 16-bits, with the 4 most significant bits being zero. We do not scale the data on the satellite, but transmit the raw values down.

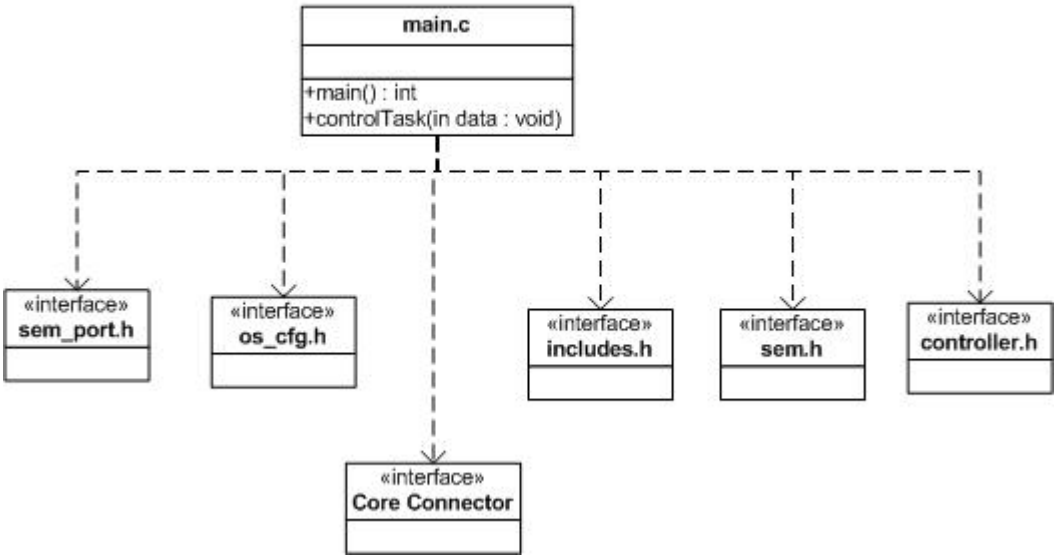
Bytes 1-2: value from ADC1, channel 1
Bytes 3-4: value from ADC1, channel 2
…
Bytes 15-16: value from ADC1, channel 8
Bytes 17-18: value from ADC2, channel 1
…
Bytes 33-34: value from ADC3, channel 1
…
Bytes 49-50: value from ADC4, channel 1
…
Bytes 63-64: value from ADC4, channel 8
Bytes 65-66: value from internal ADC on Atmega, channel 1
…
Bytes 79-80: value from internal ADC on Atmega,. Channel 8
Bytes 81-84: RTC value
Bytes 85-86: ????

COMMAND ACKNOWLEDGE

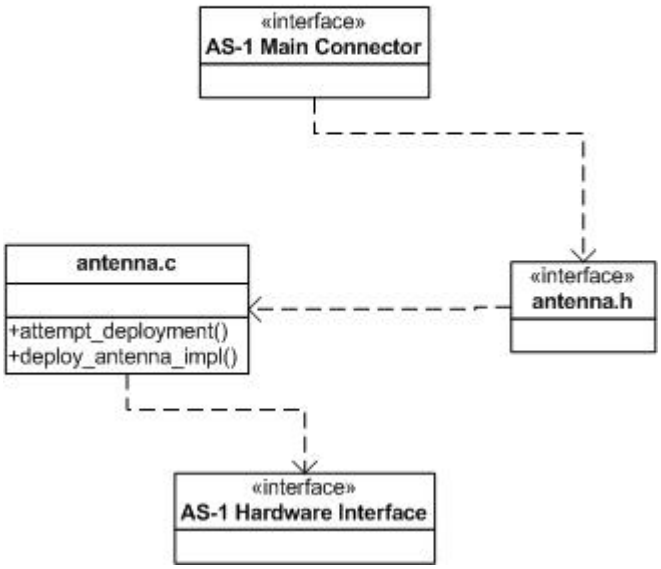
Byte 2: the command type byte from the command being acknowledged (see uplink data format)

