Incorporating Modern Development and Evaluation Techniques into the Creation of Large-Scale, Spacecraft Control Software

by

Kathryn Anne Weiss
B.S. Computer Engineering, Mathematics, Honors Certificate, Marquette University, 2001
S.M. Aeronautics and Astronautics, Massachusetts Institute of Technology, 2003

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in the field of Aerospace Software Engineering at the
Massachusetts Institute of Technology

February 2006

Signature of Author: ______________________
Kathryn Anne Weiss

Certified By: ______________________
Nancy G. Leveson
Professor of Aeronautics and Astronautics
Thesis Committee Chair

Certified By: ______________________
David Miller
Associate Professor of Aeronautics and Astronautics

Certified By: ______________________
John Keesee, Col. USAF
Senior Lecturer of Aeronautics and Astronautics

Certified By: ______________________
Jeffrey Hoffman
Professor of the Practice of Aeronautics and Astronautics

Accepted By: ______________________
Jaime Peraire
Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

[This page intentionally left blank.]

Incorporating Modern Development and Evaluation Techniques into the Creation of Large-Scale, Spacecraft Control Software

by Kathryn Anne Weiss

Submitted to the Department of Aeronautics and Astronautics in February 2006
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the field of Aerospace Software Engineering

Abstract

One of the major challenges facing the development of today's safety- and mission-critical space systems involves the construction of software to support the goals and objectives of these missions, especially those associated with NASA's Space Exploration Initiative, which has now become the focus of the US Space Program and its contractors. Consequently, the software used to implement much of the functionality in the various flight vehicles and ground facilities must be given special consideration. This dissertation outlines a new approach to spacecraft software development that focuses on incorporating modern software engineering techniques into the spacecraft domain including (1) a product-line approach to the software development enterprise and (2) a software architecture-centric design process to support that approach.

The new product-line approach is demonstrated through its application to the Exploration Initiative. The technical and managerial aspects of the product line, which are required to successfully field the line, are described in detail. Among the technical artifacts developed to support the line, the software architecture is the most important. Consequently, it was necessary to create a systems engineering-based development, evaluation, and selection process for the construction of the software product-line architecture. This development approach is known as Multi-Attribute Software Architecture Trade Analysis (MASATA) and is demonstrated on the vehicles and facilities of the Exploration Initiative, the Crew Exploration Vehicle in particular.

Based on the functional requirements of the Exploration Initiative and the quality attributes desired by the stakeholders, a software product line architecture is presented. The desired quality attributes include analyzability with respect to safety, ease of verification and validation, sustainability, affordability, buildability, ability to meet real-time requirements and constraints, and "monitor"-ability. Several architectural style options were selected for evaluation with respect to the requirements and attributes through MASATA including traditional subsystem-based decomposition, state analysis, functional decomposition and implicit invocation. The conceptual software product-line architecture selected to support the Exploration Initiative is based upon these results.

Thesis Committee Chair:	Nancy G. Leveson
Title:	Professor of Aeronautics and Astronautics

[This page intentionally left blank.]

Acknowledgements

This research was partially supported by a NASA contract to Draper Labs and MIT for Concept Evaluation and Refinement of the Space Exploration Initiative and the NASA Engineering for Complex Systems program (NAG2-1843). Special thanks go to Professor Nancy Leveson, Professor David Miller, Colonel John Keesee, Professor Jeff Hoffman, and Jim Francis of Payload Systems Inc. for their vital contributions to this work.

[This page intentionally left blank.]

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

AI	Artificial Intelligence
ARID	Active Reviews for Intermediate Designs
AS	Ascent Stage
ATAM	Architecture Trade-off Analysis Method
ATV	All-Terrain Vehicle
CBAM	Cost Benefit Analysis Method
CBM	Coupling-Between-Modules
CBMN	Coupling-Between-Module-Nodes
CBN	Coupling-Between-Nodes
CEV	Crew Exploration Vehicle
CIM	Coupling-Inside-Modules
CMU	Carnegie Mellon University
COTS	Commercial Off-The-Shelf
CPN	Colored Petri Nets
DSc	Descent Stage
DSN	Deep Space Network
EDL	Entry, Descent, Landing
ECLSS	Environmental Control and Life Support System
FDIR	Fault, Detection, Isolation and Recovery
FTA	Fault Tree Analysis
GN&C	Guidance, Navigation and Control
HAB	Habitat
HCI	Human-Computer Interaction
HLE	Human-Lunar Exploration
HME	Human-Martian Exploration
ISRU	In-Situ Resource Utilization
ISS	International Space Station
JPL	Jet Propulsion Laboratory
LEO	Low-Earth Orbit
LMO	Low-Mars Orbit
MCO	Mars Climate Orbiter
MDS	Mission Data System
MER	Mars Exploration Rover
MGS	Mars Global Surveyor
MIT	Massachusetts Institute of Technology
NASA	National Aeronautics and Space Administration
OPN	Object-Process Network
QDS	Quantified Design Space
QFD	Quality Function Deployment
RAM	Random Access Memory
SAAM	Software Architecture Analysis Method
SAE	Software Architecture Evaluation
SEI	Software Engineering Institute
SE&I	Systems Engineering and Integration
SpecTRM	Specification Toolkit and Requirements Methodology
STAMP	Systems Theoretic Accident Model and Process
STPA	STamP Analysis
TDRSS	Tracking and Data Relay Satellites
TEI	Trans-Earth Injection
TMI	Trans-Mars Injection
V&V	Verification and Validation

[This page intentionally left blank.]

Chapter 1:

Introduction and Problem Statement

One of the most important aspects of the software engineering process is the domain within which and for which the software is being developed. The software engineers must understand the characteristics of the domain because it is these characteristics that help to shape both the development techniques used and the artifacts produced as well as the strategies employed for verification, validation and maintenance.

Creating domain-specific approaches to software engineering in a variety of domains has proven to be quite successful because the strategies employed focus on the particular needs of those domains and are as close as possible to the user's conceptual view of that application domain. For example, domain-specific languages such as SQL (Standard Querying Language) and Meta-H have been used by database developers and in Honeywell's Guidance, Navigation and Control software development process respectively for years [59]. These domain-specific languages aid in minimizing the semantic distance between the programmer, who is usually a domain expert, and the problem.

One area in which a domain-specific solution to software engineering is especially applicable is spacecraft software engineering as a whole. Spacecraft engineering is extremely domain specific on a subsystem by subsystem basis; traditionally, the various subsystems that comprise a spacecraft are engineered by experts from that particular domain. The subsystems are then integrated together by system engineers to form the spacecraft software as a whole. Although this approach to spacecraft software engineering has provided the basis for many successful NASA spacecraft, satellites and rovers in the past, these techniques are becoming obsolete because they (1) do not take a systems view of the spacecraft, (2) do not take into consideration the complex physical environment in which spacecraft operate and (3) cannot rely on the redundancy techniques that provide risk mitigation for hardware components.

Writing software in the spacecraft-engineering domain is quite different from writing software for traditional enterprise applications and other information management systems because spacecraft software resides in a resource-limited environment. As described in [39], resource-limited robots such as unmanned spacecraft and Mars rovers have only enough resources (plus a small margin) to carry out their mission objectives due to the cost of launching mass into space. Consequently, mission activities must be designed with these limitations in mind: turning on a camera to take pictures uses power and storing those pictures consumes non-volatile memory. The physical side effects in this case are non-negligible and therefore must be managed; for example, monitoring the effects of a heater on power consumption, the effect of temperature on a sensor measurement, and the health of a science instrument all become paramount.

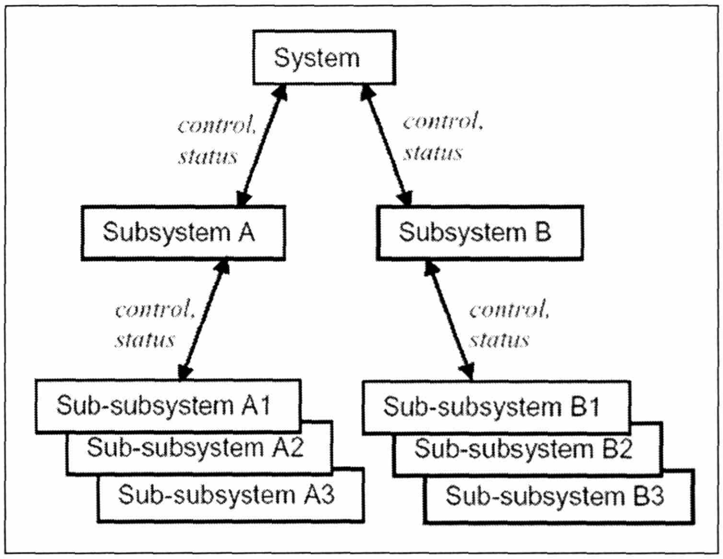


Figure 1-1. Typical Spacecraft Software Decomposition [39]

Traditional spacecraft software architectures follow a similar pattern: the spacecraft is decomposed into a series of subsystems and from that subsystem into a series of components, until the actual hardware is reached. Figure 1-1 depicts a typical spacecraft software decomposition. As stated in [39],

> "The main problem in applying a device/subsystem architecture to resource-limited systems is that the architecture provides no leverage in dealing with the many non-negligible inter-subsystem couplings. Each such coupling has to be handled as a special case, leading to a tangle of subsystem-to-subsystem interactions hidden behind the façade

> of modular decomposition. In effect, the original architecture becomes an appealing fiction."

Clearly, a new architecture is needed that takes into consideration both a systems view of the spacecraft as well as the complex physical interactions between the traditional subsystems.

Furthermore, techniques used for reliability purposes are not applicable in the software domain. Almost all mechanical components on board a spacecraft are multiply redundant and have well-known failure rates. The organizations that develop these spacecraft use hazard analysis and risk mitigation techniques that rely on this redundancy. Although these techniques work well for the hardware, they cannot be used for the software that controls not only the components' operations but also the system itself, for the reasons enumerated in the previous paragraph. In addition, redundancy is not effective against design errors, which is the only type of error software can have. Spacecraft software engineers have relied on design and implementation conservatism (in the form of command sequences and large software ground support described below) to ensure the success of projects ranging from the early Viking and Voyager missions to today's Cassini and Mars Exploration Rovers (MER).

Currently, unmanned spacecraft follow a series of time-stamped commands, frequently uploaded in real-time from ground controllers. In order to assure that the mission objectives are achieved, the spacecraft's abilities are limited to this predefined operational model. The occurrence of unpredictable events outside nominal variations is dealt with by high-level fault protection software (often added at the end of development as a separate module), which may be inadequate if time or resources are constrained and recovery actions interfere with satisfying mission objectives [105]. In these situations, the spacecraft enters a safe mode in which all systems are shut down except for those needed to communicate with Earth. The spacecraft then waits for instructions from the ground controller [54].

There are two major problems with this approach to fault detection, mitigation, and recovery. First, if the spacecraft is far from earth, there is a large communication delay. During the cruise phase of flight, the delay may not cause any problems; the spacecraft has time to await new instructions from Earth, while it is in safe mode. However, if the spacecraft is executing an event sequence during a safety-critical flight phase, such as an orbit insertion, the procedure may prove to be extremely costly and possibly fatal [54]. Second, large and expensive deep-space missions such as Cassini require an enormous number of ground controllers to support each phase of flight. One of the reasons that satellite developers are pushing for an increased software ability aboard modern spacecraft is the high cost of the human controllers and ground operations that currently support spacecraft missions. These ground operations include everything from simple monitoring tasks to complex planning algorithm construction, all done while the spacecraft is in flight. Automating many of the simple command sequences that are traditionally written and uploaded by the ground controllers in real-time may significantly lower the cost of maintaining ground control. The savings become even more pronounced when compounded over many spacecraft missions.

Command sequences in combination with ground support have been used successfully since the first space exploration missions and continue to be used today. This paradigm has remained the foremost approach to spacecraft software architecture due to a reticence on the part of many stakeholders to risk a multi-million dollar project on alternative architectures that have not been previously tried and evaluated in space. Within the last five to ten years, however, new scientific goals have led to increasingly demanding mission requirements, such as real-time decision-making and fault recovery during time-critical operations such as rendezvous and orbital insertion and planetary entry, descent and landing. Consequently, the sequencing and ground support approach to spacecraft software architecture is no longer sufficient.

While modern spacecraft have to satisfy increasingly complex mission requirements, they are also required to maintain the same high level of safety as their predecessors. In the past, spacecraft engineers have relied on proven hardware practices and conservative software technologies to ensure mission success. In particular, the software architecture used in spacecraft has remained unchanged for nearly 50 years. To satisfy these increasing requirements, spacecraft engineers need new software architectures that do not compromise safety. As stated in [48], “High-level, closed-loop control based on sophisticated model-based, fault-tolerant, configurable, and dynamic software architectures will let NASA pursue space missions that for technological or financial reasons it could not otherwise attempt.”

Given the difficulties with developing and writing spacecraft control software, i.e. the complex, resource-limited environment and the need to perform real-time risk mitigation, it becomes essential that the software is designed from the beginning with these characteristics in mind. Developing and selecting an appropriate software design is referred to as software architecting and evaluation. Performing software architecture evaluation prevents architectural mismatch, a term originally coined by Garlan in [53] to indicate the interface problems associated with COTS reuse, but used in this context to mean the inability of an architecture to achieve the functional requirements and non-functional quality attributes desired by the stakeholders of the system. There are many instances of architecture mismatch throughout several engineering domains, and two of these examples come from the aerospace field: the early stages of the Iridium and Mars Pathfinder projects.

As early as 1976, computer scientists recognized the need to select a software architecture in the early stages of design [133]. It was argued then that this architecture selection should result from a careful investigation of all possible approaches and should not be influenced by hardware or implementation considerations. The intervening decades have seen a lot of descriptions of software architectures, but little scientific evaluation of these architectures for their suitability with respect to various project characteristics [50], [120]. There is still a recognized need to evaluate software architectures, but not much research on how this should be done. Garlan observed in a paper on research directions in software architecture that "architectural choices are based more on default than on solid engineering principles" [49]. Currently, that default is usually an object-oriented architecture. The lack of careful decision-making about architectures has not always led to success.

As an example, in early 1993, Motorola began development of the Iridium telecommunications project. Although Iridium was a commercial failure, it was technologically very successful telecommunications projects. Iridium provides users with limitless worldwide communications coverage through its network of 72 satellites. With over 17 million lines of code, the development team realized the need to choose an architecture early in the process. Because of the extensive use of COTS, the projected need for expandability, and a language survey, the team chose an object-oriented approach to development in C++. Soon after these decisions were made, the software team had to "throw [their] hands up and say no more OO" [73]. Asked if there was anything he would have done differently with respect to the software development approach, Ray Leopold (then Vice President of the Iridium System) stated "We would have never started with an object-oriented design, because we lost so much time" [73]. Perhaps the reason for the switch from an OO architecture to a more traditional approach was due to a mismatch between the system requirements and the ability of an object-oriented architecture to satisfy those requirements. In fact, some software engineering research has questioned the applicability of OO designs [57], [58], [66], [93]. After the selection of a more appropriate software architecture, the Iridium project was able to make up the time lost and in fact it was delivered 52 days ahead of schedule and has a projected lifetime through 2014.

In a similar case, Stolper describes his experiences in software architecture selection for the Mars Pathfinder mission [124] where early in the project's life cycle they decided to use object-oriented design. "We expected an OO approach to accelerate development and reduce costs." Stolper and his team quickly realized that the textbook descriptions of an OO architecture did not scale to the rapidly expanding number of objects in the embedded spacecraft flight software. Furthermore, the "impressive results" claimed in the OO literature were not achieved for their system, due partly to a difficulty in evaluating the utility of new objects when they were proposed and the complex interactions between many of the objects. The original architecture was abandoned, and the developers created a "meet-in-the-middle" architecture that combined top-down functional decomposition with bottom-up OO design. Like the Iridium project, Pathfinder was highly successful and was developed in only three years and for $150 million (an unusual feat for spacecraft).

This dissertation explores the choice of software architectures for spacecraft. In particular, the objectives of this dissertation are four-fold:

(1) To discuss the difficulties with developing quality spacecraft control software,

(2) To describe the current research in both software and spacecraft engineering relevant to the problem domain, solutions that have been proposed for solving these problems, and the strengths and weaknesses of these approaches,

(3) To propose a new paradigm for creating large-scale, safety-critical software for spacecraft, and

(4) To develop a conceptual, or high-level, software architecture for a case study (NASA's Exploration Initiative) that helps to illustrate how the new paradigm can be applied to meet the needs of specific objectives within the field.

Chapter 2 contains a description of the current research relevant to the problem domain. Chapter 3 then synthesizes the information presented in Chapter 2 and presents a new approach based on the related research and background information. A new technique for creating spacecraft software is described and examples of its application to the Exploration Initiative are discussed. Chapter 4 elaborates on software architecture development and selection, which is central to the approach, and describes the steps in generating an appropriate software architecture given the characteristics of the problem domain. Chapter 5 then describes the application of these steps to the Exploration Initiative and the results of the software architecture development and selection. Finally, Chapter 6 presents a set of conclusions, recommendations, contributions, and future work in the area of spacecraft software engineering and architecting.

Chapter 2:

Background Information and Related Research

In the previous chapter, the changing nature of spacecraft mission objectives was discussed as well as their impact on how software is engineered to support these changing objectives. The out-dated techniques currently employed by spacecraft software engineers need to be replaced by newer, more state-of-the-art software engineering practices. In other words, spacecraft software engineering needs to consider and incorporate recent lessons learned in the software engineering community and with current software engineering techniques. This chapter describes (1) applicable techniques from the software engineering research community and (2) alternative approaches and architectures developed by spacecraft engineers to replace the current spacecraft software engineering paradigm.

2.1 – Applicable Software Engineering Techniques

Although there have been significant strides in creating a science of design for software engineering similar to that of other engineering disciplines, many of the research avenues that comprise these new approaches have not been carried over into the aerospace industry. Consequently, the state-of-the-art of spacecraft software engineering has lagged behind software engineering in general. This section describes some of the contemporary research areas in the software engineering community that are applicable to and can be integrated into modern spacecraft software engineering approaches. The techniques include, but are certainly not limited to, software product lines, model-based development, safety-driven design, process modeling and software architecture development and selection.

2.1.1 – Software Product Lines

A product line is a set of systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission. Product lines, or product families, are not new concepts to manufacturers or engineers; product lines abound in everyday lives and include everything from washers and driers to automobiles and airplanes. Some of the best examples of product families come from the aeronautics industry: the Boeing 757 and 767 were developed in tandem and nearly 60% of their parts overlap; the Airbus A-318, A-319, A-320 and A-321 comprise a family of aircraft that range from 100 to 220 seats but clearly share production commonalities [31].

In recent years, it has become apparent that product lines can exist in the field of software engineering as well. Studies of Japanese "software factories" have shown that the "secrets" of Japanese software companies' success can largely be attributed to a software product-line approach. In fact, a Swedish company, CelsiusTech Systems AB, who adopted the software product-line approach for creating shipboard command and control systems, claims they have reached reuse levels into the 90% range and, as an example of their productivity gains, can port one of their systems to a whole new computing environment and operating system in less than a month [31].

A software product-line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [31]. Each product is formed by taking applicable components from the base of common assets, tailoring them as necessary through preplanned variation mechanisms, adding any new components that may be necessary and assembling the collection according to the rules of a common, product-line-wide architecture. Software product lines allow an organization to take economic advantage of the fact that many of its products are very similar. Consequently, scoping the product line becomes an essential activity in not only defining which products will be included in the product line, but also building the core asset base, which forms the basis from which each product is assembled.

Fielding a software product line involves three essential activities: (1) core asset development, (2) product development using the core assets, and (3) technical and organizational management [Figure 2-1].

The goal of *core asset development* is to establish a production capability for products. There are three things required for the production capability necessary to develop products; these are the outputs of the core asset development and include:

1. Description of the product line scope
2. Core Assets, which include:
    - Requirements
    - Architecture
    - Software Components
    - Performance modeling and analysis
    - Business case, market analysis, marketing collateral, cost and schedule estimates
    - Tools and processes for software development and for making changes
    - Test cases, test plans, test data
    - People, skills, training
3. A production plan that describes how the products are produced from the core assets, including a detailed process model, which is described in greater detail in Section 2.1.4.

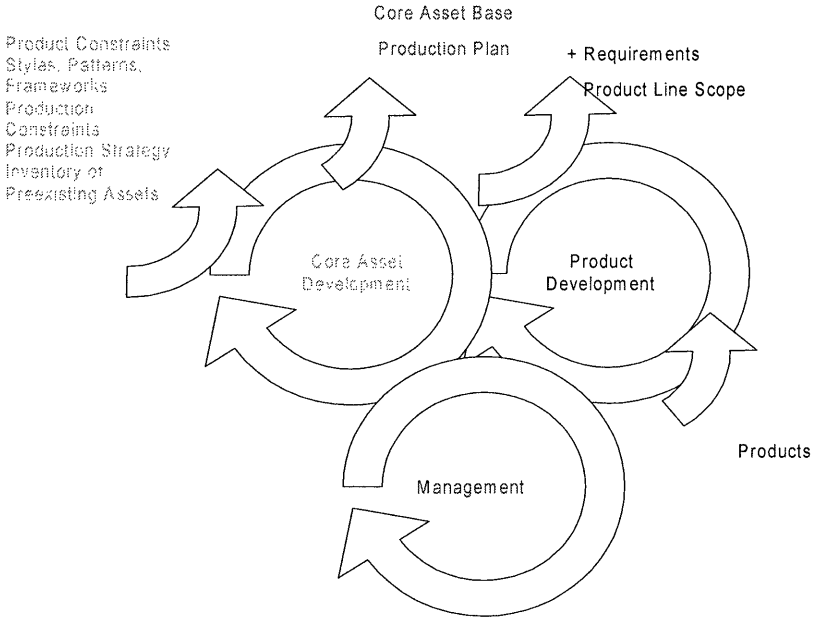


Figure 2-1. Three Essential Activities for Software Product Line Production [31]

Among the core assets, the architecture is key; the ability for a system to achieve its requirements is largely determined by the time the software architecture has been selected. Furthermore, for a software product line, in which the product-line architecture is reused from one product to the next, the product-line architecture determines whether or not each product in the line can achieve its requirements. The importance of architecture and architecture evaluation to the software product line approach is further discussed in Section 2.1.5.

The *product development activity* depends on the three outputs from the core asset development activity plus the requirements for individual products. This activity produces a new product that is based on the product line's core assets, but incorporates the requirements specific to the particular project.

The third activity, *technical and organizational management*, plays a critical role in successfully fielding a product line, because the activities of core asset development and product development must be given resources, coordinated and supervised.

Core asset development and individual product development are both supported by software engineering, technical management and organizational management. These three bodies of work must be mastered by an organization in order to successfully carry out the essential aspects of a product line. For example, the software engineering body of work includes activities necessary to make sure that appropriate technologies are applied to the creation and evolution of both core assets and products, such as architecture definition and evaluation, component development, COTS utilization, mining existing assets (which refers to the study and use of legacy assets already owned by the organization), requirements engineering, software system integration, testing, and understanding relevant domains [26].

Technical management includes those management activities that are necessary for engineering the creation and evolution of both the core assets and products. Technical management activities include: configuration management, data collection, metrics and tracking; decisions about whether to create, buy or mine (find assets among legacy components) and/or commission with an outside organization the analysis of assets; process definition; scoping; technical planning; technical risk management; and tool support [26]. Finally, organizational management refers to those activities necessary for orchestrating the entire product-line effort, including building a business case, customer interface management, developing an acquisition strategy, funding, launching and institutionalizing, market analysis, operations, organizational planning, organizational risk management, structuring the organization, technology forecasting and training [26].

The benefits and costs associated with each of the core assets as described in [31] are shown in Table 2-1. In general, the benefits from using a software product line approach are derived from the reuse of the core assets in a strategic and prescribed way. Once the product-line asset repository is established, there is a direct savings each time a product is built. Most of the costs associated with the product line are up-front costs that are connected to establishing the product line and its core assets.

Asset	Benefit	Additional Cost
Requirements: The requirements are written for the group of systems as a whole, with requirements for individual systems specified by a delta or an increment to the generic set.	Commonality and variation are documented explicitly, which will help lead to an architecture for the product line. New systems in the product line will be much simpler to specify because the requirements are reused and tailored.	Capturing requirements for a group of systems may require sophisticated analysis and intense negotiation to agree on both common requirements and variation points acceptable for all the systems.
Architecture: The architecture for the product line is the blueprint for how each product is assembled from the components in the asset base.	Architecture represents a significant investment by the organization's most talented engineers. Leveraging this investment across all products in the product line means that for subsequent products, the most important design step is largely completed.	The architecture must support the variation inherent in the product line, which imposes an additional constraint on the architecture and requires greater talent to define.
Software Components: The software components that populate the asset base from the building blocks for each product in the product line. Some will be reused without alteration. Others will be tailored according to pre-specified variation mechanisms.	The interfaces for components are reused. For actual components that are reused, the design decisions, data structures, algorithms, documentation, reviews, code, and debugging effort can all be leveraged across multiple products in the product line.	The components must be designed to be robust and extensible so that they are applicable across a range of product contexts. Variation points must be built in or at least anticipated. Often, components must be designed to be more general without loss of performance.
Performance modeling and analysis: For products that must meet real-time constraints (and some that have soft real-time constraints), analysis must be performed to show that the system's performance will be adequate.	A new product can be fielded with high confidence that real-time and distributed-systems problems have already been worked out because the analysis and modeling can be reused from product to product. Process scheduling, network traffic loads, deadlock elimination, data consistency problems, and the like will all have been modeled and analyzed.	Reusing the analysis may impose constraints on moving of processes among processors, on creation of new processes, or on synchronization of existing processes.
Business case, market analysis, marketing collateral, cost and schedule estimates: These are the up-front business necessities involved in any product. Generic versions that support the entire product line are built.	All of the business and management artifacts involved in turning out already exist at least in a generic form and can be reused.	All of these artifacts must be generic, or be made extensible, to accommodate product variations.
Tools and processes for software development and for making changes: The infrastructure for turning out a software product requires specific product line processes and appropriate tool support.	Configuration control boards, configuration management tools and procedures, management processes, and the overall software development process are in place and have been used before. Tools and environments purchased for one product can be amortized across the entire product line.	The boards, process definitions, tools, and procedures must be more robust to account for unique product line needs and for the differences between managing a product line and managing a single product.
Test cases, test plans, test date: There are generic testing artifacts for the entire set of products in the product line with variation points to accommodate product variation.	Test plans, test cases, test scripts, and test data have already been developed and reviewed for the components that are reused. Testing artifacts represent a substantial organizational investment. Any saving in this area is a benefit.	All of the testing artifacts must be more robust because they will support more than one product. They also must be extensible to accommodate variation among the products.
People, skills, training: In a product line organization, even though members of the development staff may work on a single product at a time, they are in reality working on the entire product line. The product line is a single entity that embraces multiple products.	Because of the commonality of the products and the production process, personnel can be more easily transferred among product projects as required. Their expertise is usually applicable across the entire product line. Their productivity, when measured by the number of products to which their work applies, rises dramatically. Resources spent on training developers to use processes, tools, and system components are expended only once.	Personnel must be trained beyond general software engineering and corporate procedures to ensure that they understand software product line practices and can use the assets and procedures associated with the product line. New personnel must be much more specifically trained for the product line. Training materials that address the product line must be created. As product lines mature, the skills required in an organization tend to change, away from programming and toward relevant domain expertise and technology forecasting. This transition must be managed.

Table 2-1. Costs and Benefits of Product Lines [31]

2.1.2 – Model-Based Development

Model-based development is a system development process organized around models of the desired, externally visible system behavior. These models form a complete system specification and can be used for analysis before implementation. Model-based software development plays an integral role in supporting core asset development for the software product line. Models are used to describe the externally visible behavior of the software components, thereby abstracting away design details and creating a formal requirements specification that can be tested, verified and validated before any design or implementation work has begun. These models can then be reused from one product in the line to the next, which, as outlined in [77], helps to alleviate some of the problems commonly associated with code reuse in software engineering. Although many software and system engineers associate the Unified Modeling Language (UML) with model-based development, creating software descriptions using this language does not necessarily indicate that the system was developed using a model-based approach. UML, which is discussed in greater detail in Section 2.1.4, is an informal, graphically-based language for modeling software systems and does not provide an executable or formally analyzable model that lay the foundation of model-based development. Examples of model-based development include Intent Specifications and SpecTRM-RL used to model TCAS II, a conflict avoidance system for the FAA [76], and ADAMS and POST used to model the Entry, Descent and Landing (EDL) system of the Mars Exploration Rovers (MER) [37].

One example of model-based development as applied to spacecraft software is described in [131] and called Component-Based Systems Engineering. This technique combines aspects of both systems engineering and Component-Based Software Engineering to reap the benefits of each technology without incurring some of the costs. Instead of reusing code, engineers reuse the development of requirements specifications, both informal and formal. In this approach, the reuse takes place before any detailed design is completed, any hardware built, or any software coded [131].

In this approach, components are created in a systems engineering development environment known as SpecTRM. SpecTRM is a toolkit that allows users to create intent specifications, which assist engineers in managing the requirements, design and evolution process [111]. The intent specification has a structure and incorporates a set of practices that makes the information contained in the document more easily used to support the project lifecycle. Intent specifications help engineers to (1) find specification errors early in product development so that they can be fixed with the lowest cost and impact to the system design, (2) increase the traceability of the requirements and design rationale throughout the project lifecycle and (3) incorporate required system properties into the design from the beginning rather than the end of development, which can be difficult and costly [76].

Component-Based Systems Engineering begins with a decomposition of the spacecraft, followed by a construction of components, subsystems and finally the entire system. These individual components are called SpecTRM-GSCs, or Generic Spacecraft Components, and provide users with a variety of benefits. The components are generic, which makes them highly reusable. Engineers can change project-specific information based on the instance of the component's use. They also contain component-level fault protection, laying the foundation for a fault protection scheme that parallels the spacecraft's development. Finally, the formal portion of the SpecTRM-GSC can be analyzed individually or as part of their parent subsystem- and system-level before any implementation has taken place. The feasibility of applying Component-Based System Engineering has been demonstration on SPHERES, a set of formation-flying satellites used as a test-bed for control and estimation algorithms on board the ISS [131].

Although Component-Based Systems Engineering is based on the traditional spacecraft decomposition described in Chapter 1, the incorporation of a model-based development strategy as well as system engineering tools provides an excellent basis for a new approach to spacecraft software development. However, the modeling of the spacecraft components is only one aspect of the spacecraft software development effort; it must be complemented by an infrastructure that supports the use of software product lines and software architecting.

2.1.3 – Safety-Driven Design

Traditional hazard analysis techniques do not take into consideration hazards that arise from unexpected software behavior or unexpected interactions between the various software components with each other and their environment. New hazard analysis techniques such as STPA [78] can be used to perform hazard analysis that takes software into consideration. Safety constraints need to be identified early in the software development process, recorded and managed by the system engineering and integration team and enforced throughout the software products. More information about safety-driven design can be found in [38]. It is important to note that this definition of safety refers to both human safety as well as mission assurance.

Most traditional hazard analysis techniques do not focus on accidents that result from inconsistencies between the model of the process used by the controllers (both human and automated) and the actual process state. Instead, they focus on failure events. In these event-based techniques, hazard analysis consists of identifying the failure events that can lead to a hazard, usually putting them into event chains or trees. Two popular techniques to accomplish this are Fault Tree Analysis (FTA) and Failure Modes and Effects Criticality Analysis (FMECA) [74]. Because of their basic dependence on failure events, neither does a good job of handling software or system accidents where the losses stem from dysfunctional interactions among operating components rather than failure of individual components [78].

STAMP (Systems-Theoretic Accident Model and Process) is a new model of accident causation based on systems theory rather than reliability theory. Accidents are assumed to be caused by dysfunctional interactions among components (including those related to component failures) that violate constraints on safe system behavior. The STAMP accident model is useful not only in analyzing accidents that have occurred but also in developing system engineering methodologies to prevent accidents [78]. Hazard analysis can be thought of as investigating an accident before it occurs. STAMP-Based Hazard Analysis (STPA) goes beyond component failure and is more effective than current techniques in protecting against system accidents, accidents related to the use of software, accidents involving cognitively complex human activities, and accidents related to managerial, organizational, or societal factors. The analysis starts with identifying the constraints required to maintain safety and then goes on to assist in providing the information and documentation necessary for system engineers and system safety engineers to ensure the constraints are enforced in system design, development, manufacturing and operations. It applies not only to the electro-mechanical parts of the system, but also to the software and organizational components, treating these aspects as part of the system as a whole.

This type of analysis has been demonstrated on a wide variety of complex systems including the Space Shuttle Columbia [38], the Walkerton E-Coli break-out [79] and a low-Earth orbiting, satellite [33]. The final example provides a before-deployment hazard analysis on a satellite that includes the analysis of the software components of the system. Based on the results of the experiences with using STAMP to model these systems, it is clear that this type of systems theoretic approach to safety engineering can be integrally entwined into the development process and can effectively include the software portions of the system. The complexity of space exploration systems requires this type of more powerful hazard analysis and safety design techniques than traditionally has been available.

2.1.4 – Software Process Models

Software process models describe the phased development of a software product, including the steps taken, the artifacts developed, and the criteria for moving from one phase to the next. The phases usually included in the software lifecycle are feasibility study / making the business case, requirements, specification, design, implementation, testing and maintenance. Depending on the particular product under development as well as the characteristics of the organization in which the product is being developed, different process models of these phases, their artifacts, and their transition conditions are used. Some of the most well-known process models include the waterfall model, rapid-prototyping, the incremental model, and the spiral model [130]. The following paragraphs describe these four process models in greater detail.

The waterfall is perhaps the most well-known and widely used software process model. As seen in Figure 2-2, the waterfall model is relatively simple; each phase of the software lifecycle strictly follows the previous stage. Verification and validation are performed at every phase to ensure that the documents produced during the current stage accurately reflect the user's requirements; consequently, the waterfall model is often referred to as a document-driven approach. Although often referred to as a "naïve" approach to software process, many software practitioners forget that the waterfall model allows for feedback between the various stages, thereby creating a more iterative approach to software process than normally thought of when referring to the waterfall model.

One of the drawbacks of the waterfall model is the difficulty that may be inherent in providing an accurate perception of the user requirements of many systems. When inadequate requirements analysis is performed, projects are often completed when the results do not fulfill the user's intended requirements. Consequently, the rapid-prototyping process model was proposed in which one or more software prototypes (usually focusing on the user interface) are created to ensure that the requirements specifications captured by the software engineers accurately reflects the desires of the customer.

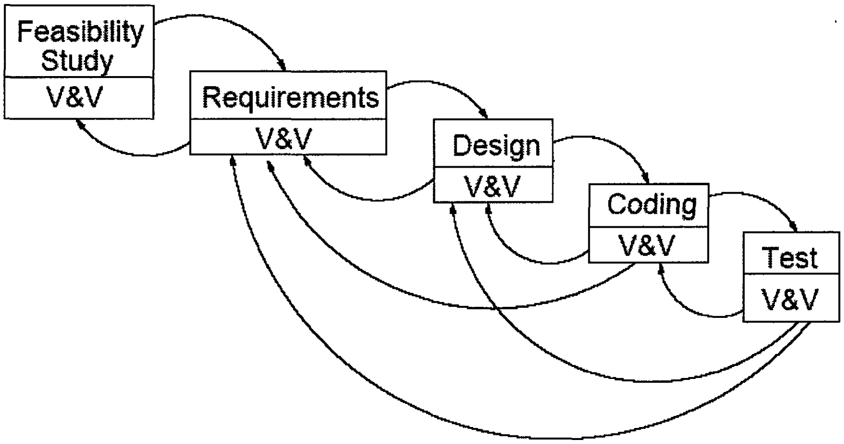


Figure 2-2. The Waterfall Model

Unfortunately, the rapid-prototyping technique is often badly misused; instead of "throwing away" the prototype after determining an accurate requirements specification, the prototype is used as part of the deliverable software. Since this software was not developed with the intention of delivery and was therefore not created under the same guidelines and standards as would be normally required, using the rapid-prototype can lead to catastrophic consequences, especially in the case of safety-critical systems.

Although both the waterfall model and the rapid-prototyping model have many redeeming qualities, i.e. verification and validation at each phase and feedback channels, these process models are far too simplistic for the type of large-scale, safety-critical systems required for a sustainable presence is space. An applicable and effective software process model for such systems needs to take into consideration the inevitability of changing and additional requirements as well as evaluation of software artifacts developed during each stage. There are two, more well-developed software process models that take into consideration some of the concerns of these types of systems. These models include the incremental and spiral models.

The incremental model combines aspects of both the waterfall and the rapid-prototyping models. In the incremental approach to software development (as depicted in Figure 2-3), software is delivered in small, but usable pieces known as increments. The software system produced during each increment is built upon the preceding increment. In other words, the incremental model, like rapid-prototyping, is iterative. However, unlike rapid-prototyping the result of each increment is a stand-alone, operational software product. The first increment is often called the core product and addresses the basic requirements of the entire software system. Each additional product is followed by the result of adding functionality during each increment until the final, completed product is delivered. This model is especially useful for projects with constrained resources, because each increment can be implemented with fewer people than if the entire software system was developed at once.

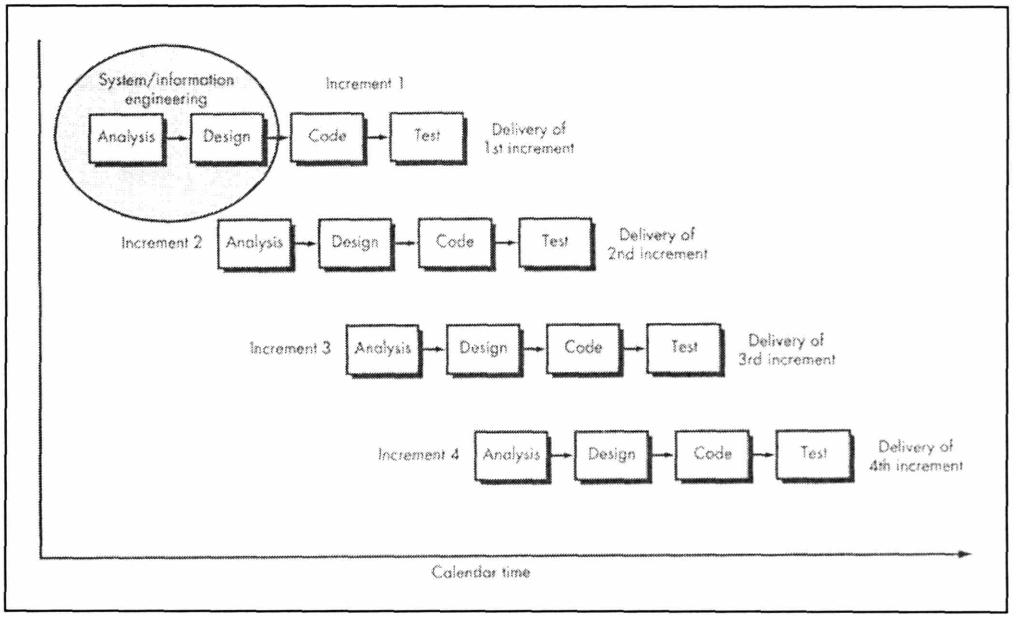


Figure 2-3. The Incremental Model [104]

One example of an incremental model is the Rational Unified Process, or RUP, developed by the Rational Software Corporation. This process model is component-based, which means that the software system is built up of components interconnected via well-defined interfaces and modeling using the Unified Modeling Language (UML), which was developed along with the RUP. Furthermore, the creators of the RUP state that its distinguishing aspects are that it is use-case driven, architecture-centric, and iterative and incremental. A use case is a piece of functionality in the system that gives a user a result. Use cases collectively make up the use-case model, which captures the functional requirements of the system. Use-case driven means that the development process proceeds from the identification, design and subsequent test-case construction from use cases. Use cases are developed in tandem with the system architecture. The architecture is comprised of a series of UML models (subsystems, classes and components) that describe the form of the software. Each phase of the process yields another aspect of the architecture, or artifact, as seen in Figure 2-4.

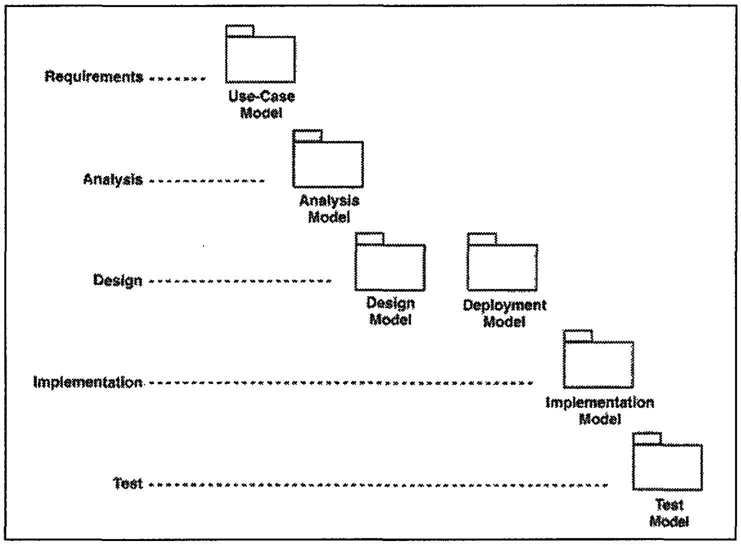


Figure 2-4. Workflows and Associated UML Models [19]

Finally, the Unified Process is iterative and incremental. Iterations refer to steps within the development of a particular component, while increments refer to the growth of the product as a whole. As seen in Figure 2-5, the various iterations that make up the development of one mini-project consist of five workflows – requirements, analysis, design, implementation and test – that take place over the four phases – inception, elaboration, construction and transition. The proponents of RUP approach contend that controlled iteration (1) reduces the cost risk to the expenditures on a single increment, (2) reduces the risk of not getting the product to market within the planned schedule, (3) speeds up the tempo of the whole development effort and (4) acknowledges the reality that user needs and the corresponding requirements cannot all be fully defined up front [63].

Like the other process models, the RUP also has drawbacks. It is often difficult to add functionality to a partial product, with subsequent high costs and low reliability of the system. It is also nearly impossible to guarantee safety in such an approach. In fact, it may be cheaper and result in a better product if the incremental components are thrown away and developed again from scratch.