Appendix B – AubieSat-1 Detailed Software Documentation

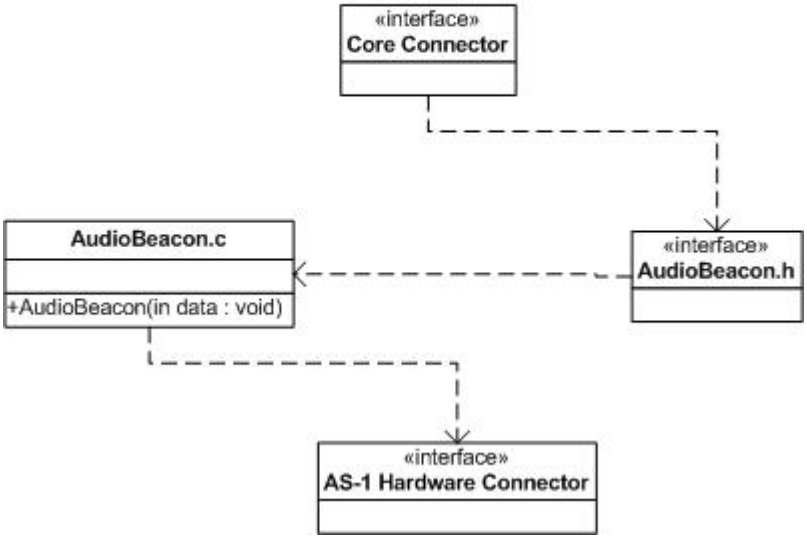
AubieSat-1 Main Component



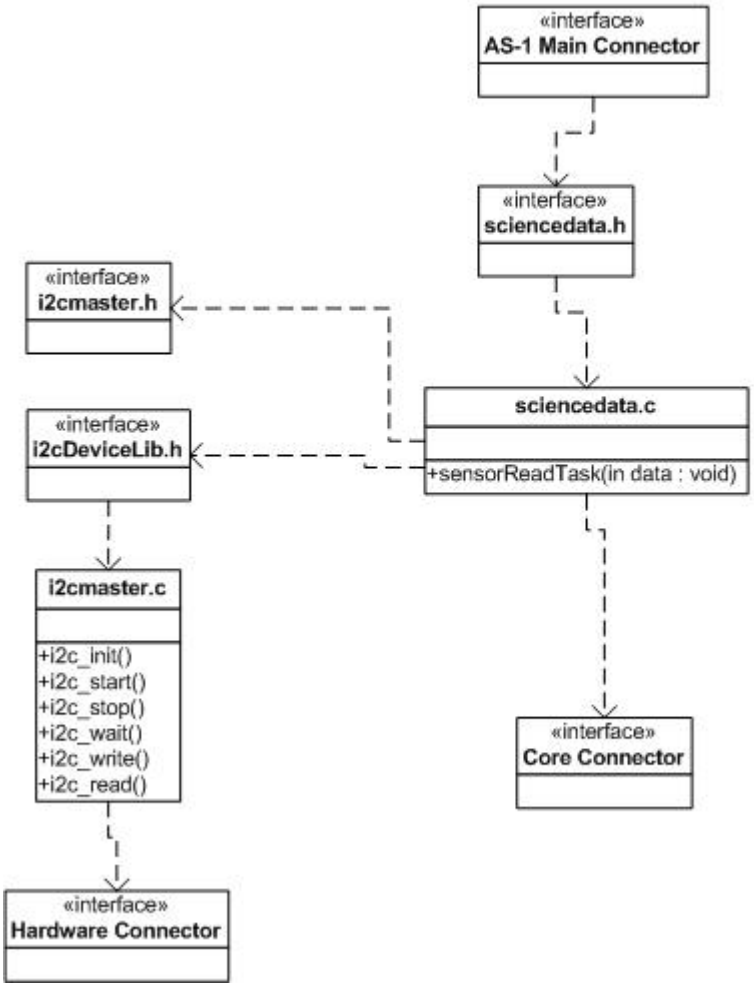