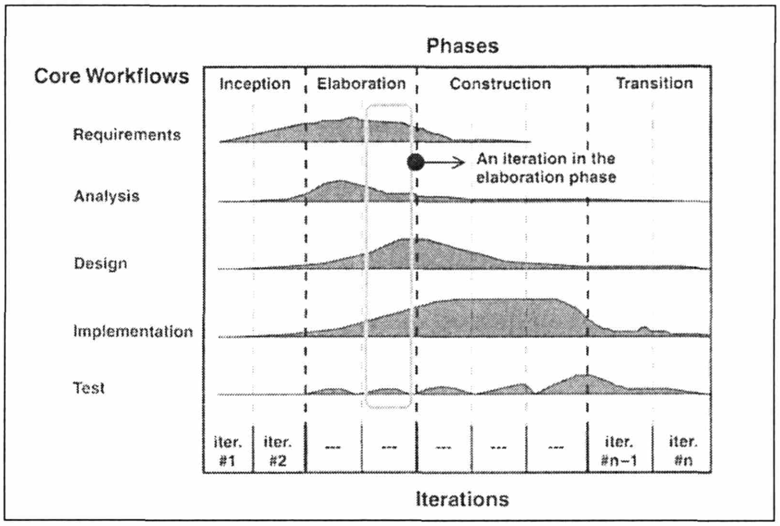


Figure 2-5. The Rational Unified Process [63]

The spiral process model is a risk-driven approach to software engineering that incorporates the other approaches and therefore can accommodate most previous models as special cases [16]. Consequently, the model guides software engineers toward which model is best for a given software situation because it becomes equivalent to one of the existing process models under certain circumstances, examples of which will be given later. Risk-driven means that the transition criteria for progressing from one stage of the spiral to the next are largely determined by the risk-assessment performed at the end of each stage. In addition, the spiral model is not linked to any particular design paradigm or implementation language (unlike the Rational Unified Process) and therefore does not impose an architecture, design or implementation on the system without scientific analysis and evaluation. Figure 2-6 presents a graphical depiction of the spiral process model.

The radius of the spiral indicates the cumulative cost incurred by completing steps to date. The angle indicates the progress made in completing each cycle of the spiral, where a cycle includes the following steps:

1. Identify the objectives for the portion of the product that is currently being worked on.
2. Identify the alternative means of implementing those objectives. For example, Design A, Design B, reuse from a previous project, COTS, etc.
3. Identify the constraints that are associated with each of these alternatives.
4. Evaluate the alternatives relative to the objectives and constraints identified in the previous steps. This evaluation will help to identify areas of uncertainty associated with specific alternatives that are significant sources of risk.

5. Perform a cost-effective strategy of risk-mitigation to deal with the remaining risk associated with the selected alternative(s). These strategies may include prototyping, simulation or benchmarking.
6. As the remaining risk decreases, move toward the right arms of the spiral. If the risk is high, multiple prototypes (rapid prototypes are just one risk-mitigation strategy – other methods can be used instead) may be necessary as seen in Figure 2-6.
7. Verification and Validation of the selected alternative(s) is performed.
8. Review any and all developments made during the cycle. Formulate a detailed plan of the next cycle. Participants must be committed to the next cycle approach before work commences. These plans may (and probably will) include a decomposition of the problem into smaller components. Each component will then operate in a parallel spiral model.
9. Repeat until the product is finished with an acceptable amount of risk.

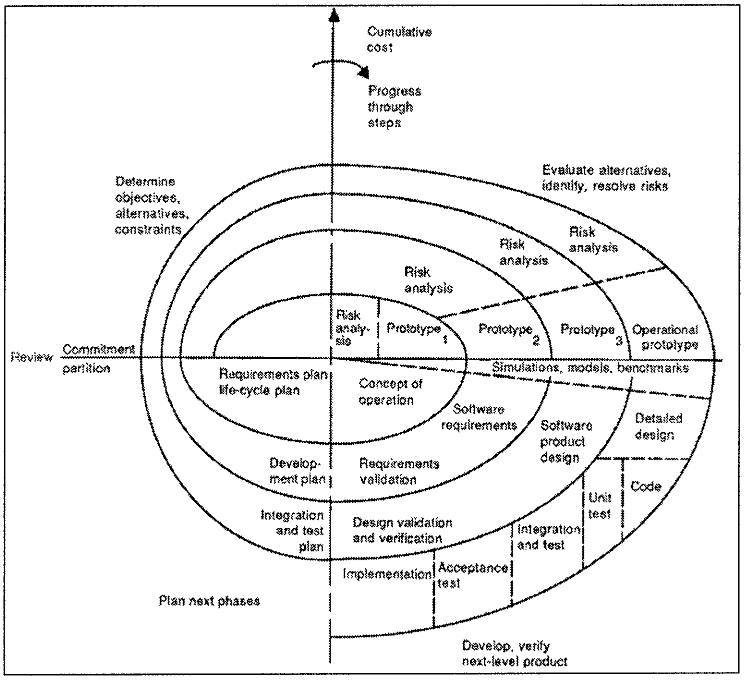


Figure 2-6. The Spiral Model [15]

Each cycle corresponds to a different step in the software engineering lifecycle. For example, Cycle 0 would be a Feasibility Study, Cycle 1 a Concept of Operations (Requirements Gathering) phase and Cycle 2 Top-Level Requirements Specification. These cycles can also be repeated if reworks or go-backs are required due to new risk issues that need resolution. Multiple prototypes can be built, as in the incremental model, if this is needed to reduce risk. In these ways the spiral model accommodates the positive features of the other process models without their difficulties, which Boehm claims are alleviated by the risk-driven approach. Furthermore, numerous instances of the spiral model progress simultaneously – usually one spiral model is used to track and guide the development of the system as a whole, while many other spiral model instances are created to describe the process of subsystems, components, or any other modules that derive from the system's decomposition.

Since it was originally introduced in the late 1980s, Boehm has created new versions of the spiral model, one of which is called the WINWIN Spiral Model and is shown in Figure 2-7. This software process model focuses on negotiation between the customer and developer. Successful negotiation occurs when both sides "win" through achieving a "win-win" result. For example, the customer may have to balance functionality and other system characteristics against cost and time to market [16], [17]. Rather than a single requirements elicitation phase from the customer, the following activities are performed during each pass around the spiral: (1) identification of the system or subsystem's key "stakeholders," (2) determination of the stakeholders' "win conditions," and (3) negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the project team) [17]. In other words, the WINWIN model is a means by which traditional system engineering requirements practices can be integrated into the spiral model.

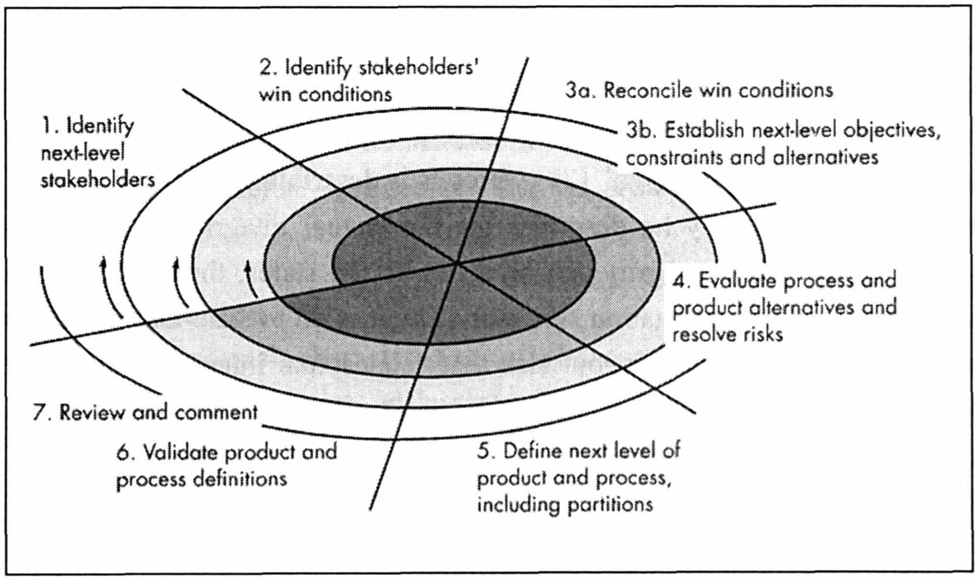

1. Identify next-level stakeholders
2. Identify stakeholders' win conditions
3a. Reconcile win conditions
3b. Establish next-level objectives, constraints and alternatives
4. Evaluate process and product alternatives and resolve risks
5. Define next level of product and process, including partitions
6. Validate product and process definitions
7. Review and comment

Figure 2-7. The WINWIN Spiral Model [104]

There are also several potential limitations to the both the spiral model and the WINWIN model. First, both models require a long development process with the potential for much unnecessary iteration, leading to increased development cost and time. In addition, performing accurate risk analyses is a difficult task, especially for software where budget and time-to-market estimates have been notoriously inaccurate. Finally, because the spiral models are instantiations of the other process models, they also have the limitations of those models. For example, if rapid-prototyping is used to evaluate requirements in the spiral model, then the rapid prototype can also be used for the final product, just as in the traditional rapid-prototyping approach.

Due to the limitations of current software process models, especially with respect to component-based and safety-critical systems, a new process model is needed. This process model must focus on both component-based development as a part of a product-line approach and integrating safety engineering techniques into the software development process. The new process model is described in Chapter 3.

2.1.5 – Software Architecture

The field of software architecture has emerged over the past decade as an important and integral step in the software development lifecycle. The term software architecture has been defined in a number of ways, by various experts in the field. Since as early as 1976, computer scientists have recognized the need to select a software architecture in the early stages of design [133]. This architecture selection should result from a careful investigation of all possible approaches and should not be influenced by hardware or implementation considerations [133].

In his essay on software design entitled "Aristocracy, Democracy, and System Design" from the seminal work *The Mythical Man Month* (a collection of essays describing various aspects of software engineering), Fred Brooks describes an aspect of software engineering known as "conceptual integrity" [21]. He uses the Reims cathedral in France as an example of conceptual integrity, stating "...eight generations of builders, [each of which] sacrificed some of his ideas so that the whole might be of pure design" [21]. Brooks is describing the architecture of the Reims cathedral, and in the same way he describes the conceptual integrity, or architecture, of the software system. Conceptual integrity can be achieved, he states, through a separation of the architectural effort from implementation concerns. He goes on to state that it is not only crucial to have a software architecting effort, but also that "conceptual integrity is the most important consideration in system design" [21]. Clearly, software architecting was recognized early on as an important part of the software engineering effort.

Perry and Wolf formalized the idea of software architecture as an independent study area in software engineering and proposed the first definition:

> "Software architecture consists of elements, form and rationale, where the elements are either processing, data or connecting elements, form is defined in terms of the properties of and the relationships among the elements, that is the constraints on the elements and the rationale provides the underlying basis for the architecture in terms of the system constraints which most often derive from the system requirements" [102].

Garlan and Shaw define software architecture as, "[involving the] description of elements from which systems are built, interactions among those elements, patterns that guide their composition and constraints on those patterns" [119]. Another definition of software architecture builds upon the two previous definitions: "The software architecture of a program or computing system is the structure or structures of the system which comprise software elements, the externally visible properties of those elements and the relationship among them" [9].

Although the above definitions differ in their interpretation of what software architecture involves, they are all based on the same theme and have distinct commonalities; the definitions all focus on the components and connectors that comprise a software system and the rules that govern their assembly. It is also important to note that the software architecture artifact is not a static entity – it evolves over time throughout the increments and iterations of the software development lifecycle.

Furthermore, software architecture can be subdivided and defined at two different levels of abstraction: *conceptual* and *realizational.* The two levels of software architectures are developed at different stages in the software lifecycle when the software system is at different levels of maturity. Conceptual software architecture is the level of abstraction used during requirements analysis and specification – it merely holds a representation of a model of the architecture with the purpose of explaining or understanding how the functional requirements are allocated to collaborating structures [21]. Conceptual architectures are often represented as box and line diagrams. A software product can have several conceptual architectures. On the other hand, a realizational architecture is at a much lower level of design. Realizational software architecture is directly visible in a representation of a model that corresponds directly to source code [21]. A single conceptual architecture can be used to generate several realizational architectures.

2.1.5.1 – Quality Attributes

Quality attributes are external characteristics of the software that can be measured only indirectly. For example, correctness, efficiency and maintainability are all quality attributes desirable for different types of software. Although maintainability cannot be measured directly, the ease with which a change can be made in the software can be measured, for example, by determining how coupled the changed module is to other modules in the system. This measurement of coupling can give an indication of how easy the system will be to maintain. It is the mapping of a system's functionality onto software structures that determines the architecture's ability to support quality attributes. Consequently, it is the job of the software architect to create software structures that support the quality attributes desired of the system. It is important to note that although a good architecture cannot completely guarantee that all quality goals are achieved (poorly executed downstream development efforts may still sabotage the realization of non-functional requirements), a poor architecture selection almost always ensures that the quality goals are not achieved [135]. In other words, software architecture alone cannot achieve quality attributes; it can only provide the foundation.

Support may be achieved through many possible structures, or combinations of structures, called "tactics" by researchers at the Software Engineering Institute (SEI) [9]. A tactic is a design decision that is influential in the control of a quality attribute response; tactics provide insight into what software engineers can do in order to affect a quality attribute response measure [27]. A quality attribute response is the response of the software architecture to a quality-related scenario (used in software architecture evaluation); the quality attribute response measure is a measurable quantity for evaluating that response, such as latency or throughput [27].

Tactics are independent of any specific system. For example, software architects can decompose the system to ease parallel development or increase hardware redundancy to increase availability. It is important to note, however, that adding redundancy creates a need for synchronization; this trade-off point results from tactics that affect more than one quality attribute and are often referred to as sensitivity points [9]. Sensitivity points are places in a specific architecture to which a specific response measure is particularly "sensitive" (that is, a little change is likely to have a large impact) [27].

Consequently, if achieving the quality attributes of a software system is largely dependent on the construction of the software architecture, then the architecture must be evaluated to ensure its appropriateness in achieving those attributes. In other words, software architecture should be evaluated to determine if it provides a suitable framework for solving the problem at hand.

2.1.5.2 – Software Architecture Styles

Given a set of high-level requirements, both functional and non-functional, architecture *styles* can be identified that will help in constructing an overall software architecture that caters to these particular requirements. Architectural styles are idiomatic patterns of system organization that are developed to reflect the value of specific organizational principles and structures for certain classes of software [130]. Styles define a vocabulary of component and connector types and a set of constraints on how they can be combined. When selecting a certain architecture style with its corresponding engineering principles and materials for a given project, architects are guided by the problem to be solved as well as the larger context in which the problem occurs. Consequently, certain architectural styles are more appropriate for use on a particular project than others; the styles either support or detract from achieving the requirements and quality attributes desired by the stakeholders [118].

Garlan and Shaw have identified and classified a number of software architecture styles in [51] and [119]. Several examples include layered, implicit invocation, process model and abstract data types. Figure 2-8 shows an example of a software architecture style description and its classification [118]. The format of this architecture style description is adapted from [3] and [4]. As described in Figure 2-8, a layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. The most common use is in operating systems. The main benefits of using a layered architecture include support for abstraction, enhancement and reuse [51]. Implementers that use a layered approach can partition a complex problem into a sequence of incremental steps, from low-level details to high-level programs that interface with users. Abstraction of detail makes the high-level programs easier to implement and use. Enhancements to the higher-levels of the architecture are simple, because they merely build on the software that already exists at the lower levels, the details of which have been abstracted. Finally, reuse is supported because different implementations of the same layer can be used interchangeably. Some of the more well-known problems with layered architectures include close coupling between high-level functions and their low-level implementations and difficulty in finding the right level of abstraction for each desired function.

Style: Layered

Problem: We identify distinct classes of services that can be arranged hierarchically. The system is depicted as a series of hierarchical layers, one on top of the next, where services in one layer depend upon, or call services from layers at a lower level in the hierarchy. Quite often these systems are partitioned into three layers – one for basic services, one for general utilities and one for application-specific utilities.

Context: Each class of service has to be assigned to a specific layer. It may occasionally be difficult to properly identify the function of a layer succinctly and, as a consequence, assign a given function to the most appropriate layer. This holds more if we restrict visibility to just one layer.

Solution:

System Model: The resulting system consists of a hierarchy of layers. Usually, visibility of inner layers is restricted.

Components: The components in each layer usually consist of collections of functions.

Connectors: Components generally interact through procedure calls. Because of limited visibility, the interaction is limited.

Control Structure: The system has a single thread of control

Variants: A layer may be viewed as a virtual machine, offering a set of 'instructions' to the next layer. Viewed thus, the peripheral layers get more and more abstract. Layering may also result from a wish to separate functionality, e.g. into a user-interface layer and an application-logic layer. Variants of the layered scheme may differ as regards to the visibility of components to the top layers. In the most constrained case, visibility is limited to the next layer up.

Common Uses: Operating Systems, Communication Protocols

Figure 2-8. Layered Architecture Style Characterization [118]

Given these characteristics, several application domains can be identified to which a layered architecture is both an appropriate and inappropriate solution. For example, the most widely known examples of this architectural style are layered communication protocols. Other application areas for this style include database and operating systems. But not all systems are easily structured in a layered fashion, and therefore a layered architecture style may not be an appropriate solution. For example, a data-centric application in which sequential information processing takes place would benefit not from a layered style, but from a pipe-and-filter style [51].

In order to determine which styles are most appropriate for a particular application, the characteristics of the application domain must be observed and studied, and techniques suitable for dealing with these characteristics employed [118]. More sophisticated methods of software architecture evaluation have been created to help create a process for determining appropriate architecture solutions. These methods are described in the following subsection.

2.1.5.3 – Systems Engineering Architecture Evaluation

Traditional systems engineering activities have involved the concept of trade studies and scientific evaluation for many years. Evaluation of possible architectures begins during concept exploration and refinement studies and continues throughout the system test and evaluation phases. In other words, evaluation is a continuous process that begins during conceptual design and extends through the product use and logistic support phase until the system is retired [32]. Figures 2-9 and 2-10 illustrate the process of evaluating alternative architectures and evaluating the selected system architecture with respect to the requirements.

As outlined in [12], Figure 2-9 illustrates the first three steps of the System Design and Development phase of the production lifecycle. These steps include:

> "1. Establishing criteria (qualitative and quantitative technical parameters, bounds and constraints) for system design.
>
> 2. Evaluating different alternative design approaches through the accomplishment of system/cost effectiveness analyses and trade-off studies."

Furthermore, during the preliminary system analysis portion of system design and development, the following activities take place: (1) defining the problem, (2) identifying feasible alternatives, (3) selecting the evaluation criteria, (4) applying the modeling techniques, (5) generating input data and (6) manipulating the model [12]. Distinct alternative designs exist during these early stages, and it is important to determine how these partial designs are evaluated to determine the superior approach [128]. Figure 2-10 illustrates the process of evaluation of alternatives used in systems engineering.

The evaluation of real options shown in Figure 2-10 will help answer the following three questions:

> "1. How much better is the selected approach than the next-best alternative? Is there significant difference between the results of the comparative evaluation?

2. Have all feasible alternatives been considered?
3. What are the areas of risk and uncertainty?” [12]

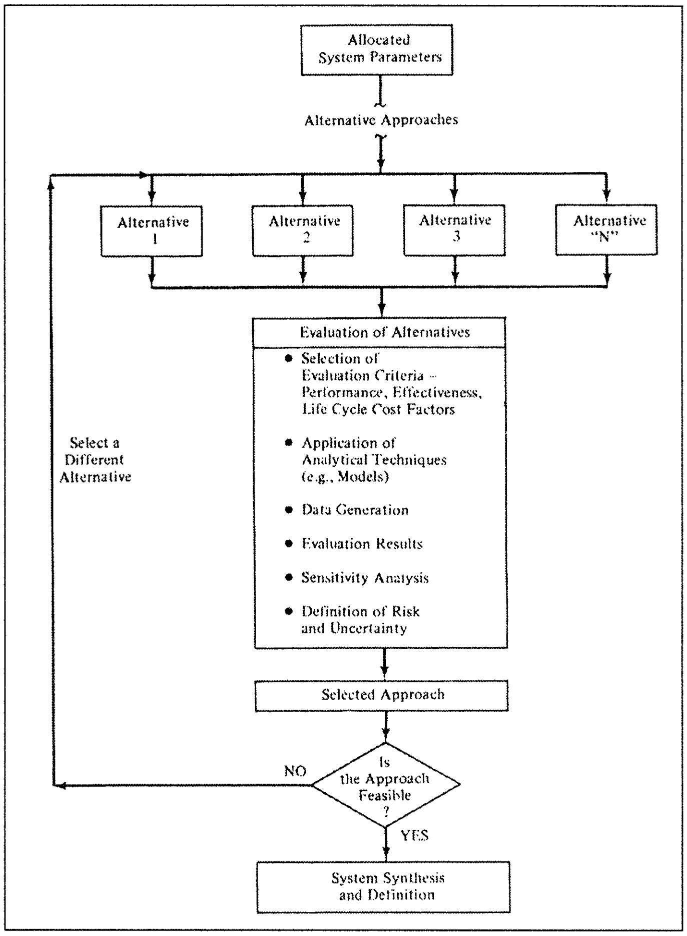

Allocated System Parameters
Alternative Approaches
Alternative 1
Alternative 2
Alternative 3
Alternative "N"
Evaluation of Alternatives
• Selection of Evaluation Criteria – Performance, Effectiveness, Life Cycle Cost Factors
• Application of Analytical Techniques (e.g., Models)
• Data Generation
• Evaluation Results
• Sensitivity Analysis
• Definition of Risk and Uncertainty
Select a Different Alternative
Selected Approach
NO
Is the Approach Feasible ?
YES
System Synthesis and Definition

Figure 2-9. Evaluation of Alternatives [12]

This portion of system design and development is also often referred to as concept generation and selection. During concept generation, engineers identify a set of customer needs and specifications and generate a set of product concepts that satisfy those needs and requirements [82]. Concept generation is followed by concept selection in which the concepts are ranked, possibly combined and improved and finally one or more concepts is selected for elaboration into the system architecting process [106]. System architecting involves the assignment of functional elements of the product to the physical building blocks of the product through the concept developed in the previous stage of development. The architecture therefore becomes the central element of the product concept [106].

These activities continue throughout system design and development and culminate in the evaluation activities outlined in Figure 2-10 during system test and evaluation. Through prediction, analysis and measurement, the true characteristics of the system are compared to requirements of the overall system developed during conceptual design in order to demonstrate that the system fulfills its intended mission [12]. This approach allows for corrective actions to be incorporated into the architecture relatively easy and at little cost. "Changes will usually be more costly as the life cycle progresses" [12].

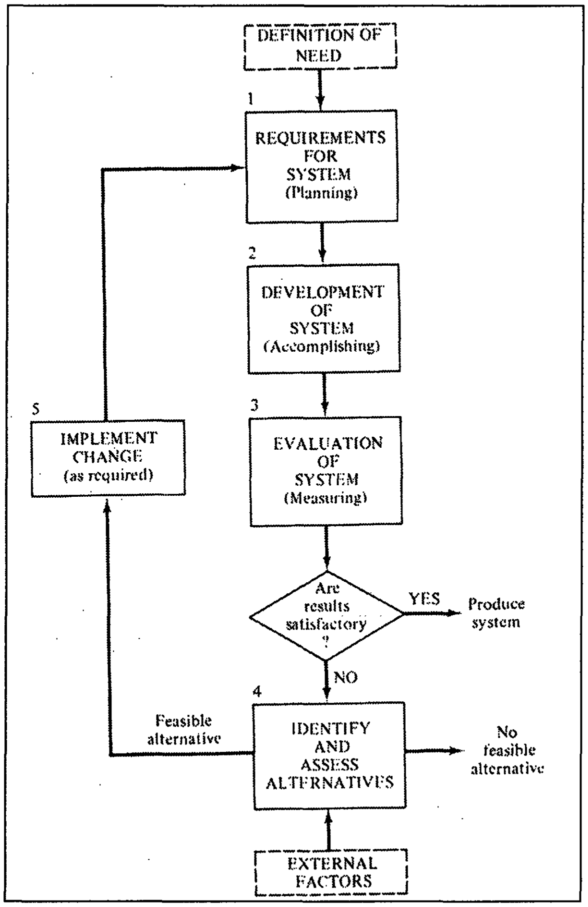

DEFINITION OF NEED
1
REQUIREMENTS FOR SYSTEM (Planning)
2
DEVELOPMENT OF SYSTEM (Accomplishing)
3
EVALUATION OF SYSTEM (Measuring)
5
IMPLEMENT CHANGE (as required)
Are results satisfactory ?
YES
Produce system
NO
4
IDENTIFY AND ASSESS ALTERNATIVES
Feasible alternative
No feasible alternative
EXTERNAL FACTORS

Figure 2-10. System Requirements and Evaluation [12]

Rechtin claims that some of the reasons why software engineering is in a relatively poor state when compared to other forms of engineering is "the failure to make effective use of methods known to be effective" and "the lack of an architectural perspective and the benefits it brings" [82]. Rechtin further notes that existing software modeling techniques fail to capture "ill-structured" requirements such as modifiability, flexibility and availability and therefore these qualities cannot be transformed into implementation [82]. Furthermore, the primary focus in developing and modeling the software architecture (often referred to as architecting) should not be on the structure of the problem that the software is to solve, but the structure of the software itself. In other words, software architects should not fit the architecture of a solution to many different problems regardless of their suitability to addressing the characteristics of that problem

[82]. Instead software engineers should be providing insight into the problem to ensure that the right problem characteristics are considered, the architecture caters to those characteristics and disparate requirements are reconciled [82]. These activities can be accomplished through a more rigorous approach to software architecture evaluation that is based on the design techniques that have been used in system engineering for many years.

2.1.5.4 – Software Engineering Architecture Evaluation

This subsection describes the current software architecture analysis methodologies employed in both the research arena and in industry. As previously mentioned, software architecture evaluation (SAE) techniques can be classified with respect to their phase, approach and abstraction. Table 2-2 contains a sample listing of current SAE techniques and their classifications. The following paragraphs describe these techniques in greater detail. This list is not exhaustive, but it includes some of the more notable architecture evaluation techniques as well as the more novel approaches. Several papers have been written that provide classifications of other SAE techniques [36].

SAE Technique	Phase	Approach	Abstraction
ATAM	Late: Middle	Scenarios	Particular System
ARID	Early	Scenarios	Particular System
CBAM	Late: Middle	Scenarios	Particular System
CPN	Late: Middle	Metrics	Domain
QDS	Early	Metrics	Particular System
Empirical	Late: Post-Deployment	Metrics	Quality Attribute (Maintainability)

Table 2-2. Software Architecture Evaluation Techniques and Classifications

The first documented and widely publicized architecture analysis method, called the Software Architecture Analysis Method (SAAM), supports expert assessments of software architectures that were effectively untestable [30]. SAAM was originally created to evaluate architectures with respect to modifiability, but since then has been extended to assessing other quality attributes including portability, extensibility and integrability. It eventually evolved into the most well known architecture evaluation technique, called the Architecture Trade-off Analysis Method, or ATAM [67]. Figure 2-11 illustrates the basic steps in this methodology.

An ATAM evaluation begins with the elicitation of business drivers and software architecture from the project management. The information is refined through expert, architect and stakeholder opinion into scenarios and the architectural decisions that were made in support of those scenarios. The scenarios are then analyzed to identify risks, non-risks, sensitivity points and trade-off points in the architecture. Risks are synthesized into a set of risk themes, showing how each one threatens a business driver [30]. The ATAM evaluation culminates in a written report that includes the major findings, which typically include:

1. the architectural styles identified,
2. a "utility tree," or hierarchical model of the driving architectural requirements,
3. the set of scenarios generated and the subset that were mapped onto the architecture,
4. a set of quality-attribute specific questions that were applied to the architecture and the responses to these questions,
5. a set of identified risks, and
6. a set of identified non-risks [30].

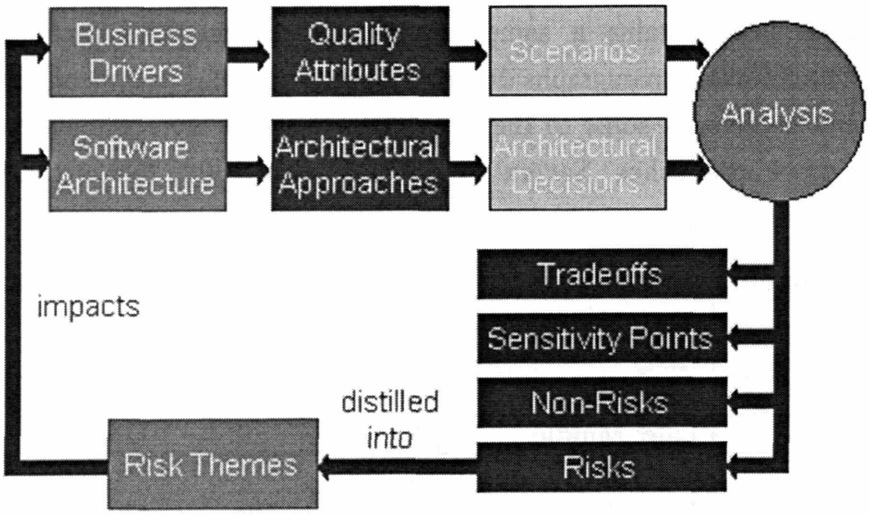


Figure 2-11. The Architecture Trade-off Analysis Method [28]

Another approach is called ARID, which stands for Active Reviews for Intermediate Designs. Whereas ATAM is used to analyze a complete software architecture, ARID can be used in the earlier stages of architecture development when designers need insight into portions of their early design strategies, before a comprehensive analysis can be performed [30]. ARID is a combination of active design reviews and scenario-based architecture evaluation. It consists of the following steps:

1. The designer works with the evaluation facilitator to identify a set of reviews including the software engineers who will be expected to use the design.

2. The designer briefs the review team on the design and walks through examples of using the design.

3. The reviewers brainstorm scenarios for using the design to solve problems they expect to face during detailed design activities, implementation, and testing. These scenarios are

prioritized and a set is chosen that defines what it means for the design to be usable. If the design performs well under this set, it has passed the review

4. Finally, the reviewers jointly write code or pseudo-code using the design to solve the highest priority scenario to gain insight into any issues or problems [30].

Whereas ATAM and ARID provide software architects with a framework for understanding the technical trade-offs they face during design and maintenance, CBAM (Cost-Benefit Analysis Method) provides guidance for understanding the economic and business-critical design decisions associated with the software architecture [68]. CBAM considers both the costs and the benefits of architectural decisions; for example, long-term maintenance costs are taken into consideration as well as the return on investment of any architectural decision [28].

The cubes labeled P, A, S, and M represent architectural decisions that have been made (or are being considered) for the system for Performance, Availability, Security and Modifiability respectively. CBAM helps engineers determine how these decisions link to the business goals and quality attributes desired by the stakeholders [28]. Given this information, the stakeholders can decide among potential architecture decisions that maximize their return on investment [30].

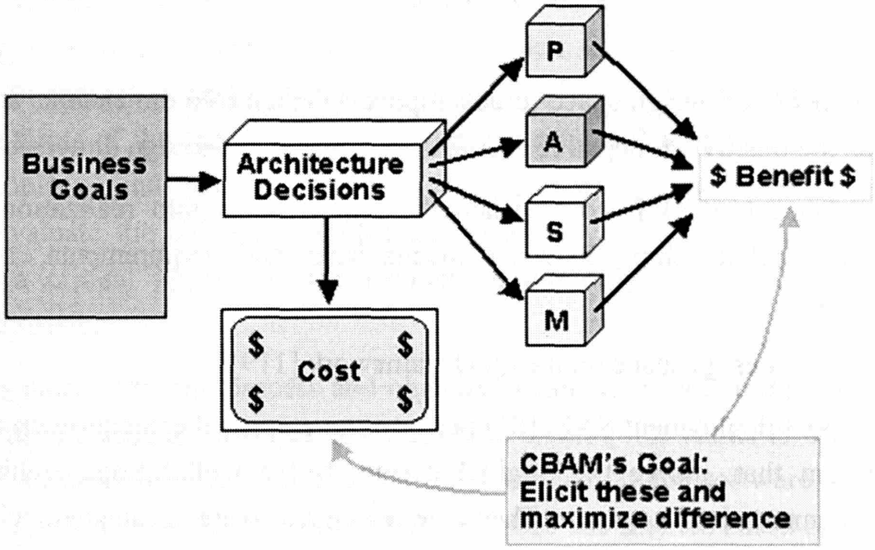


Figure 2-12. Cost-Benefit Analysis Method [28]

Another interesting approach to evaluating architectures prior to implementation involves the use of Coloured Petri Nets (CPNs) [46]. CPNs carry computed data values through tokens that result from the evaluation of expressions attached to the arcs of the net. When these expressions represent metrics related to a quality attribute of the architecture, the CPN can be executed to evaluate the architecture with respect to that quality attribute. An example application of using the CPN technique on networking software, measuring for reliability, security, and efficiency can be found in [46].

The Quantified Design Space (QDS) is an approach to software architecture evaluation that parallels trade-space analyses of traditional system engineering. QDS builds on the notion of design spaces and quality function deployment (QFD) and provides a tool for evaluating the strengths and weaknesses of a set of architectural styles given a set of requirements in a particular application domain [119]. The QDS provides a mechanism for translating system requirements into functional and structural design alternatives and then analyzing these alternatives quantitatively [119]. A design space is a multidimensional space that classifies system architectures, where each dimension describes a variation along a specific characteristic of design decision. A specific system, therefore, corresponds to one point in the design space, associated with the dimensional values that correspond to its characteristics and structure [119]. Design spaces provide functional and structural design guidance through illustrating which choices are most likely to meet the functional requirements for a new system. Quality function deployment is a quality assurance technique that supports the translation of requirements into realization mechanisms at each stage of product development. Through combining these two techniques, the quantified design space approach emerges. In short, the QDS process follows these steps:

1. Use the concept of design space to decompose a design into dimensions and alternatives as well as the positive or negative correlations between the design dimensions.
2. Use the concept of QFD to translate the requirements into realization mechanisms, analyze the relationships between mechanisms and requirements, and determine correlations.
3. Implement the design space on the QFD framework [119].

In an example of post-deployment SAE, [80] performs an empirical evaluation on a redesign of a client-server system that, unlike the original design, had a well-defined architecture before design and implementation began. These architectures were evaluated with respect to maintenance. The authors approached the evaluation scientifically, following the recommended best practices for SAE outlined in [2]; they determined the purpose and process of the evaluation and came up with a set of metrics to quantitatively evaluate the maintainability of the original system, the redesign of the system, and the actual implementation of the redesign. They found that the redesign and the actual implementation of the redesign were more maintainable than the original design according to their defined metrics because of the component-connector relationship of the new, and more appropriate, architecture [80].

2.1.5.5 – Limitations of Current Architecture Evaluation Techniques

Although there has been significant progress in the area of software architecture and evaluation over the past two decades, the techniques currently being explored and employed have several short-comings. There are four main areas that require further examination in order to make an effective software architecture development and evaluation process within the software development lifecycle, especially with respect to software architecture for embedded, safety-critical systems. First and foremost, most approaches to software architecture evaluation evaluate the architecture with respect to only one metric such as performance [135], complexity [138], testability [44] or maintainability [80]. Very few architecture evaluation techniques take into consideration multiple criteria and the trade-offs that must occur to balance the achievement of those requirements and quality attributes.

Second, the evaluation is performed after the architecture has been developed; the evaluation does not coincide with the development process. Consequently, the intermediate conceptual software architectures have not been analyzed with respect to their appropriateness in fulfilling requirements and quality attributes, thereby possibly resulting in a sub-par realizational architecture.

Third, as discussed in the previous section, many of the current software architecture evaluation techniques focus on scenarios as the means by which to evaluate different aspects of the architecture with respect to quality attributes. Expert opinion is used to qualitatively judge the architecture in terms of the scenario and the quality attributes that are affected by that scenario. While expert opinion and analysis is an important part of any evaluation process, it is also important to evaluate the software architecture objectively. In addition, the selection of the scenarios has a critical impact on the evaluation, but only a limited number of scenario evaluations are feasible.

The need for a more scientific process and objective evaluation tools brings forward the fourth limitation of current software architecture evaluation techniques, which is the loose use of the term "metrics" to describe some of the criteria for evaluating the software architectures. For example, techniques such as the Quantified Design Space use metrics based on expert opinion to evaluate the architectures after a realizational architecture has been decided upon. Consequently, numbers are assigned to the various requirements, but these numbers are quite arbitrary and dependent on the expert assigning them. If true software metrics are to be used in analyzing software architectures, it is important that these metrics provide an accurate assessment of the architecture with respect to a particular quality attribute. For example, in [80] SAE is performed on a post-deployment architecture with respect to maintenance and defines several coupling metrics, such as coupling-between-modules and coupling-inside-modules, as the evaluation metrics.

2.2 – Related Research in Spacecraft Software Architecture

One of the most well known efforts in shifting the approach to spacecraft software architecture is the Remote Agent experiment flown on NASA's Deep Space One (DS-1) mission to demonstrate the use of AI controllers. The Remote Agent architecture consists of a Smart Executive that executes plans and fault recovery strategies, monitors constraints and runtime resource usage, and coordinates the top-level command loop; a Planner/Scheduler that coordinates ground-supplied mission goals with the current spacecraft state and produces a set of time-delimited activities for the Smart Executive to perform; and a Mode Identification and Reconfiguration module that performs model-based diagnosis and recovery based on the spacecraft state [64]. Although a traditional AI approach to software architecture, the Remote Agent was the first new spacecraft software paradigm to be tested by NASA.

The success of this experiment laid the foundation for JPL's most ambitious software engineering endeavor, MDS, which is a concerted effort by NASA to fully change their approach to spacecraft software development. The Mission Data System (MDS), created by JPL for NASA, is a software architecture for the next generation of deep-space (outside of Earth-orbit) spacecraft. A graphical depiction of the MDS software architecture can be seen in Figure 2-13. MDS is a goal-based system, which is defined as a system in which all actions are directed by goals instead of command sequences. A goal is a constraint on a state over a time interval [61]. Types of states include:

- Dynamics – vehicle position and attitude, gimbal angles, wheel rotation
- Environment – ephemeris, light level atmospheric profiles, terrain
- Device status – configuration, temperature, operating modes, failure modes
- Parameters – mass properties, scale factors, biases, alignments, noise levels
- Resources – power and energy, propellant, data storage, bandwidth
- Data product collections – science data, measurement sets
- Data management and transport policies – compression, deletion, transport priority
- Externally controlled factors – spacelink schedule and configuration

In MDS, the hardware adapter receives information about the environment and the hardware itself from the sensors. These measurements are used as evidence (the sensors may not be functioning properly) and are passed to a state estimator. The state estimators use the evidence provided by the hardware adapter, the history of the states, future predictions and plans, and a degree of uncertainty to estimate the current state of the system. The states combine to form a current model of the system. The states are passed to the controller along with operator goals.

Operators express their intent in the form of goals declaring what should happen, as opposed to low-level command sequences that dictate how the intent is to be achieved. The goals and constraints are submitted to the spacecraft controller through a telecommand. These goals and constraints along with the current state of the system as measured by the state estimator, the

Mission Planning and Execution function elaborates and schedules the goals. It is in the Mission Planning and Execution function that AI can be employed (it does not need to be) for elaboration and scheduling the plan.

Goal elaboration is the process by which a set of rules iteratively expands a high level goal into a goal network. The new low-level goals are then scheduled (to eliminate state conflicts) and merged with the previous goals. Flexibility can be built into the goal network, providing the spacecraft with the ability to deal with previously unknown situations. The plan defined by the goal network is executed by issuing goals to the controllers, which issue appropriate commands to the hardware to achieve these goals.

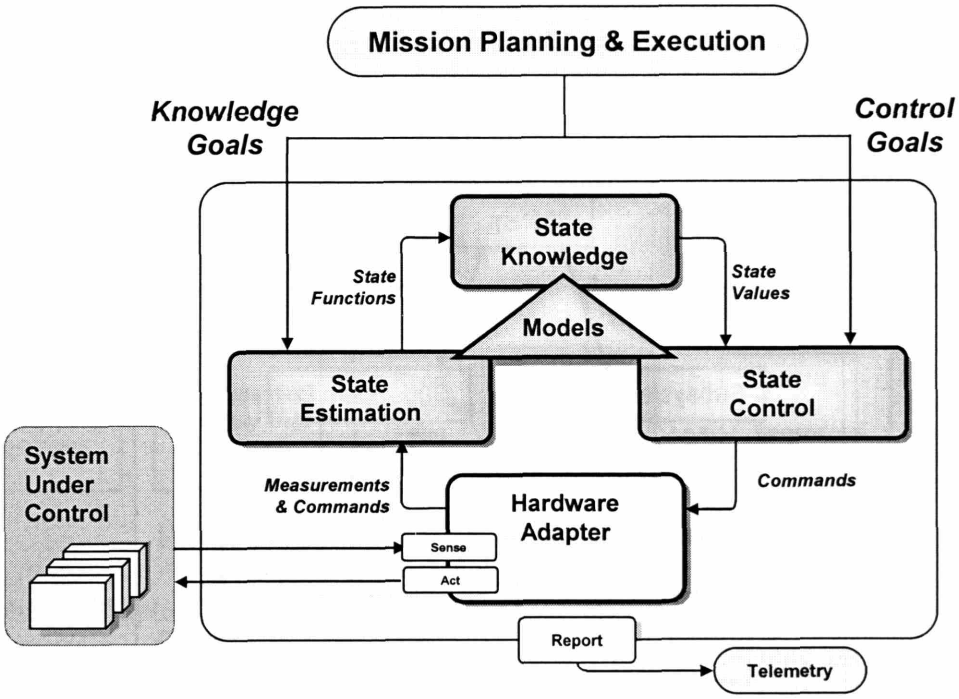


Figure 2-13. The Mission Data System Software Architecture [62]

The proponents of MDS suggest that the problems with traditional software (complexity, brittleness of command sequences, and inability to handle faults) are easily dealt with through the use of such an architecture. First, the entire architecture is reused from one spacecraft to the next; the aspects of MDS that change from one mission to the next are the goals for a particular mission and the design of the control loops (state variables, estimation and control algorithms and hardware adapters) [40]. Second, fault tolerance is no longer considered a separate entity. Because a failure mode or other anomalous condition is treated as just another possible set of system states, the spacecraft does not have to alter its nominal operations. It simply relegates the fault by attempting to fulfill its mission goals given its current state, a process no different than if the spacecraft was not in a fault state [40]. This proposed approach to the software specification, design and implementation leverages the ability to reuse the architecture and many of the goals from mission to mission, thereby decreasing the cost of spacecraft software development.

2.2.1 – Critique of Current and Proposed Spacecraft Software Architectures

As discussed in Chapter 1, current spacecraft all follow a similar pattern of software architecture: they are decomposed first into a set of subsystems and then the components that comprise those subsystems. Traditionally, this software architecture, or method of decomposition, has been referred to as functional decomposition and is often depicted as in Figure 2-14. There are two major fallacies with this approach to spacecraft software architecture. First, as mentioned in the Introduction, this type of architecture does not take into consideration the complex interactions between subsystems and components that result from the resource limited environment of space and the physics of the space environment.