Antenna Deployment Component



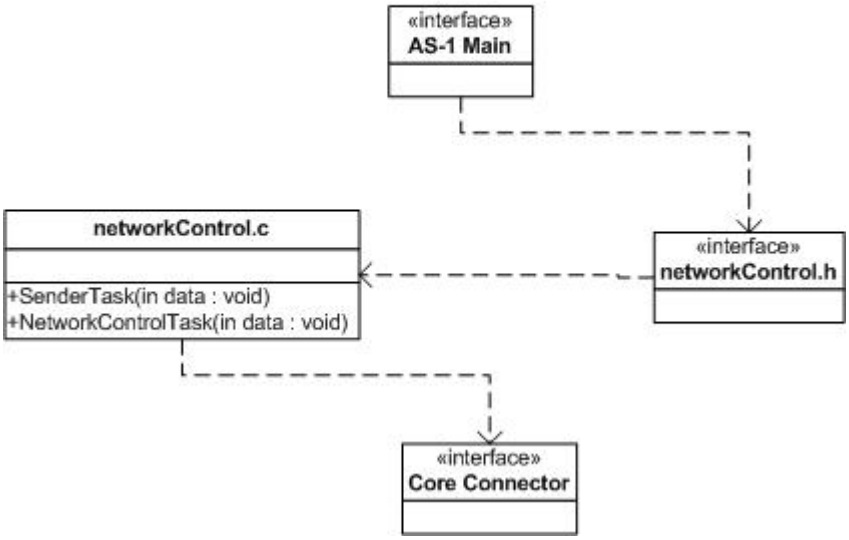
Beacon Transmission Component



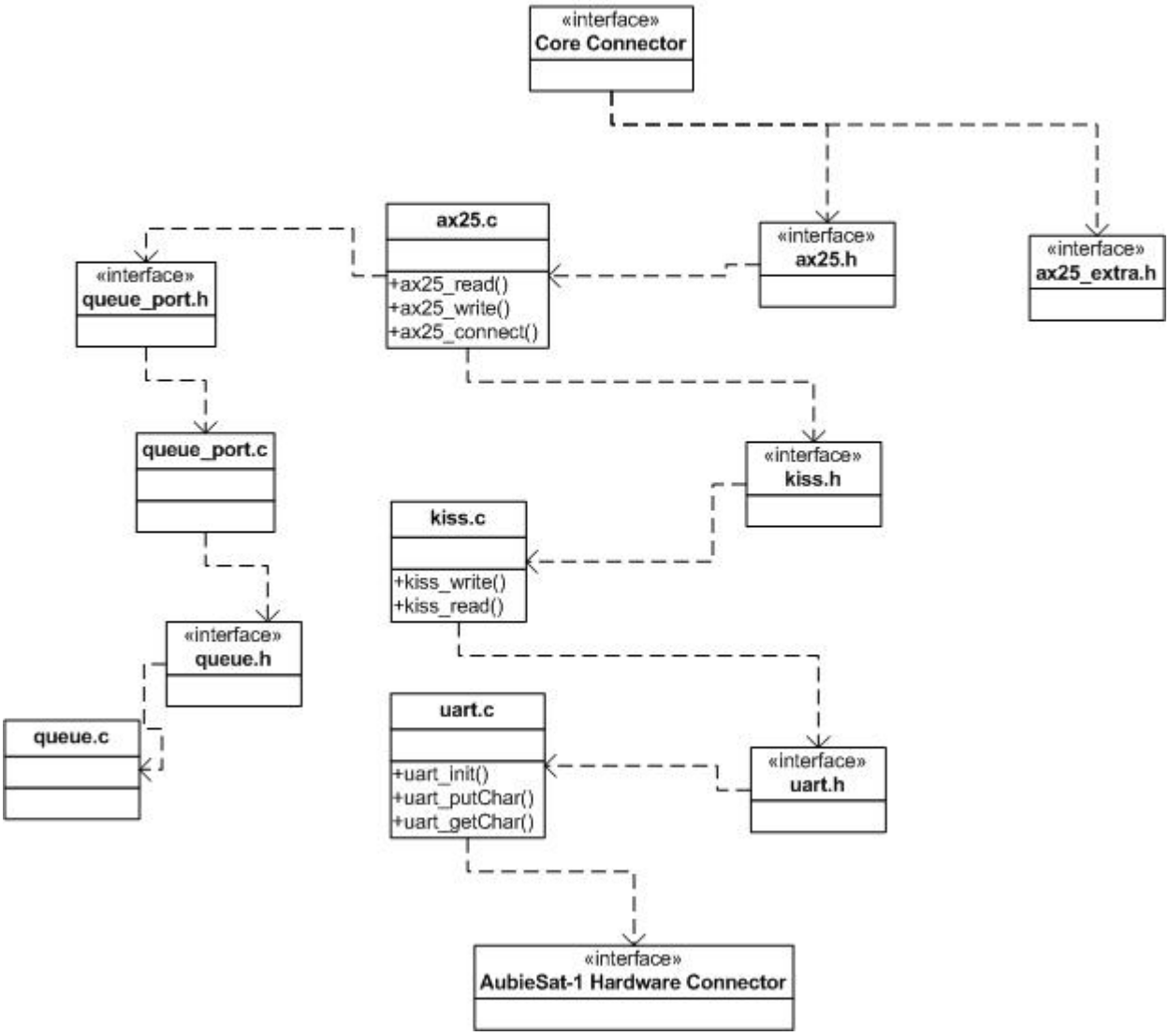
Scientific Data Logging Component