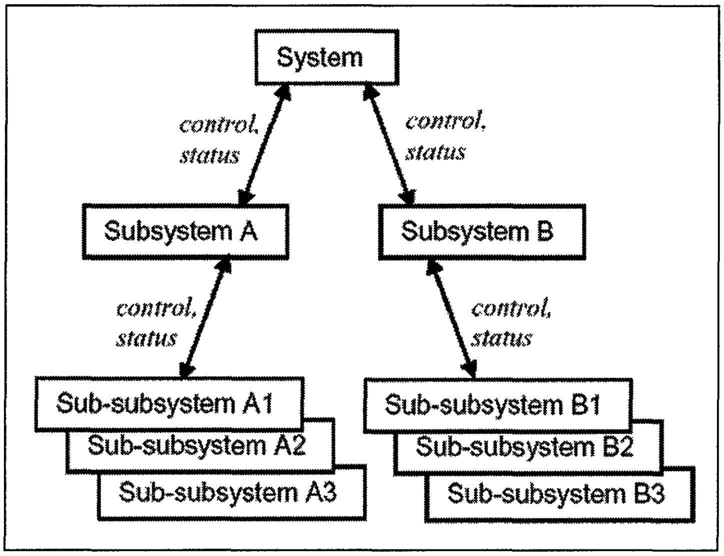


Figure 2-14. Typical Spacecraft Software Decomposition [39]

Second, the typical spacecraft software decomposition shown in Figure 2-14 is not actually a functional decomposition, but a structural or subsystem decomposition. In pure functional decomposition, systems are decomposed with respect to the functions they must provide in order to achieve their goals. For example, instead of the spacecraft being decomposed into subsystems like Thermal or Guidance, Navigation and Control and then components like temperature sensor and reaction wheels, the spacecraft is decomposed into functions such as "set temperature" or "maintain orbit." These functions are then further decomposed into sub-functions as seen in Figure 2-15. Set temperatures is decomposed into "Read Temperature Setpoint," "Read Actual Temperature," "Determine Temperature Actuation Values," and "Command Temperature

Actuation to Device Drivers." These sub-functions can be again decomposed into lower level functions such as "Read Reaction Assembly Cold Point Temperature." This decomposition takes into consideration the interactions of the thermal functions onboard the spacecraft such as set temperature with other subsystem components such as the reaction wheel assembly and the batteries, which both generate heat. The traditional spacecraft decomposition does not take into consideration these interactions, which leads to high coupling between software components.

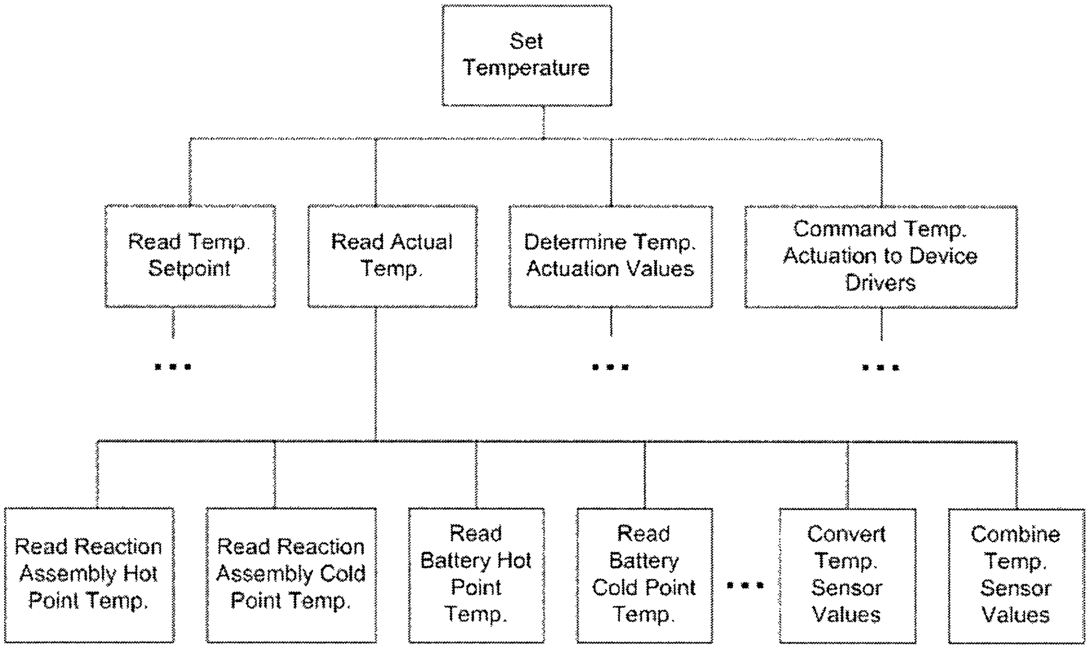

Set Temperature
Read Temp. Setpoint
Read Actual Temp.
Determine Temp. Actuation Values
Command Temp. Actuation to Device Drivers
...
...
...
Read Reaction Assembly Hot Point Temp.
Read Reaction Assembly Cold Point Temp.
Read Battery Hot Point Temp.
Read Battery Cold Point Temp.
...
Convert Temp. Sensor Values
Combine Temp. Sensor Values

Figure 2-15. Example Functional Decomposition

As described in Chapter 1, the techniques currently employed are becoming obsolete, because they (1) do not take a systems view of the spacecraft, (2) do not take into consideration the complex physical environment in which spacecraft operate and (3) cannot rely on the redundancy techniques that provide risk mitigation for hardware components. MDS has been proposed as a solution to the limitations of these traditional approaches.

Although MDS seems like a viable new approach to spacecraft software development, it has not yet been finished nor tested on a real spacecraft, despite the significant amounts of time and money NASA invested in its development. In addition, the Remote Agent project ran into difficulties during its 24-hour control of DS-1. There was a race condition between two threads in the Smart Executive, which lead to deadlock and caused DS-1 to enter safe mode and shut the Remote Agent experiment down [64]. Furthermore, neither of these architectures has been evaluated for their appropriateness in the control system of a safety-critical system, mimicking the lack of safety-critical architecture and pattern evaluation in the software engineering community. So, although State Analysis provides much of the infrastructure needed for effective spacecraft software development, the software architecture in and of itself may not be suitable given current needs and desirable quality attributes. Lessons can be learned however from the MDS experience to formulate a new approach that exploits the advantages provided by MDS as well as the advantages of several other modern software engineering techniques.

2.4 – Summary

This chapter provided the background information and related research necessary for understanding the context in which a new approach to spacecraft software development will be constructed. The incorporation of modern software engineering techniques with current spacecraft software development will help the domain reach greater levels of potential, by bringing the field "up to speed" with advancements made in other domains. Furthermore, the incorporation of the successful aspects of related approaches to revising spacecraft software development will also aid in creating a new and effective approach to writing spacecraft control software. The next chapter describes in detail how modern software engineering techniques and aspects of related approaches can be integrated to form a new approach to spacecraft control software development.

[This page intentionally left blank.]

Chapter 3:

Engineering Spacecraft Control Software Using a Software Product Line Approach

Based on the foundation of background information and related research laid in the previous chapter, this chapter outlines, in greater detail, the approach to engineering spacecraft control software recommended in this dissertation. Creating an approach for future embedded control applications, especially for the unique characteristics of the Exploration Initiative depends highly on the ability to create a seamless paradigm for spacecraft software development that takes into consideration these new and promising trends in the software engineering industry. This approach is based on techniques derived from software product-line research and is supported by a common software project management structure, model-based development, safety-driven design and an incremental, iterative and flexible software process model. Central to the software product-line approach is the development and selection of the software product-line architecture. Software architecting and architecture evaluation is performed through a process called Multi-Attribute Software Architecture Trade Analysis, described in Chapter 4. First, a more detailed description of the example (the Exploration Initiative) used throughout the remainder of this dissertation is provided.

3.1 – The NASA Exploration Initiative

In January of 2004, President Bush outlined the US Vision for Space Exploration. The Exploration Initiative describes the President's objectives for creating a sustainable, human presence in space by first returning to the Moon and then continuing human exploration to Mars [96]. To assist in attaining these objectives, a team comprised of MIT and Draper Laboratories was established to study the three main aspects of the Exploration Initiative: Human Lunar Exploration (HLE), Human Mars Exploration (HME) and the Crew Exploration Vehicle (CEV) [37].

The objectives of the MIT/Draper alliance can be summarized by four guiding principles that were followed while achieving the recommended scientific, economic and security objectives:

1. Design for sustainability (includes affordability);
2. A holistic view of the entire system (including software and safety), with a focus on value delivery;
3. A highly modular and accretive design that allows for extensibility and evolvability; and
4. A system design that uses the Moon as a reference goal to validate the Martian exploration concept, otherwise known as "Mars-back" [37].

The first phase of the system architecting process was comprised of (1) enumerating stakeholders and value, (2) creating mass models of spacecraft subsystems for a variety of vehicle types, (3) evaluating and optimizing possible combinations of vehicles that comprise transportation architectures for both HLE and HME, (4) exploring the surface mission option space, (5) performing a preliminary hazard analysis and (6) beginning the software development process for the software that will support HLE and HME [37].

The result of the work completed during this base period was a preliminary selection of system architectures consisting of concepts for both the transportation and surface components of the Exploration Initiative. The work culminated in the selection of a system architecture that was recommended to NASA as the result of many detailed trade studies [37].

Each HLE and HME system architecture is comprised of two components: (1) the transportation architecture, which defines the vehicles required to travel to various points in space and (2) the surface operations architecture, which defines the infrastructure and facilities necessary for achieving mission objectives for a particular exploration campaign.

Figure 3-1 depicts one example of a Mars transportation architecture out of more than 2000 architectures considered [37]. The horizontal lines depict physical locations in space where various vehicles reside and interact; for example, "E Orbit" and "M Orbit" stand for Earth and Mars orbits respectively. Each colored shape represents a different vehicle in the transportation architecture; for example the small red circle represents a CEV that will take the crew from the Earth's surface to Low-Earth orbit.

In this architecture, a return CEV (CEVb) and ascent stage (AS) are pre-deployed to the Martian surface using an autonomous propulsion stage (TMI4) and autonomous descent stage (DS4). A return habitat (HAB3) and propulsion stage (TEI) are also pre-deployed into Mars orbit. After these assets are in place, the crew is transferred to Mars orbit on the CEVa with a dedicated propulsion stage (TMI1), descent stage (DSc), and a combined transit and surface habitat (HAB1), which combine to form the transit stack. Once in Martian orbit, the DSc and HAB1 separate from the CEVa and TMI1 and descend to the Martian surface.

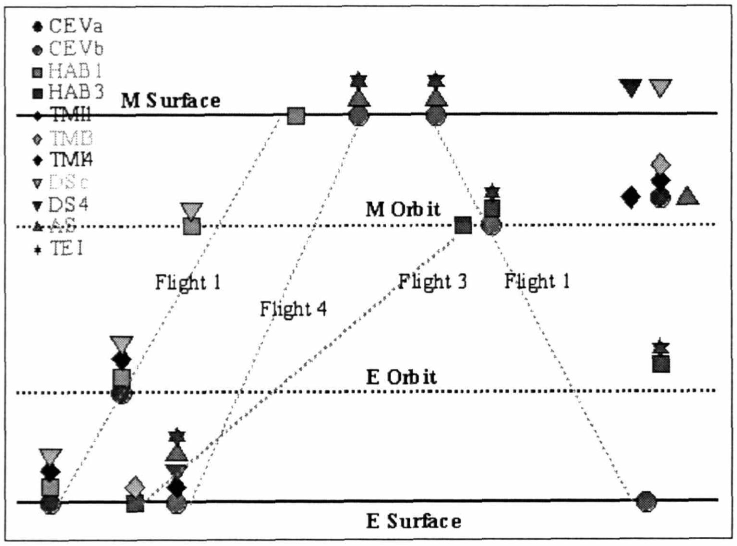
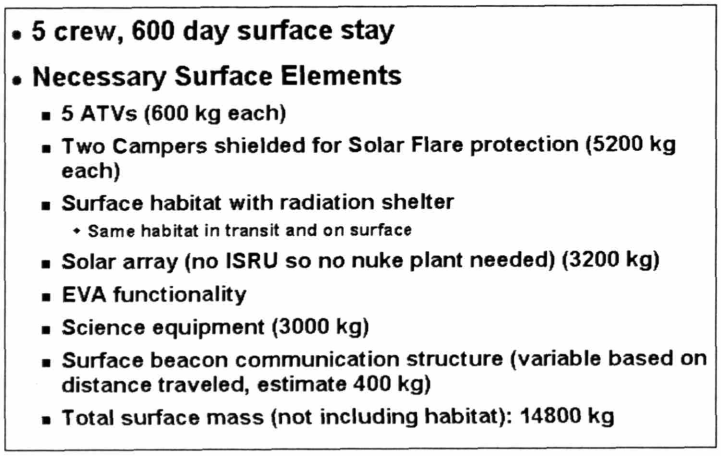


Figure 3-1. Transportation Architecture Example

Upon surface mission completion, the crew moves into the CEVb/AS assembly and ascends into Martian orbit. Once there, the stack rendezvous' and docks with the pre-deployed return propulsion stage/HAB3 assembly and discards the used AS. This stack transits back to Earth. In Earth orbit, the crew moves into the CEVb, separates from the used propulsion stage and habitat, and descends to Earth surface.



- **5 crew, 600 day surface stay**
- **Necessary Surface Elements**
  - **5 ATVs (600 kg each)**
  - **Two Campers shielded for Solar Flare protection (5200 kg each)**
  - **Surface habitat with radiation shelter**
    - **Same habitat in transit and on surface**
  - **Solar array (no ISRU so no nuke plant needed) (3200 kg)**
  - **EVA functionality**
  - **Science equipment (3000 kg)**
  - **Surface beacon communication structure (variable based on distance traveled, estimate 400 kg)**
  - **Total surface mass (not including habitat): 14800 kg**

Figure 3-2. 600-Day Surface Operations Architecture

Figure 3-2 describes the aspects of a 600-day Mars surface operations mission [37]. Surface operation begins with an automated pre-deployment of surface assets, including the nuclear power system, science equipment, five unpressurized all-terrain vehicles (ATVs), two pressurized, unmotorized mobile habitats (Campers), two autonomous rovers (mostly used for scouting before the crew arrives), and any surface beacons necessary to support precision landings.

After equipment is pre-deployed, the crew precision lands with the descent vehicle and transit/surface habitat within walking range of the pre-deployed equipment. After landing, the crew uses the ATVs to carry supplies and equipment back to the habitat landing site, and connect the solar arrays to the transit/surface habitat.

During exploration, crewmembers use the ATVs to explore within the local area around the habitat site and to carry science equipment to sites of interest. To extend their range for overnight stays, crewmembers attach the mobile habitats (campers) to the ATVs. Once at the study site, the crewmembers can disconnect a camper from the ATV, and use the ATVs to explore the area around the study site. Extended exploration is carried out by the two autonomous, unmanned rovers.

During surface exploration, the habitat communicates to Earth directly via communications relay satellites in Mars stationary orbit. When not in a line of site of the main habitat, the ATVs, campers and unmanned rovers communicate with the habitat via the stationary-orbit relay satellite. Surface navigation for the ATVs and unmanned rovers is supported by navigations beacons pre-deployed on the Martian surface.

3.2 – A Software Product Line for the Exploration Initiative

Creating a software product line for spacecraft in general and the Exploration Initiative in particular involves the development of infrastructure in three different areas: (1) technical (software engineering) activities, including core asset development, individual product development and product line scoping; (2) technical management activities, including defining a production plan, lifecycle model and approach to safety; and (3) organizational activities such as developing a management structure. This section addresses the results of performing these activities in each of the three practice areas.

3.2.1 – Technical (Software Engineering) Activities

The software engineering activities involved in establishing the infrastructure necessary for a successful software product line include core asset development, product-line scoping and individual product development.

3.2.1.1 – Core Asset Development

In terms of a software product line approach for the Exploration Initiative, the core assets for the product line may include the following:

Requirements → Requirements (should) typically comprise nearly 60-70% of the software development effort [74]. As described in Section 2.1.2, one of the technologies recommended for creating an effective software engineering enterprise involves the use of model-based development. Executable requirements specifications allow the various contractors creating different vehicles to tailor their models to the variation points required in their projects. Consequently, the models are reused from one product to the next across the line before any design or implementation has begun.

These models can be managed by the NASA SE&I group (see Section 3.2.3) and used by the various contractors to help ensure conformance to the product line guidelines and discovery of requirements specification errors early in the development lifecycle. The ability to perform testing and V&V early in the development lifecycle is especially important when core assets are used in a new context or in an unexpected way. The behavior of the software can be modeled and observed in its new context to help ensure that no unexpected behavior arises that could lead to a hazardous state.

As described in [131], in a traditional spacecraft decomposition, each of the subsystems and components that comprise the subsystem can be modeled in a system engineering development environment that supports both formal and informal requirements specification. These models can be built by the individual contractors creating the various vehicles and facilities for the Exploration Initiative, but they must conform to the guidelines and standards described in Section 3.2.3. Furthermore, these models will be maintained by the SE&I organization and shared with other contractors or project teams that may need the components for their vehicles or facilities. For example, one software development contractor may need to build a GN&C subsystem for the ascent vehicle for the Mars campaign. However, as described in the production plan outlined later in this chapter, the ascent vehicle will be developed much later than other vehicles such as the CEV. Consequently, the generic, executable requirements specification that was developed for the CEV GN&C software can be reused for vehicles like the ascent stage, thereby eliminating much of the development cost that would normally be associated with creating an entire new, proprietary software module for use by the ascent module contractor.

As proposed in [77], safer reuse can be achieved through a model-based approach that focuses on the reuse of requirements specifications, both informal and formal. Reuse takes place before any detailed design is completed, any hardware built, or any software coded. Performing this type of model-based development helps engineers to (1) find specification errors early in product development so that they can be fixed with the lowest cost and impact to the system design, (2) increase the traceability of the requirements and design rationale throughout the project lifecycle and (3) incorporate required system properties into the design from the beginning rather than the end of development, which can be difficult and costly. These benefits help software engineers avoid the unexpected interactions between current and reused code that often results from a lack of requirements review, architecture analysis, and integration testing, as in the SOHO accident [95]. Furthermore, miscommunication between engineers from different fields may lead to severe misunderstandings, which can eventually lead to costly mishaps. Formal models provide a common means of communication through which various engineers of different disciplines can discuss the same aspect of the project [77].

Architecture → As outlined in Section 2.1.1, the software architecture represents the most important artifact in the core asset base, because the architecture governs how each product in the line will be assembled from the other core assets. Consequently, software architecture evaluation during each phase of development is even more essential in a software product line approach to help ensure that the architecture is suitable given the characteristics of each product in the line. Due to the importance of software architecture as a core asset in the product-line approach, the next chapter is dedicated to the details of the software architecture development and selection process.

Software Components → Although several recent aerospace accidents ([45] and [81]) have demonstrated the potential risks involved with reusing code, the use of software components, including both COTS and shared code assets, will be necessary and cost effective to the Exploration Initiative software product line. Although implementation only comprises 10% of typical software development costs, additional savings can be achieved through software code reuse if the software components are built for the product line as a whole from the beginning of the core asset development effort. For example, the software product line architecture may prescribe that the same avionics hardware and operating system be used across multiple products, in this case vehicles, in the line. Consequently, a common operating system could be built and reused from one vehicle to the next.

Furthermore, other common subsystem components, such as Guidance, Navigation and Control (GN&C) modules can be used across a wide range of vehicles that have similar mission objectives. For example, some of the system architectures call for a number of TMI stages for transporting crew vehicles and pre-positioning habitat modules between the Earth and Mars. These stages can make use of the same GN&C system. Also, manned vehicles can all share the same Environmental Control and Life Support System (ECLSS) software.

It is important to note, however, that judicious reuse of software component assets must be employed to help ensure safety and reliability. Mars Climate Orbiter (MCO) provides a clear example of the difficulties with code reuse and an example of reuse leading to an accident. Not only did the MCO development team reuse software from Mars Global Surveyor (MGS), but they also used vendor-supplied equations in their angular momentum desaturation files. Because the MGS software had worked before with the vendor-supplied equations, there was little or no attention paid to the interface between these components. A critical conversion factor from English to Metric units was included in the MGS software, but was never documented; consequently, the conversion factor was left out of the MCO software [45].

Instead of reusing code, the *models* of the desired behavior of the software, in the form of executable requirements specifications can be reused from one product in the line to the next. These components need to be built for all of the products from the beginning of the core asset development. Therefore, the requirements models need to reflect the fact that the component is used between multiple vehicles and the software component must be simple enough that the entire module can be used in all the products without changes or extra functionality. For example, an ECLSS system can be reused in all the flight vehicles, because the functionality will always be the same. Similarly, portions of the GN&C system, such as attitude determination, may be able to be reused between injection stages using the same attitude determination devices. Furthermore, if COTS software code is reused, these components must follow any guidelines set by the SE&I organization and must have a corresponding model for testing before adding it to the core asset base.

Performance Modeling and Analysis → Performance modeling and analysis become especially important for spacecraft control software given the real-time constraints on many of the components. Consequently, the Exploration Initiative must either supply tools for performing this type of modeling and analysis or guidelines as to what characteristics the modeling and analysis tools must have if they are to be used by a contractor working on a vehicle for the Exploration Initiative.

Furthermore, a project-wide safety-analysis needs to be conducted at the NASA SE&I level and also within each of the individual product groups. As described in the previous chapter, a STAMP-based hazard analysis should be conducted throughout the system development process, which includes the software development. Software architecture development and safety-analysis is described in greater detail in Section 3.2.2.1, which discusses the process model used in the software product-line approach.

Business case, market analysis, marketing collateral, cost and schedule estimates → Constructing a business case and analyzing the market for Exploration Initiative software helps engineers to identify the stakeholders in the system as a whole and thereby identify the quality attributes desired by these stakeholders. These quality attributes describe the non-functional requirements for the software and therefore help shape the software product-line architecture. For example, the Exploration Initiative will span multiple decades. Consequently, NASA and its contractors will need to build both extensibility and evolvability into the software architecture to help make the software development for this endeavor possible. In other words, the software product line must be sustainable over the duration of the Exploration Initiative. Sustainability is just one of the quality attributes desired by Exploration Initiative stakeholders. The other stakeholders and the associated desirable quality attributes are described in greater detail in the following chapter as part of the software architecture development and selection approach.

Tools and Processes for software development and for making changes → The tools and processes for software development may be different among the various contractors involved in the enterprise. However, it is essential that the systems engineering and integration team be responsible for providing tools and processes to maintain the core assets, incorporate new core assets, make changes to existing assets, and notify all the contractors of how the assets have been affected and how that in turn affects the projects they are working on. Change control, requirements tracking and interface standards become the backbone of maintaining the core asset base. Interface standards are especially important, because any software components added to the core asset base must follow these standards in order to be successfully and safely used with the architecture.

Test cases, test plans, test data → Using a model-based development approach also allows for the easy and safe reuse of test cases, test plans and test data. Much of the system testing can be completed before any code is written because the analysis of the software can be performed on the model. These tests cases, plans and data can be reused along with the model from one product to the next. Furthermore, traditional testing information can also be added to the core asset base for use across multiple products and contractors to help foster creativity and coverage in testing practices and analyses.

People, skills, training → By using a software product-line approach, people, skills, and training can be utilized across the various products that comprise the Exploration Initiative. Personnel can be more easily transferred among product projects as required because of the shared knowledge of the core asset base; the common project management structure; and the guidelines, standards and processes advocated by that organization.

3.2.1.2 – Product Line Scope

One of the most important aspects of creating a software product line is defining the scope. The software product-line scope outlines the type of products that will comprise the line and, more importantly, what will not be included in the line. Typically, a product-line scope consists of a description of what the products have in common and the ways in which they vary from one another. For example, these variation points might include features or operations each product provides, performance or other quality attributes exhibited by the products or the platforms upon which they run [31]. The variation aspect of product line scoping is discussed in greater detail in the following chapter as part of the discussion on software product-line architecture development and selection.

For a software product line to be successful, its scope must be clearly defined [34]. If the scope is defined too widely, the products will have many variation points and the core assets will be strained beyond their ability to accommodate that variability. Furthermore, if the scope is defined too narrowly, the core assets will not be built in a generic fashion and extensibility and evolvability will be sacrificed. Clearly, the definition of product-line scope is important to the success of this software development approach.

In terms of spacecraft development and the Exploration Initiative in particular, the product-line scope must take into consideration both the variability between the numerous vehicles and the facilities that will be used to achieve mission objectives as well as the variability in destinations of the Initiative, namely low-Earth orbit (LEO) and the International Space Station (ISS), the Moon and Mars. As described in Section 3.1, one of the system architecture options includes the following vehicles and facilities to achieve mission objectives: CEVs, habitat modules, trans-Mars injection stages, descent stages, an ascent stage, a trans-Earth injection stage, ATVs, and Campers. The software product line scope must include the wide-variety of functionality that must be provided for all of these vehicles and facilities to help ensure the sustainability of the software product line as well as the Exploration Initiative as a whole. The differences between the requirements for the various vehicles and facilities are best described through the software product-line architecture and associated variation points described in the next chapter.

3.2.1.3 – Product Development

The product development plan outlines how the core assets are built and then assembled over the course of the software product-line lifetime. Figure 3-3 illustrates the inputs, outputs and activities involved in the product development strategy for the software product line.

The inputs into individual product development include the product-line scope (described in the subsection above), the core assets, a production plan, and the requirements for that particular project. The requirements for a product are often expressed through the variation points from a generic product description contained in the product-line scope or the set of product-line requirements [26]. These inputs from the software engineering technical activities flow into the product development activities and are supported by both technical and organizational management activities. Figure 3-3 illustrates these relationships.

It is important to note that the process of product development is rarely linear; the creation of products often has a strong feedback effect on the product-line scope, the core assets, the production plan and sometimes even the requirements for specific products. Each additional product may have similarities with other products that can be exploited by creating new core assets. Thus, as more products are developed as part of the line, the product line as a whole extends and evolves to accommodate the new requirements and needs of the enterprise.

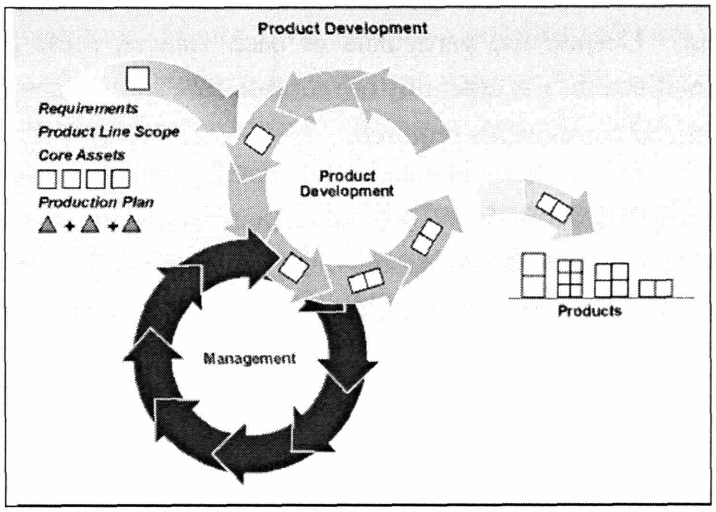


Figure 3-3. Product Development [26]

Product development for the vehicles and facilities in the Exploration Initiative software product line includes both an elaboration of the core assets, the variation points and any additional elements required by a particular product and how these components are assembled to produce a given product. A customized development plan is then created for the particular system. These development plans guide the design, assembly and testing of a given version of a particular product, based on the available core assets, which new core assets require development, the variation points that need to be applied and any product-specific elements required for development. Feedback is also provided back into the core asset base on any core asset design changes, process changes, etc. that came as a result of experience gained for the product development.

The descent stage of the habitat landing stack provides an example of how product development plans need to be customized for the particular products. Many of the standard requirements specification models can be reused from the core asset base including power, thermal and communication subsystems. The first step in developing the descent stage is to tailor these core assets with respect to the specifics of the vehicle and cater to the variation points outlined in the architecture description. Next, additional components not a part of the core asset base need to be developed because of their specificity to the particular vehicle or facility, in this case the descent stage, need to be developed. The descent stage requires an EDL (Entry, Descent and Landing) subsystem for landing on the Moon or Mars. EDL systems are very specific, not only characterized by which planetoid they are landing on, but also the location on that planetoid, what navigational aids are provided, and the type of landing aids used, i.e. parachutes or landing rockets.

As demonstrated in this example, the production plan is tailored to the particular requirements of the vehicle. In the same manner, other production plans can be developed for the remaining vehicles in the system. Despite the particulars of each vehicle, these production plans will always include the guidelines for the assembly of reusable core assets, the variation points for the vehicle and any specialized components required.

3.2.2 – Technical Management Activities

Technical management oversees the core asset development and the product development activities by ensuring that the groups who build core assets and the groups who build products are engaged in the required activities, follow the processes defined for the product line, and collect data sufficient to track progress [26]. This section addresses (1) the software lifecycle model and safety approach and (2) the production plan that the technical management will follow in order to ensure that the various product engineers are adhering to their core asset and product development activities.

3.2.2.1 – Software Lifecycle Model and Approach to Safety

One of the most important aspects of creating an effective software approach, and a software product-line development process in particular, involves clearly identifying and adhering to a software process model. The process model used to support the Exploration Initiative needs to be (1) incremental and iterative to support extensibility, evolvability and inevitably changing requirements, (2) flexible to support sustainability throughout the long lifespan of the Initiative and (3) must integrate both the system and safety engineering processes with the software engineering process, supporting feedback between the various engineering efforts and thereby enhancing understanding and communication between the engineering disciplines. New components and systems need to be easily added to the project and requirements, specifications and constraints easily and safety changed or added. Furthermore, the process model needs to be flexible, allowing for the multiple contractors to use the techniques most appropriate to their domain, whether that be a particular design technique, rapid prototyping strategy, etc.

This software process model must be followed by the SE&I organization throughout the duration of the Exploration Initiative because the SE&I team has ownership over the core assets, which includes the software architecture. Since software architecture is central to the software's ability to achieve quality attributes, this artifact is elevated to a position of importance where developing, selecting, and refining this core asset becomes central to the software process model followed by the SE&I contractor. The individual contractors will undoubtedly have their own, in-house software process models and therefore do not need to follow the enterprise-wide software process model. The enterprise software process model is used to help ensure the successful development and maintenance of core assets.

Several software development processes were described in the previous chapter. Furthermore, system architects have many well-defined processes to follow that were developed for systems engineering and manufacturing. Figure 3-4 illustrates a possible software process model for the Exploration Initiative software product development derived from combining aspects of both software and system development processes. It shows how the software engineering process interacts with the system and safety design techniques to create an integrated system of systems approach to development. It is important to note that not all of the system and safety process steps are listed in this diagram; the diagram focuses on the software process and how it is embedded in the system and safety architecting processes.

The software process model shown in Figure 3-4 focuses on software architecture development and selection because of the impact of software architecture on the ability of software to not only meet the requirements but also quality attributes. Furthermore, the SE&I organization's development and selection of core assets impacts the development of the vehicles that comprise the various missions throughout the enterprise endeavor. Each contractor is dependent upon the core assets, particularly software architecture. Therefore, these artifacts should be the focus of the software development effort. The next chapter describes, in greater detail, software architecting and the information necessary for understanding, describing and evaluating a software architecture.

The software development process begins as any other system architecting process does with a concept development and feasibility stage. During this preliminary development stage, the system and safety engineers perform concept analysis, trade studies and a preliminary hazard analysis. Information from the system engineering concept development and trade studies as well as information from the preliminary hazard analysis feeds into the feasibility study for the software that will be used to support the system design concepts. Furthermore, the results of this feasibility study are fed back into the concept development and trade studies to help the system engineers understand the feasibility of developing the software required to support their various system designs.

After this preliminary feasibility and concept development stage is completed, the system and software engineers can elaborate upon the requirements gathered from the customers to create a more detailed picture of which parts of the system's functionality and non-functional attributes will be addressed with software. Requirements, specifications, quality attributes and trade-off points are recorded during this phase and fed into the software architecture description, selection and refinement activities. As the system and system component designs are changed and/or

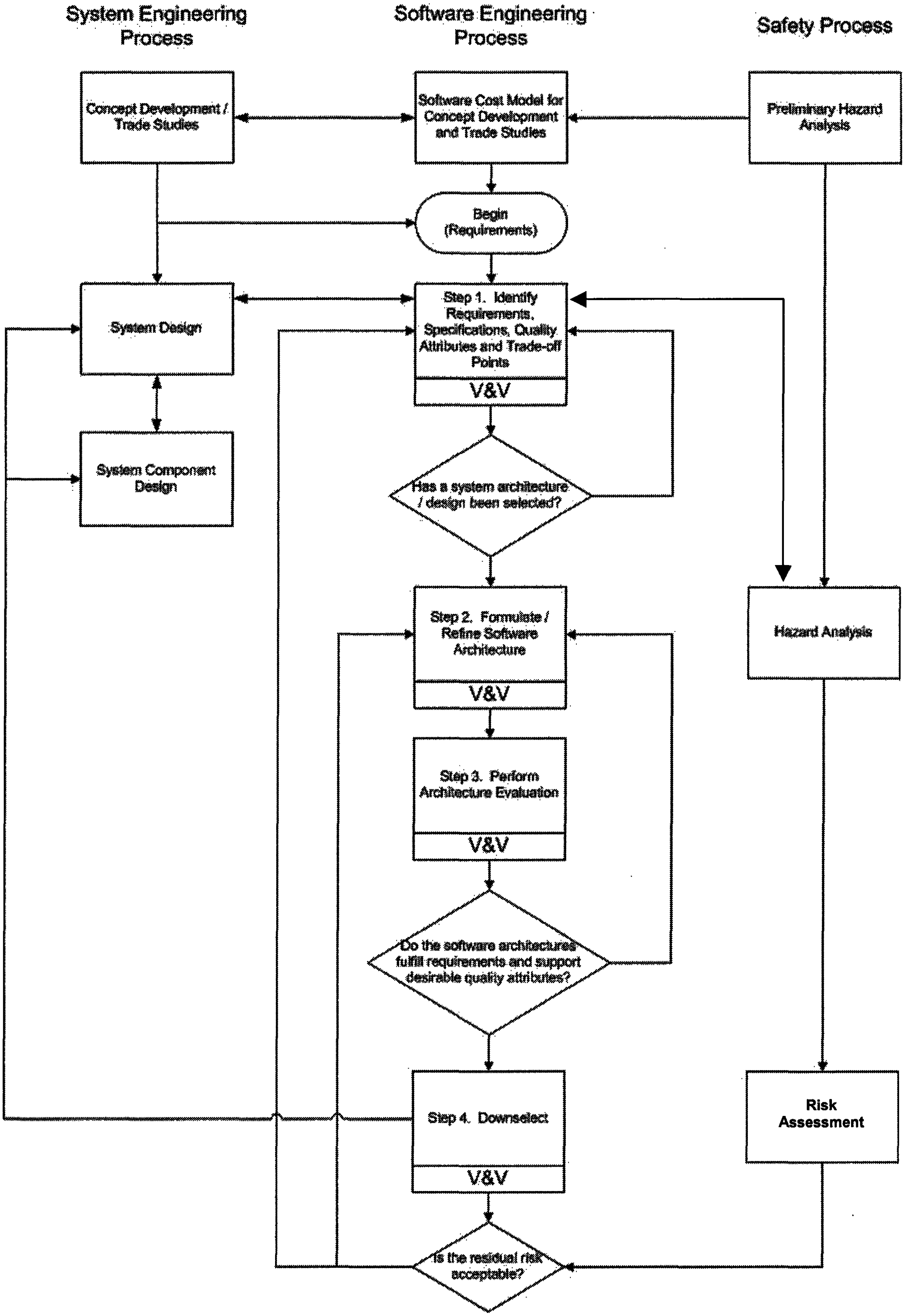

System Engineering Process
Software Engineering Process
Safety Process
Concept Development / Trade Studies
Software Cost Model for Concept Development and Trade Studies
Preliminary Hazard Analysis
Begin (Requirements)
System Design
Step 1. Identify Requirements, Specifications, Quality Attributes and Trade-off Points
V&V
System Component Design
Has a system architecture / design been selected?
Step 2. Formulate / Refine Software Architecture
V&V
Hazard Analysis
Step 3. Perform Architecture Evaluation
V&V
Do the software architectures fulfill requirements and support desirable quality attributes?
Step 4. Downselect
V&V
Risk Assessment
Is the residual risk acceptable?

Figure 3-4. Process Model Diagram

refined, more detailed requirements specifications are fed to the software engineers from the system and safety engineers. As this information becomes available, the software architecture can be further evaluated and refined to cater to any new needs. Once a system architecture or design has been selected, the software engineers can move on to formulating and refining preliminary software architecture options to support the system architecture that was chosen.

It is important to note that verification and validation activities are performed during each stage of the software development lifecycle. Verification determines whether or not the system meets its requirements (are we building the system right) while validation determines if the system meets the user's requirements (are we building the right system). V&V activities are necessary in each phase, to ensure that the results obtained adhere to those required. These activities are especially important in the early phases of development when mistakes are easier and cheaper to fix.

3.2.2.2 – Production Plan

The product development plan provides guidelines to software engineers as to how a product can be assembled using the core assets and provides management with a rough schedule of core asset production and product production. For the Exploration Initiative, product and core asset development will have to be performed in tandem because of budget and schedule constraints.

Appendix A illustrates a portion of an example product-line development schedule, which coordinates the evolution of the product-line core assets and enterprise products. It also illustrates an example of how product-line assets and products combine into a downstream task and force synchronization points and precedence between scheduled activities.

In order to complete the software for the CEV given the time constraints of the Exploration Initiative, two development activities must begin as soon as possible:

1. Development and establishment of the product-line infrastructure, and
2. CEV software development simultaneously to the product-line core assets.

A parallel software development effort is required to deliver a CEV that can travel and dock with the International Space Station (ISS) by the end of 2010. A product-line development approach provides the ability to quickly advance the CEV to support short lunar missions by 2015 along with the ability to accelerate development of the HAB software for use in long-duration lunar missions by 2018.

The approach uses the vehicle subsystems to partition the software development effort. The example schedule assumes that requirements and modeling and the software architecture work for each subsystem will take about two years to complete; that the translation of the models through the architecture will take about one year to complete; and that once the subsystems have been constructed, they will take roughly two years to integrate and test. Given a start date in 2006, this development plan will yield a completed and tested CEV near the end of 2010.

Once the software architecture for the CEV to 2010 is complete, work can commence on the core asset's variation points required to support short-duration lunar exploration with the CEV. Work can also start on newly required subsystems, such as support for utilizing the Deep Space Network (DSN). This new development can occur while the previous version of the CEV is being integrated, tested and operated. The process continues throughout the development of the enterprise vehicle and subsystem software. As this process continues, the approach provides for continual updates to the production plan, as warranted by events and technology improvements, and objective evaluation of the software architecture core asset.

The acceleration of the CEV schedule for a first flight in 2010 presents a significant software development challenge to the Exploration Initiative. The current development schedule calls for the specification, design, construction and test of a flight vehicle in three and half years, assuming the down-selected CEV contract starts by mid-2006. Given a minimum of two years of testing prior to flight, this schedule mandates the delivery of man-rated flight software in three years.

In addition to the development of the core assets in tandem with the CEV software, functionality will also be added into the core asset base over time to help ensure that resources are allocated to the most important requirements. This type of subsystem evolution is illustrated in Figure 3-5.

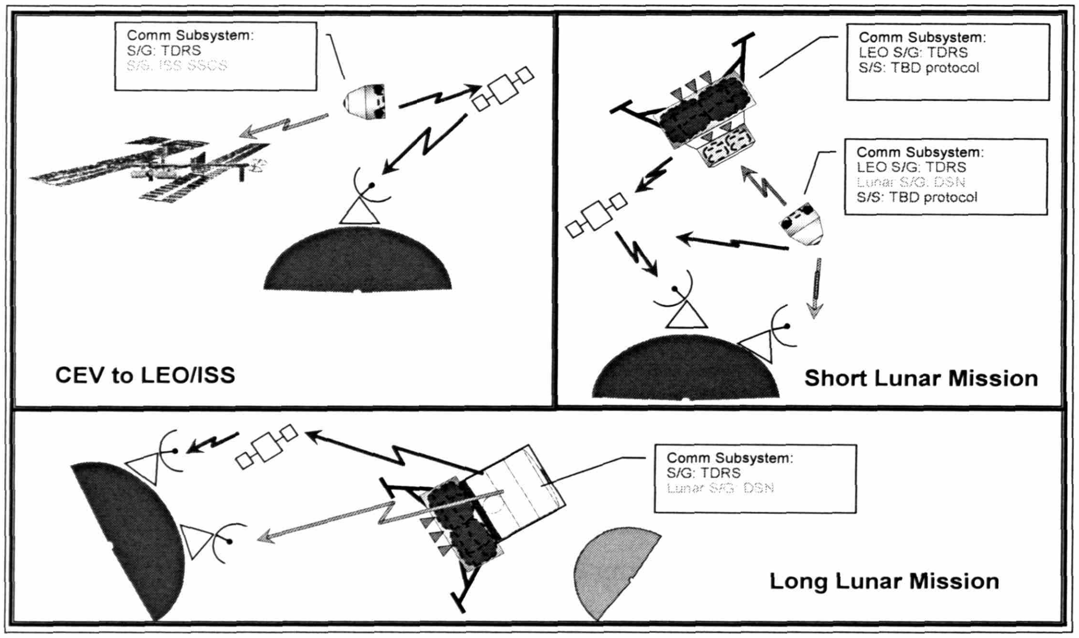


Figure 3-5. Example Subsystem Core Asset Evolution

In the CEV to LEO and ISS configuration, the CEV communication subsystem needs to support tracking and data-relay satellite communications and ISS space-to-space communication service. However, when the CEV software is upgraded to support a short lunar mission, for example, the CEV communication subsystem no longer requires ISS communications, but needs to add Deep Space Network (DSN) capabilities once out of LEO. Furthermore, the CEV communication subsystem software requires space-to-space communications capability to support rendezvous and docking with the ascent and descent stages in LEO and the ascent and descent stages need to support Tracking and Data Relay Satellite System (TDRSS) capability for LEO operations, i.e. to relay status information to the CEV during docking.

These capabilities are not needed in the initial version of the CEV communications subsystem software and therefore resources do not need to be allocated initially to developing this software. The requirements specifications and the architecture, however, need to be constructed with these variation points built into the communication subsystem. Variation points such as these are discussed in greater detail in the following chapter.

The final operation illustrated in Figure 3-5 is the communication subsystem functions required for a long lunar mission. In this configuration, the CEV, ascent stage and descent stage software remain the same as for the short lunar mission. However, the habitat/descent stage assembly requires both TDRSS and DSN capabilities.

There are three development strategies that could be employed for writing the communication subsystem software for the Exploration Initiative: (1) develop TDRSS/ISS/space-to-space/DSN assets up front and migrate the assets to all systems requiring them, (2) develop only TDRSS/ISS capabilities until 2010 for the CEV, being mindful of future use, then develop space-to-space and DSN assets or (3) develop the software for each system separately, i.e. the CEV TDRSS/ISS/DSN/space-to-space system, the TDRSS/DSN/space-to-space for the ascent and descent stages, and the TDRSS/DSN for the habitat assembly. In the first development strategy, a complete communication software subsystem, including TDRSS/ISS/space-to-space/DSN capabilities, are commissioned as a core asset. The asset needs to be completely provided by 2010 for the CEV and then altered to meet any variation points for the ascent/descent stages and the habitat as needed. The difficulty with this approach is that it requires the creation of full communications capability in the form of software even though much of this software is unnecessary to support the short duration 2010 trip to LEO and the ISS.

Furthermore, the third option is equally unappealing because a separate communication subsystem would be developed independently, probably by different contractors, for each of the vehicles and facilities that require such a system, even though much of the required functionality could have been reused. Block upgrades of the communication software can be completed as part of the CEV extension for short and long lunar missions, and separate development efforts for the ascent/descent and habitat contracts. Again, this development approach, not based on a product-line strategy with a core asset base, will be costly, especially considering the amount of possible reuse.