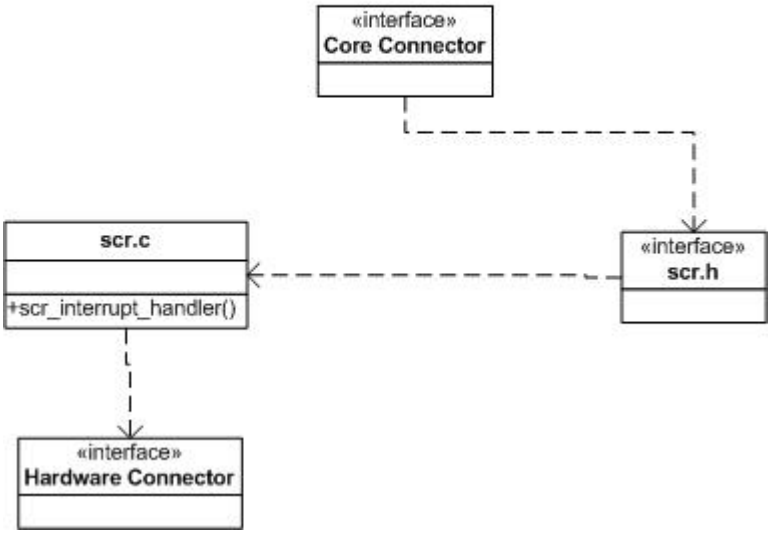
Networking Protocol Component



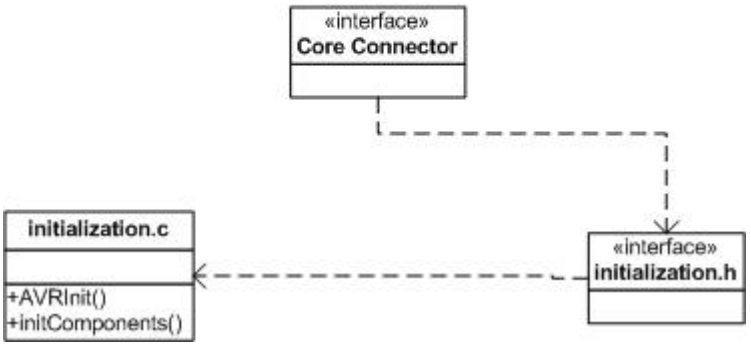
Networking Component