The final development strategy employs an approach of slow accretion of modular assets. In this software product-line approach, the core asset functionality associated with the communication subsystem is built as needed, keeping in mind the need to both extend and evolve the communication subsystem abilities as the mission campaigns become more complex. In this production plan, the following steps comprise the phased approach to communication subsystem software development:

1. Commission only TDRS/ISS portion of the core asset for the CEV 2010 flight to LEO and the ISS.
2. Commission completion of remaining DSN/space-to-space capabilities for the core asset as part of the short-lunar CEV block upgrade.
3. Commission application of variation points to the communications asset for ascent/descent and habitat.

In summary, using a phased production plan in which the software product-line core assets are built with both extensibility (easily allowing the addition of new features at a later date) and evolvability (allowing the system to achieve ultimate purpose gradually) in mind helps decrease the upfront costs of creating reusable components. Functionality can be slowly added as mission objectives change, thereby fulfilling requirements in such a way as to minimize expenditures as functionality is accreted.

3.2.3 – Organizational Management Activities

The activities of the management organization encompass ensuring that a proper structure is in place to support the software product line and that it makes sense for the enterprise. Organizational management must also allocate resources, maintain and monitor communication channels and help ensure that the technical managers and the contractors are following the guidelines and standards they set. Because of the importance of both organizational and technical management to the success of the Exploration Initiative software product line, a model of the enterprise organization structure is needed to provide clear to which the organization must adhere. This section describes a common software project management structure that supports the product-line approach. Through the use of a system engineering and integration (SE&I) organization (NASA will serve this role), guidelines and standards, the core asset base, and new technology exploration will be managed for the entire enterprise. This approach is especially essential for the Exploration Initiative because of the numerous contractors that will inevitably participate in the Exploration Initiative; a common software project management structure is necessary to help prevent and/or mitigate any issues that may arise due to propriety concerns of the various organizations.

3.2.3.1 – Common System Engineering and Integration Organization Structure

All the software for the vehicles and facilities necessary to support the various campaigns will not be developed by one firm; there is just far too much software and infrastructure necessary to support all the functionality for going to both the Moon and Mars. Consequently, multiple contractors will be responsible for different pieces of the system architecture. As described previously, however, the software product line is at the heart of the approach proposed in this dissertation for developing the software for the Exploration Initiative. Therefore, there must be one contractor or organization that is responsible for maintaining the software product line as a whole, which mostly entails the maintenance of the core asset base.

Inevitably, problems will arise due to the proprietary issues associated with multiple contractors working on the software product-line core assets; the various contractors will want to keep portions of their code proprietary and not share the details with the other contractors. As seen in several recent aerospace accidents, software reuse without the explicit documentation (whether it be executable requirements specifications, pseudo-code, or other forms of documentation) of what the software does and how the software works can lead to major mishaps [45], [81]. For example, operational software should not contain unused executable code. However, COTS software in safety-critical systems often contains code that was written for reuse or in a general manner that contains features not necessary for every system or tailored to the specifics of that system. As stated in [74], "the extra functionality and code may lead to hazards and may make software hazard analysis more difficult and perhaps impractical." As seen in the Mars Polar Lander accident, the unused executable code that shut down decent engines was running when it should not have been. Consequently, a spurious signal from the lowering of the landing legs triggered the shutdown of the engines, and the Lander plummeted to the Martian surface [45].

One way of mitigating the problems that arise due to proprietary issues is to create a common systems engineering and integration organization structure that supports all software development across the enterprise, manages the core assets, and ensures that each software contractor conforms to the software product-line guidelines and standards. Figure 3-6 illustrates an example software project management structure that can be used to support the software product-line development approach for the Exploration Initiative. As seen in Figure 3-6, a system engineering and integration team directs the asset development and sustainment as well as retaining ownership of the software product-line core assets. Furthermore, this organization is responsible for setting guidelines and standards for the tools and processes used by the various contractors. There will undoubtedly be a number of contractors working on different aspects of the Exploration Initiative system architecture. These organizations will report directly to the system engineer, who works for the system engineering and integration organization, but operates out of the contractor site.

At the contractor level, the system engineers are responsible for (1) scoping the requirements and business case for their particular products, (2) making sure that these requirements and the company's approaches mesh with the guidelines of the system engineering and integration organization, and (3) interfacing directly with the customer to ensure that the requirements fulfill the stakeholder needs. Finally, the system engineer must enforce guidelines, constraints, and best practices imposed by the system engineering and integration organization on the project team and project team leaders responsible for engineering the various products created by that contractor. The project teams themselves are responsible for product development and sustainment.

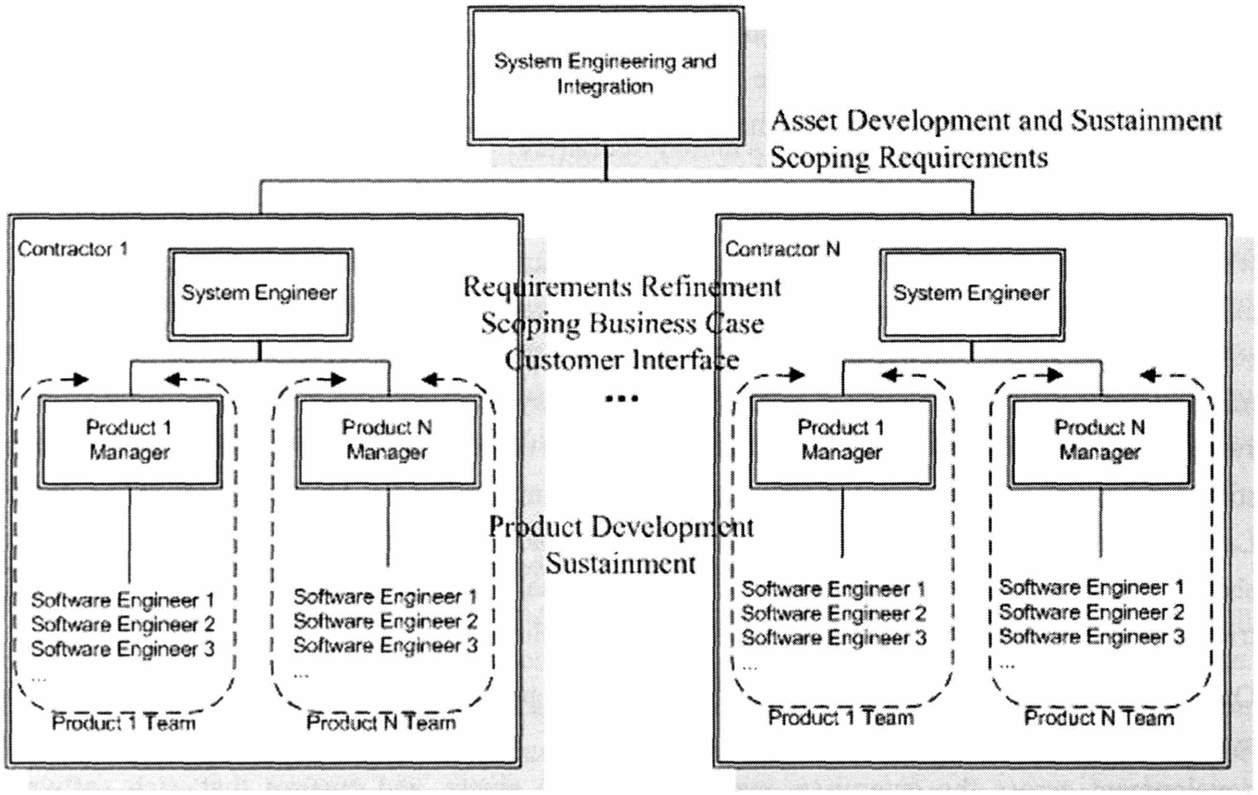


Figure 3-6. Common System Engineering and Integration Structure

There are three areas that are critical to the success of the software product-line approach for which the SE&I organization must provide: change control, requirements tracking, and interfaces. First, a centralized, pre-specified and organized change control policy and tools need to be established at the project management level in order to ensure that the changes are accurately and effectively recorded and then communicated to the various product teams. These standards will aid in controlling the impact of changes to the software (critical to ensuring software safety [74]) and supporting sustainability. The individual contractors can have their own change-control systems within their organizations as long as the interface to the SE&I organization is well defined. There exists a need to manage change locally prior to integration points. However, since SE&I is mainly involved in integration, forcing every local change at the lower-level development stages may impose a burden that inhibits local development efforts by adding too much overhead. Consequently, this type of standard change-control policy is kept at the SE&I level and aids the common project management structure in the organization and maintenance of the core asset base throughout the lifespan of the Exploration Initiative. In addition, the variety of contractors that will participate in vehicle development throughout the lifespan of the Exploration Initiative must be educated in the particulars of the standard SE&I change-control policy. The management of the individual projects will be able to easily interface their change-control policy with that of the SE&I organization and communicate any changes to the core assets quickly and effectively. Finally, feedback is delivered to the SE&I team from the various contractors through the system engineers and project managers of each project addressed by a particular contractor. In particular, decisions that may impact the core assets are discussed with the SE&I team before changes are implemented. Discussions about changes to the core assets and feedback from various stakeholders are considered before the core assets are altered.

Furthermore, an enterprise-wide standard requirements-tracking policy and system are also essential for supporting the software product line approach. Given the amount of reuse prescribed by software product lines and the limited amount of funds available to NASA and its contractors for the Exploration Initiative, tracking requirements and intent throughout the core asset base becomes even more critical. These policies will help ensure that the core assets are used properly in any new context in which they are employed. It is necessary to have an enterprise-wide policy and system for core asset maintenance because if the requirements are tracked only locally within a given project element, it becomes extremely difficult, and in some cases hazardous, for the other enterprise elements to take advantage of that asset or suggest slight changes to improve the ability of the system.

Finally, interface standards are necessary to (1) increase the reusability of a particular core asset across multiple products, (2) support and enhance the interoperability of a particular core asset with other assets in the base and specialized components created for particular products, (3) increase enterprise knowledge and understanding through a much higher degree of uniformity than if the standards were set within the contractors themselves, and (4) provide higher confidence in the effectiveness and safety of element communication. In addition, individual development firms will have less code to write because their components will not have to translate or adjust their interfaces from one format to another; all components developed within the core asset base of the Exploration Initiative will follow the same standards.

This approach does, however, raise questions about the use of COTS (Commercial, Off-the-Shelf Software). In a software product-line development approach, COTS components form an integral part of the core asset base. However, basing an interface standard on a particular COTS solution or COTS provider can lead to serious problems, especially for an endeavor with a long duration such as the Exploration Initiative. Choosing a single-source COTS solution as the basis of your interface standard forces unnecessary restrictions on the rest of the assets when the company whose standards are used goes out of business or no longer supports that particular component. This type of problem arises often in the use of COTS software and, in fact, was one of the reasons for the failure of the NASA Checkout and Launch System (CLCS) project failure. Due to the safety-criticality of the Exploration Initiative software as well as the wide-variety of contractors that will participate in the core asset base development, COTS software can be used, but their interfaces must comply with the standards designated by the SE&I organization and follow strict documentation guidelines as previously discussed [126].

3.3 – Summary

This chapter addressed the details of developing a software product line for the Exploration Initiative, which forms the basis of the approach described in this dissertation. Technical (software engineering) activities, as well as both technical and organizational management activities were described. These three practice areas provide the foundation for creating an effective software product-line development approach for the Exploration Initiative. The next chapter focuses on one portion of these practice areas, namely the software architecture core asset development, which is part of the technical branch of the software product line. It outlines a new technique for software architecture development, evaluation and selection called Multi-Attribute Software Architecture Trade Analysis, which complements the software process model described in this chapter by outlining the software architecting steps completed during each phase of the software lifecycle.

[This page intentionally left blank.]

Chapter 4:

Multi-Attribute Software Architecture Trade Analysis – Costing and Requirements

Software architecture evaluation supports the product-line approach through fostering scientific development and evaluation of the core asset most important to the success of the product line – the product-line architecture. The essence of building a successful software product line is discriminating between what is expected to remain constant across all family members and what is expected to vary. Software architecture is ready-made for handling this duality, since all architectures are abstractions that admit a plurality of instances. Creating a product-line architecture is achieved through three steps: (1) identifying variation points, (2) supporting variation points, and (3) evaluating the architecture for product-line suitability [31].

Given the number of systems that will rely on the architecture, the third step, evaluation, takes on an even more important role in a software product-line approach. The evaluation will have to focus on the variation points to make sure they are appropriate, that they offer sufficient flexibility to cover the product line's intended scope, that they allow products to be built quickly, and that they do not impose unacceptable runtime performance costs. These evaluations should take place at each stage of the software lifecycle, as the architecture matures and becomes more detailed.

The focus of the next two chapters involves the software architecture development and evaluation portions of the techniques outlined in the previous chapter for creating spacecraft control software for the next generation of spacecraft and, in particular, the Exploration Initiative. Architecture evaluation is used throughout each phase of the software development lifecycle to help ensure that the software achieves the requirements and quality attributes desired by the stakeholders in the project. This chapter outlines the typical phases of the software development lifecycle and describes the architecture development and selection practices employed during the first two development phases, namely concept generation and feasibility study and requirements, which make up the Multi-Attribute Software Architecture Trade Analysis (MASATA) methodology. The following chapter contains the development and selection practices employed during the design phase of the software development lifecycle.

4.1 – Software Lifecycle Phases

There are many process models that describe the phases of the software engineering lifecycle, the artifacts produced during each stage, and the conditions for transitioning to the following stage. In the previous chapter, an incremental, iterative and flexible software process model was described that lays the foundation for elevating architecture as the most important artifact in the lifecycle. This model is just one of many that are used throughout the software engineering industry including the waterfall model, spiral model [15], and iterative development. Despite the difference in opinion about how to proceed through the various software engineering lifecycle phases, the phases themselves do not vary much beyond a common set. These phases include:

- Concept Generation and Feasibility Study
- Requirements
- Design
- Implementation
- Testing
- Maintenance

These phases can be further decomposed. For example, the requirements phase often consists of high-level requirements capture between the customer and the contractor, requirements analysis, and requirements specification. The latter is a lower-level and sometimes formal description of the functional and non-functional requirements for the system. In addition, design can be broken up into preliminary and detailed design phases, representing two levels of detail or abstraction, in the description of the software organization.

The focus of this dissertation and research, and the objective of Multi-Attribute Software Architecture Trade Analysis (MASATA), is the development portions of the software engineering lifecycle, namely, concept generation and feasibility study, requirements and design. MASATA is a new software development process that focuses on architecture as the primary artifact in development. This approach to software engineering can be used across multiple disciplines; in this dissertation, MASATA is demonstrated on spacecraft software engineering for the Exploration Initiative. The following two sections describe the architecture development and evaluation steps completed during the concept generation and feasibility study and requirements.

4.2 – Concept Generation and Feasibility Study

As illustrated in Figure 4-1, during the concept generation and feasibility study phase of the typical systems engineering development process, engineers identify a need based on a want or desire for something arising from a real or perceived deficiency. At this stage, particular solutions to the identified problem should not be specified; the purpose here is to establish that there is a need, not to make quick judgments about the appropriate solution to a problem. Identification of need is followed by a feasibility analysis, which is comprised of technology-oriented studies to evaluate approaches that could be applied in system development. The objective is to propose feasible technology applications and to integrate these into the system requirements definition process [12]. In short, the concept generation and feasibility study for any project should include the following steps:

1. Define the problem
2. Identify feasible alternatives
3. Select the evaluation criteria
4. Apply modeling techniques
5. Generate input data
6. Evaluate the model

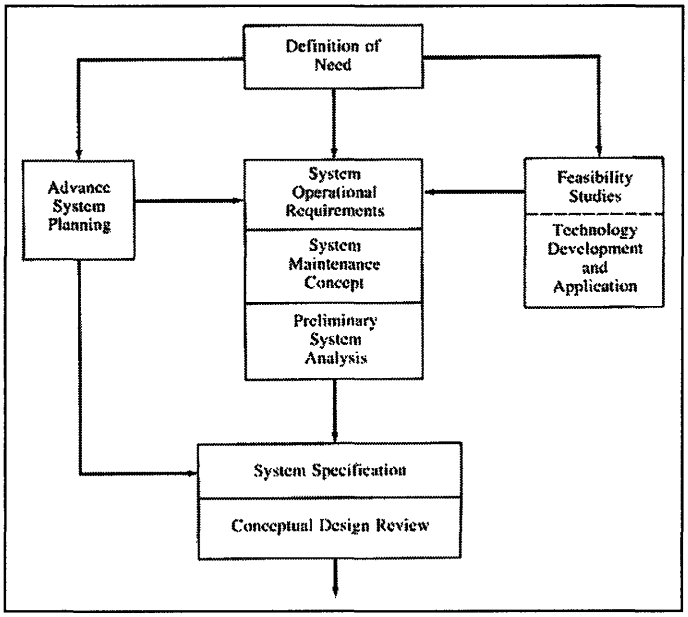

Definition of Need
Advance System Planning
System Operational Requirements
System Maintenance Concept
Preliminary System Analysis
Feasibility Studies
Technology Development and Application
System Specification
Conceptual Design Review

Figure 4-1. System Requirements Definition Process (Example) [12]

In terms of the Exploration Initiative, a need was identified to create a "Safe, sustained, affordable human and robotic exploration of the Moon, Mars and beyond..." [96]. The system engineers proceeded to generate plausible alternatives for the type of vehicles and operations that would be necessary to achieve the goals of returning to the Moon and traveling to Mars. Mass was considered the most prominent evaluation criteria, as placing mass in space is the most costly portion of the project. Models were created for the various vehicles and vehicle configurations, data was entered, and the models were tested. Several system architectures were then selected for the next phase of the study – the more detailed requirements specification phase.

From a software engineering perspective, it is important during this stage of development to identify the stakeholders with respect to the software system and the quality attributes that they desire from that software system. This information is crucial to later steps in MASATA and helps to guide the software engineers when constructing their software solutions. Furthermore, writing software to fulfill a particular function onboard any vehicle or facility is one of many possible solutions to a particular problem. Since the goal of concept generation and feasibility studies does not include identifying specific solutions to problems, it becomes more important at this point to analyze the impact of the different system architectures (and what functionality is allocated to a software solution) on the cost and risk associated with developing a software system to support that architecture. In short, the concept generation and feasibility study steps taken in the MASATA process include:

1. Enumerate stakeholders and define desirable quality attributes for use in later architecture evaluation steps
2. Conduct a software cost and/or risk assessment for system trade studies

4.2.1 – Step 1: Identification of Stakeholders and Quality Attributes

The stakeholders for the software system will vary somewhat from the stakeholders of the system as a whole. The stakeholders identified for the Exploration Initiative as a whole include explorers, scientific, commercial, executive and congressional government, the US population, international partners, the media, national security and educators [37]. From a software perspective, however, it is not only important to create the system with these stakeholders in mind, but also the stakeholders in the software itself. These include the software engineers (designers, programmers and testers), maintenance engineers, operators, and avionics engineers.

Based on the goals of the Exploration Initiative as a whole and the stakeholders identified above, the following quality attributes were identified that deliver value to the software stakeholders:

- Analyzability with respect to safety
- Ease of verification and validation
- Sustainability
- Affordability
- Buildability

- Ability to meet real-time constraints
- "Monitor"-ability

Table 4-1 provides a preliminary, qualitative definition of the quality attributes desired by the software stakeholders.

Analyzability with respect to Safety	The characteristic of the software architecture to support engineers in the construction of software that is analyzable with respect to safety.
Ease of Verification and Validation	The characteristic of the software architecture to support engineers in the construction of software that allows both ease of verification and validation at every phase of the software lifecycle.
Sustainability	The ability of a software system to support meeting stakeholder needs in the present, while preserving the ability of the system to support these needs indefinitely as well as any new needs (both planned and unplanned) that may arise. Encompasses both extensibility (easily allows the addition of new features at a later date) and evolvability (allows system to achieve ultimate purpose gradually).
Affordability	The ability of the software development enterprise to construct software such that the amount of resources required to build the software does not exceed the available resources.
Buildability	The ability of software engineers to construct software from the software architecture. Building the system is intellectually manageable given the available resources.
Ability to meet Real-Time Constraints	The ability of the software engineers to ensure that deadlines will be met and the software will operate within the available resources.
"Monitor"-ability	The characteristic of the software that allows operators to gain insight into the software's operations during runtime. The operators must be able to monitor the performance and telemetry of the various subsystems and devices on board the spacecraft.

Table 4-1. Quality Attributes and Definitions

These seven quality attributes were derived from analyzing the customer needs, key business drivers, stakeholders and a target market assessment as outlined by the enterprise architecting team of the CE&R effort. For example, one of the customer needs outline by the team was "the development of highly modular and accretive designs, including both extensibility and evolvability" [37]. This need led to the identification of sustainability as a desirable quality attribute of the software architecture. The other quality attributes were derived in a similar manner.

4.2.2 – Step 2: Impact of System Architecture on Cost and Risk of Software

After identifying the software system stakeholders, the impact of the system architectures on the cost and risk of developing software to support each system architecture is determined. Performing this type of analysis helps the system engineers determine whether or not particular system architectures are feasible given what is required of the software to support them. Furthermore, the software engineers begin to understand which portions of the software system will be the most difficult, time-consuming and expensive to implement. This information provides the software engineers with a foundation for beginning the software architecting process during the requirements specification lifecycle phases.

There are a variety of techniques that can be used to estimate the cost of software development. These software costing techniques rely heavily on expert opinion, previous company experience, historical data, estimated lines of code or function points, or a combination of these various techniques. The most commonly used software cost estimation technique is Algorithmic Cost Modeling, in which developers build a model of the proposed software project by analyzing the costs and attributes of already completed projects. Many algorithmic cost models have been proposed, the most well known of which is COCOMO II. COCOMO II is a newly revised version of the original Constructive Cost Model (COCOMO) [13]. COCOMO II uses the estimated size of a project (primarily in terms of estimated lines of code) and type of project (organic, semi-detached or embedded). More advanced versions add a series of multipliers for other factors such as product attributes (reliability, database size, complexity), computer attributes (timing and storage constraints, volatility), personnel attributes (experience, capability), and subsystems, and allow consideration of systems made up of non-homogeneous project attributes (use of tools, development schedule).