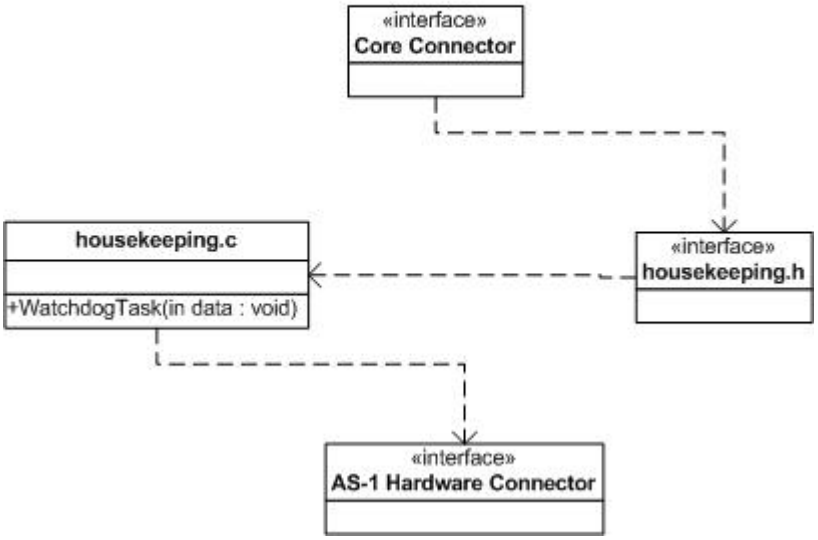
Secondary Command Receiver Component



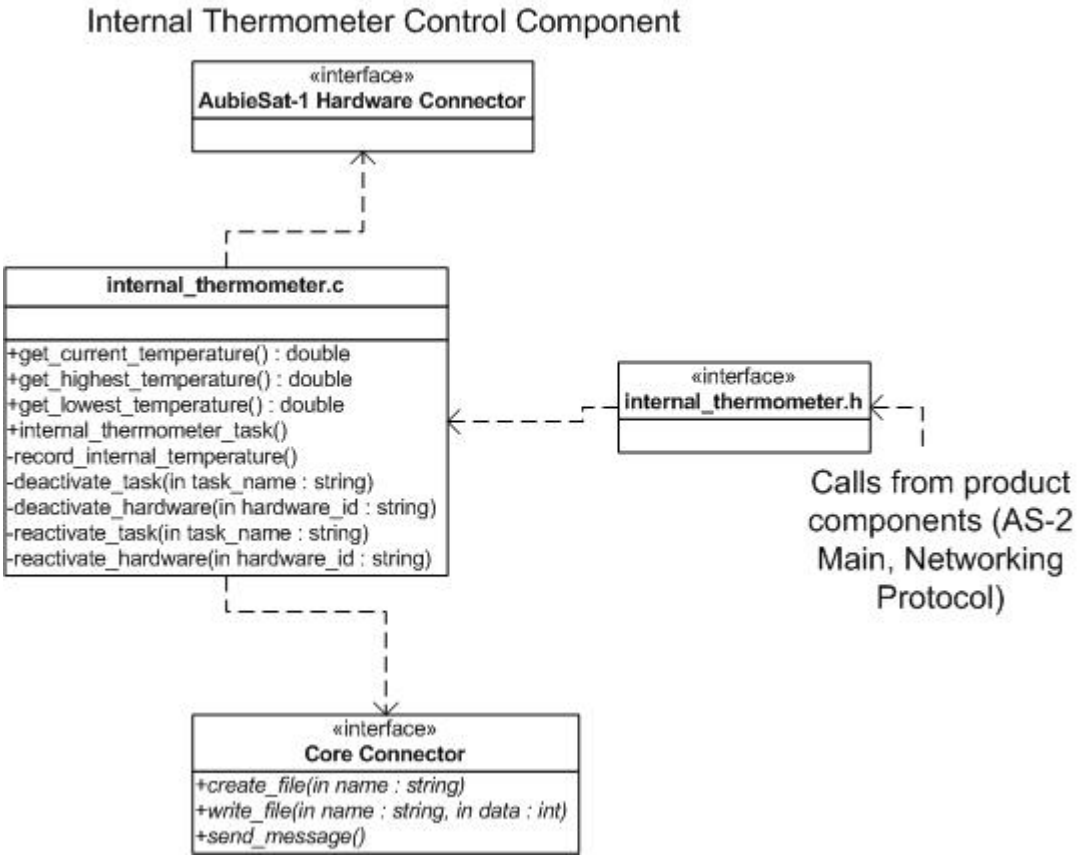
Initialization and Startup Component



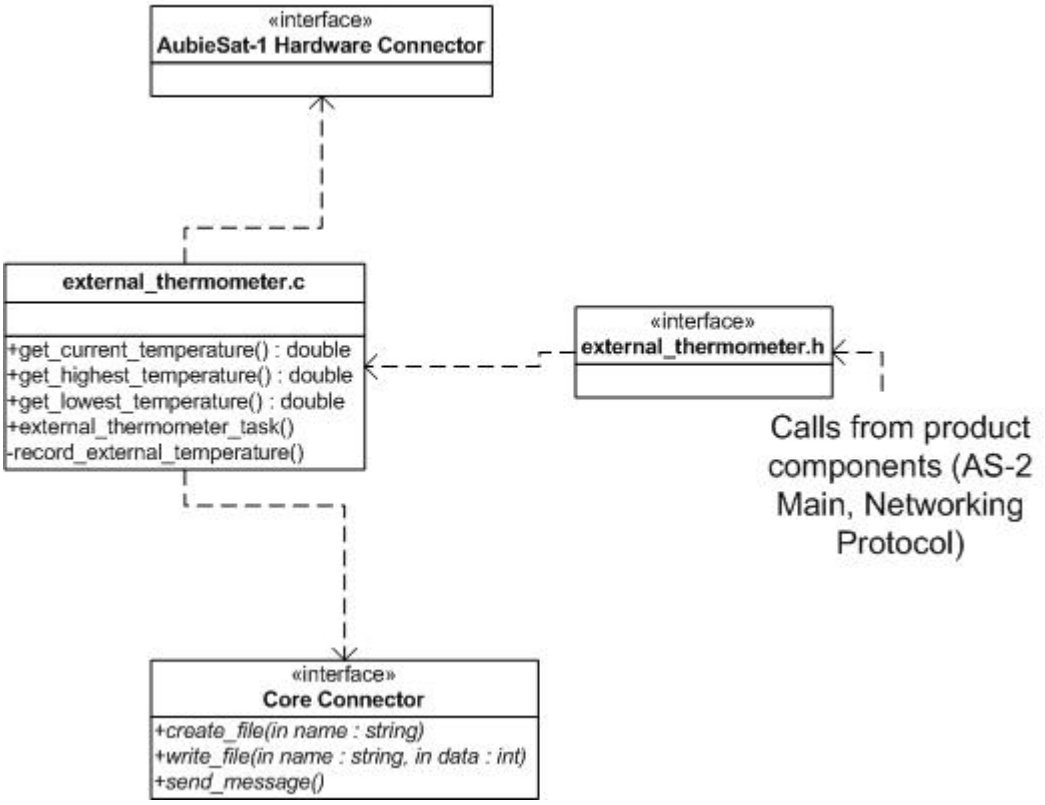
Watchdog Timer Component



APPENDIX C- AubieSat-2 Detailed Software Documentation



External Thermometer Control Component



Temperature Sensing Component