For large, unprecedented systems, these types of techniques break down, because of a lack of historical data and experience and the unreliability of lines of code or function point estimates. Furthermore, the information required to generate the parameters required for these models is usually not available at the time system engineers are selecting an overall system architecture and need the cost and programmatic risk information. For some systems, particularly those in the aerospace industry, this problem is compounded because similar systems have never been built before and therefore historical information is lacking.

A new cost and risk methodology was developed that is applicable to early concept formation stages. In this costing methodology, relative rankings of development cost, development risk and mission risk are assigned to the software and HCI (Human-Computer Interaction) functional components of each system architecture using several cost- and risk-related parameters. Values are assigned to the parameters with respect to software functions beyond a common, baseline set and summed and normalized to provide relative rankings between the various system architectures. The following subsections describe the evaluation criteria used for both the software and HCI evaluations. Appendix D, Tables 1 and 2 contain a full listing of the criteria and the values they can take in questionnaire form. Two examples of the evaluation criteria as applied to functions in the CEV are used to assist in understanding the software and HCI parameter definitions.

First, in order to evaluate system architectures from a software point of view, it is necessary to establish the basis of comparison between the architectures. Because spacecraft are traditionally decomposed into subsystems and their associated functions [131], a set of functions derived from a functional decomposition was used as the basis of comparing multiple system architectures.

Creating spacecraft control software is not a new task; there are certain subsystems and functions that are implemented by software on almost all spacecraft. However, much of the software to support the new and unique functions (such as autonomous rendezvous and docking) and human-computer interactions such as crew override of autonomous functions have not been previously implemented, particularly on manned spacecraft, and are therefore outside the baseline set of functions that comprise most spacecraft software systems.

Table 4-2 lists a small portion of functions carried out by the various subsystems on a typical spacecraft. This particular example includes the Operations Management Subsystem functions. Other subsystems include Vehicle Control, which is comprised of GN&C (Guidance, Navigation and Control), Thermal Control, Electric Power, Environmental Control and Life Support, Fluids Management; Communication and Tracking; Systems Management; Information Management; Science and Payload Processing; and Crew Display and Control. Appendix B contains a full list of subsystems for both flight vehicles and ground facilities and their associated functions.

...	GN&C	Operations Management Subsystem	Thermal Control	...
	• ...	• Rendezvous and Docking Control	• ...	
	• ...	• Proximity Operations Mon. & Ctrl.	• ...	
	• ...	• Area Traffic Mgt. & Ctrl.	• ...	
	• ...	• Flight Planning & Ctrl.	• ...	
	• ...	• Caution & Warning Generation	• ...	
	• ...	• ...	• ...	

Table 4-2. Example Function List for Flight Vehicles Operations Management Subsystem

These software functions can be separated into two groups: functions that are performed by most typical spacecraft (which are referred to as the baseline set) and functions that are unique or are implemented differently from one architecture to the next (which are referred to as functions beyond the baseline set). For example, as seen in Table 4-2, almost all current spacecraft have software that controls and carries out the flight plan. However, few spacecraft are required to rendezvous and dock with another spacecraft. Therefore, Flight Planning and Control are part of the baseline set of functions while Rendezvous and Docking Control are beyond the baseline set.

The functions used to describe typical spacecraft operations were adapted from [136]. Each of these functions is carried out by a vehicle, a facility, a combination of vehicles (otherwise known as a stack) or a crewmember. These functions are further separated by the level of automation required, i.e. whether the function is carried out by the software or whether the software supports a human in accomplishing the task. This distinction is important because taking the human out of the loop and implementing the function in software does not necessarily result in reduced risk and cost over the system lifecycle [74]. Furthermore, replacing humans with automation will most likely increase mission risk, because human decision-making and intuition are removed from the process.

The following subsections contain two sets of results that illustrate how feedback was given to the system engineers concerning their architectures and the impact of those architectures on software development. First, the software and HCI assessment parameters are demonstrated by evaluating these parameters with respect to two example functions for the CEV from the system architecture described in the previous chapter. Second, the results of performing a full analysis of all functions beyond the baseline set for fourteen of the plausible system architectures are also presented.

The two example functions beyond the baseline set used to illustrate how the cost model parameters are evaluated are: the Crew Exploration Vehicle "Rendezvous and Docking Control" and "Crew Override and Control of Guidance Navigation and Control Functions." In the system architecture described in Chapter 3 (OPN 969), the CEV must autonomously rendezvous and dock with other vehicles and system components. Because the CEV is manned, crew backup of the rendezvous and docking control will also need to be implemented. These two functions are evaluated separately, because they represent two different approaches to implementation and therefore their associated cost and risk scores are mutually exclusive. Autonomous rendezvous and docking control does not have a human component and therefore only the cost parameters pertaining to software are evaluated. On the other hand, "Crew Override and Control of Guidance, Navigation and Control Functions" requires human-computer interaction as well as software to support the human tasks. Consequently, both sets of parameters are evaluated for the latter function. A full listing of the evaluated parameters and their associated numeric values are shown for each of the two functions in Table 3, Appendix D.

4.2.2.1 – Software Cost and Risk Assessment Parameters

In the new cost and risk model, the parameters are evaluated with respect to the software aspects of each function. Not every parameter has an associated evaluation for both functions. For example, "Rendezvous and Docking Control" is more software-centric, while "Crew Override and Control of Guidance Navigation and Control Functions" is more human-centric.

Consequently, many of the human-computer interaction parameters were not evaluated for the "Rendezvous and Docking Control" function.

The software cost and risk assessment parameters include:

- Type of Timing Constraints Required (Hard vs. Soft)
- Use of a Human Override or Backup Function
- Precedentedness of the Software
- Requirements for Deterministic vs. Non-Deterministic Software
- Existence of Multiple Algorithms that can be used for a Backup System
- Need for Distributed vs. Centralized Processing
- Required Computing Hardware Resources
- Number of Software Functions
- Amount of Software Decision-Making
- Number of Required Interfaces
- Average Breadth of the Interfaces

Type of Timing Constraints Required (Hard vs. Soft) → [Hard, Soft, None]

Writing real-time control software for spacecraft, especially spacecraft that must perform autonomously, is costly and risky. Whether a function has hard or soft real time constraints will impact not only the development cost and risk of a particular piece of software, but also mission risk. As deadlines become more stringent and functions more complex, cost and risk increase drastically.

Autonomous rendezvous and docking control requires hard real time constraints. On approach to the target vehicle, the approach vehicle must adjust its attitude, velocity and approach vector using its GN&C Subsystem at correct timing intervals. If these deadlines are missed, the spacecraft may collide with one another or miss each other completely, endangering mission completion. In this case, the function is also safety-critical because the CEV is manned.

Use of a Human Override or Backup Function → [Yes, No]

Incorporating a human override or back-up function separate from the default or primary method of control decreases mission risk. However, the infrastructure needed to train and support the human controller and/or the resources required to develop the back-up function may increase development cost and risk.

In the example provided, human override and back up of rendezvous and docking is allowed. This decision provides the CEV controllers with the ability to take control of the spacecraft if they determine such action is necessary.

Precedentedness of the Software → [TRL Level 9 – 1]

Has software like this been written or used before? NASA uses the TRL scale, or Technology Readiness Level scale, to determine the readiness level of a particular piece of technology [83]. At the lowest level, TRL 1, only basic principles have been observed and reported, while at TRL 9, an actual system has been flight "proven" through successful mission operations. The lower the TRL level, the higher the cost and risk of developing the technology to flight status and the greater the mission risk associated with utilizing that technology.

Autonomous rendezvous and docking of two manned spacecraft is TRL 3 - analytical and experimental critical function and/or characteristic proof-of-concept. This TRL is quite low and development of such a technology requires immediate investment and significant resources, especially if the CEV unmanned test flight is to occur on schedule. In contrast, "Crew Override and Control" of rendezvous and docking is TRL 9 – this function has been used on the Space Shuttle and the International Space Station.

Requirements for Deterministic vs. Non-Deterministic Software → [Yes, No]

Non-deterministic software makes it difficult and very costly to validate that the software will provide the required behavior at a particular time or in a particular situation [74]. There are certain provable properties of deterministic software that decrease the cost and risk associated with human ratings for safety.

Current rendezvous and docking algorithms are deterministic; based on set initial conditions, two vehicles execute a series of maneuvers to bring them into proximity with one another according to a pre-specified approach path. Because these algorithms are implemented deterministically, no additional mission risk associated with the uncertainty of software behavior is added to the total mission risk.

Existence of Multiple Algorithms that can be used for a Backup System → [Yes, No]

If multiple algorithms can be used to accomplish the same function, these can be used to create backup software, which can decrease mission risk. At the same time, this may increase development cost and risk because of the extra resources needed to create backups.

For autonomous rendezvous and docking, multiple algorithms exist that can be used to provide a back up. For example, two different approaches to rendezvous and docking are the R-bar and V-bar algorithms. In the R-bar approach, the spacecraft with the rendezvous and docking control software approaches a spacecraft in orbit around a planet along the radius (hence the name R-bar) vector. The maneuvering spacecraft reaches the higher orbit at the desired time to mate the two spacecraft. In the V-bar approach, the maneuvering spacecraft approaches the second spacecraft along the velocity vector, moving ahead of the second spacecraft in its new orbit. The second spacecraft then "runs into" the first spacecraft and they are mated.

These rendezvous and docking algorithms take very different approaches to connecting two spacecraft in space. Therefore, the rendezvous and docking software can use one algorithm for its primary controller and the other for a back up. The rendezvous and docking algorithms that optimize fuel consumption and docking time will most likely be used as the primary software controller; the back-up controller would probably consist of the "bare minimum" needed to rendezvous and dock two spacecraft.

Need for Distributed vs. Centralized Processing → [Yes, No]

Operations requiring the distribution of their implementation across different computers will increase development cost and risk by requiring additional software to provide the communication between those computers and additional testing complexity to cope with any required synchronization between those computers. Mission risk is also increased due the potential failure of the communication path between the computers, the difficulty of fully testing synchronization between the computers, and the added software required by the hardware configuration. For the functions analyzed in this example, the software runs on one processor, that of the spacecraft being controlled.

Required Computing Hardware Resources → [High, Medium, Low]

Depending on the implementation technique employed, varying levels of computing resources (CPU cycles, volatile and non-volatile storage requirements) are necessary. Increasing resource demands, especially on board spacecraft whose hardware must be radiation hardened before use will increase development cost. Furthermore, algorithms that push the limits of hardware specifications also increase development cost and risk, because they must be optimized to fit within these constraints.

For autonomous rendezvous docking and control, resource utilization is high, because autonomy achieved through software is computationally intensive and requires large amounts of RAM to perform the necessary calculations. On the other hand, providing user-interface software for a crew-controlled, backup GN&C system uses significantly less resources. In this case, CPU cycles and volatile memory requirements are medium, which indicates lower cost and risk values for the crew-controlled GN&C system.

Number of Software Functions → [High, Medium, Low]

As the number of functions required to implement an operation increase, the development cost and risk will increase due to the additional development artifacts that need to be designed, implemented and tested. Mission risk will also increase due to the increase in the number of software elements that may not perform as intended.

Again, the number of functions required of the software for the autonomous system and the crew-controlled system are high and low respectively, indicating a decrease in cost and risk when the software developed is used to support the crew instead of replace the crew.

Amount of Software Decision-Making → [High, Medium, Low]

The number of decisions required to implement a given operation affects the amount of code and testing required to ensure that these decisions are implemented and operate correctly. The number of decisions made by the software will increase overall development cost and risk. The additional decision points also increase the number of places the system can make a mistake during operation, thereby increasing mission risk. For autonomous rendezvous and docking control, the amount of software decision-making is high.

Number of Required Interfaces → [High, Medium, Low]

Increasing the number of software interfaces will increase development cost and risk and contribute to increasing mission risk. The increases in development cost and risk are due to the additional analysis, design, code and test required to implement and verify each interface protocol. Mission risk also increases due to the increase in the number of elements that must be controlled to avoid failure during the execution of the operation.

Average Breadth of the Interfaces → [Large, Medium, Small]

The breadth of an interface (i.e. the diversity of control and data elements crossing the interface) will affect development cost and risk and also affect mission risk. Development cost and risk will increase with the breadth of an interface due to the additional design, code, and test elements required to implement the wider number of interface elements. The mission risk will also increase due to number of items, which must be controlled across the interface during operation.

The number of interfaces is higher for the autonomous rendezvous and docking system than for the crew-controlled backup system. In both cases the breadth of these interfaces is at a nominal level, indicating that the implementation does not require passing large amounts of control and data across multiple elements. Consequently, development cost and risk as well as mission risk is slightly higher for the autonomous system than for the crew-controlled guidance, navigation, and control backup system.

4.2.2.2 – HCI Cost and Risk Assessment Parameters

The following parameters are used to evaluate functions beyond the baseline set when a human performs a task with support from software:

- Function of the Human (Primary Controller vs. Lesser Responsibility)
- Difference Between the Communication Delay and the Time Deadline
- Type of Response Required from the Controller
- Situation Awareness Level Required by the Human in the Loop
- Amount of Infrastructure Needed to Support the Controller
- Skill-Level Required to Perform the Task
- Potential for the use of Incremental Control
- Potential to use Feedback
- The Workload Associated with the Task
- Length of the Task
- User-Interface Requirements

Function of the Human (Primary Controller vs. Lesser Responsibility) → [Yes, No]

This parameter addresses the level of automation used to implement particular functions required by a particular system architecture. If the human is the primary controller of the system, cost and risk derive from (1) having a human in the loop and (2) creating software, training and support materials necessary for the human to complete the task. The type of system controller will impact the cost and risk associated with other parameters, including the location of the controller with respect to the controlled system.

In the autonomous rendezvous and docking system for the CEV, which is a manned vehicle, the primary controller of is the autonomous software during rendezvous and docking and the crew provides backup control.

Difference Between the Communication Delay and the Time Deadline → [Small, Medium, Large]

For long-range space missions, the communications delay between the controller and the controlled system may prevent certain function implementation alternatives. For example, having a ground backup for a rendezvous between two vehicles in Martian orbit will increase mission risk because of the small difference between the communication signal delay and the length of the task or deadline. In these cases, it may be more sensible from a risk point of view (despite the increase in cost) to carry out some functions autonomously through software.

In the examples provided, the margin between the delay and the deadline is very large because the human in the loop is onboard the CEV. Consequently, mission risk is not drastically affected. A "one" is used for the mission risk parameter to indicate the multiplier to be used for the next parameter, which addresses the type of response expected of the controller in the amount of time he or she has allotted.

Type of Response Required from the Controller → [Simple, Choice, Decision-Making]

The type of response required by the controller changes the impact of communication delays on mission risk. There are three categories of tasks used in this evaluation to judge the impact on cost and risk given the time margin between the propagation delay and the deadline: simple, choice, or decision-making [134].

A simple response to a signal, such as pushing a button, is achievable and reasonable during a short time span between a communication delay and a controller deadline. However, a decision-making task, in which the controller must determine the appropriate action, takes quite a lot more time. Consequently, if the difference between the communication delay and the time deadline is small and the controller must make a decision, the risk of a mission failure increases. Low mission risk is associated with a short controller response time in combination with a short deadline, which helps to ensure that the requirements are fulfilled by the controller.

In the examples provided, both the software and human controllers reside within the spacecraft executing the particular function. For example, the rendezvous and docking algorithm is executed on board the CEV, the manned vehicle that carries out the function in Marian orbit. Furthermore, the individuals controlling the back-up function of rendezvous and docking are the crewmembers also on board the CEV. As seen in these examples, the type of response required in combination with difference between the communication delay and the timing deadline determines the amount of mission risk associated with the task required from the controller.

Situation Awareness Level Required by the Human in the Loop → [Level 1 – Perception of Elements in Current Situation, Level 2 – Comprehension of Current Situation, Level 3 – Projection of Future Status]

Situation awareness is the combined operations in perception, working memory and long-term working memory that enable the decision maker to entertain hypotheses about the current and future state of the world [99]. According to Endsley, situation awareness can be broken up into three different levels: (1) the perception of the elements in the environment within a volume of time and space, (2) the comprehension of their meaning and (3) the projection of their status in the near future [99].

For crew-controlled rendezvous and docking of two spacecraft, Level 3 situation awareness is required. The crewmember will have to project the status of the vehicle stack in the near future to determine the appropriate maneuvers to successfully mate with that stack. Ensuring that the user interface provides the human controller with the situation awareness required to rendezvous and dock with another spacecraft will increase the development cost and risk of the crew-controlled back-up system.

Amount of Infrastructure Needed to Support the Controller → [Substantial, Moderate, Minimal, None]

Another aspect of human-computer interaction involves the infrastructure needed to support the controller. Infrastructure refers to and includes everything from training materials and simulators to existing controls and displays. Four categories are used to describe any changes required to the preexisting infrastructure available to create an effective HCI: substantial (new infrastructure required, major adaptations to existing infrastructure), moderate (no new infrastructure, adaptations to existing infrastructure), minimal (minor adaptations to existing infrastructure), or none.

For a crew-controlled GN&C, substantial adaptations of existing user interfaces and controls are needed to provide appropriate infrastructure to support the controller. Training materials and simulators will have to be developed to support the new CEV controls and displays. Furthermore, current rendezvous and docking controls will have to be adapted to fit with the type of docking mechanisms chosen for the Moon and Mars missions.

Skill-Level Required to Perform the Task → [High, Medium, Low]

This parameter reflects the amount of training, knowledge, and/or expertise required to perform the task, measured on a low, medium, high scale. It indicates the development cost and risk associated with providing the training needed to accomplish the task as well as the mission risk associated with performing a task that requires a particular skill level.

Developing the necessary, high level of expertise required to rendezvous and dock with another vehicle will greatly increase development cost and risk as well as mission risk. However, these effects may be lessened if incremental control is possible.

Potential for the use of Incremental Control → [Yes, No]

If a critical task can be broken up into a series of discrete tasks, such as "arm, aim, fire," the controller can use feedback from the behavior of the controlled process to assess the correctness of the incremental actions and take corrective action before significant damage is done [74].

Crew-controlled rendezvous and docking requires continuous control on the part of the operator, and incremental control is not possible. Consequently, the mission risk associated with the high skill level required to perform the task is compounded.

Potential to use Feedback → [Yes, No]

Open-loop control systems, or systems without feedback, have a tendency toward instability. Consequently, constant feedback is preferable from a mission risk perspective. Human controllers are especially sensitive to a lack of feedback, which further lowers situation awareness [134]. Consequently, if feedback cannot be provided to the controller, mission risk increases significantly.

In the case of crew-controlled rendezvous and docking, feedback can not only be provided through displays, but also through a docking window (which is the current method of performing docking on board the shuttle).

The Workload Associated with the Task → [High, Medium, Low]

Using the subjective workload assessment (SWAT) technique, which measures workload on three three-point scales, we can determine the mission risk associated with a human controller carrying out a specific task. Workload is measured with respect to Time Load, Mental Effort Load and Stress Load [134]. Each of these workloads is rated as high, medium, or low.

Time Load refers to the amount of spare time the operator has when completing a task and the number of interruptions or overlaps between tasks. Mental Effort Load measures the amount of conscious mental effort, or concentration, required by the task. Attention is also an important factor in Mental Effort Load. Finally, Stress Load refers to any confusion, risk, frustration or anxiety related to the task. The ideal level for each of these three measurements of workload is medium; two little workload creates complacency and a lack of attention, while too much workload creates a stressful working environment increasing mistakes and confusion.

For crew-controlled rendezvous and docking, the Time Load is high, while the Mental Effort Load and Stress Load are medium indicating the demanding nature of the task. The task takes all of the controllers' attention, with little to no spare time available. Furthermore, high levels of concentration are needed to successfully mate the two vehicles. Stress load, although greater than during nominal operations, is not rated high, because of the amount of training and simulations the controller will have gone through before the actual operation takes place.

Length of the Task → [Prolonged, Medium, Short]

The length of the task also impacts the mission risk associated with the implementation strategy. The length of the task can be classified as either prolonged, medium or short. Prolonged tasks increase mission risk, because of the extended amount of time and resources spent on carrying out that particular task. Prolonged tasks also increase the time component of workload, thereby increasing mission risk.

In the case described in this example, rendezvous and docking of a manned spacecraft, the duration of monitoring the state of the CEV affects the mission risk associated with the human in the loop that is controlling this particular task. The longer the human must monitor the state of the controlled system, the more susceptible the human is to making a mistake. Conversely, the shorter the required response or action, the less likely the human is to make a mistake concerning the current state of the system. However, depending on the mode of stimuli (auditory, visual or manual) most natural to the controller's response, the amount of associated mission risk varies.

User-Interface Requirements → [Yes, No]

User interface design is an important aspect of both cost and risk. Creating an effective user interface for human-computer interaction may drive up development cost and risk. However, not providing the user with such an interface or an inadequate interface will drastically increase mission risk. A user interface for rendezvous and docking is required and also complemented with a window/viewing area that allows the controller to directly view the docking mechanisms.

4.2.2.3 – Cost and Risk Assessment Parameters for Software and HCI

The following questions are evaluated for functions allocated to both software and a human controller:

- Need for Ultra-High Reliability
- Number of Control Mode / State Variables in the Controller's Process Model
- Reliability of the Source of Information Update
- Environmental Considerations and their Impact
- Criticality of Software and Human Actions
- Potential Mitigation Strategies

Need for Ultra-High Reliability → [Yes, No]

Almost all the systems and functions necessary for the space exploration initiative require ultra-high reliability. One example of a function that does not require ultra-high reliability is video communication – a failed video communication system will not impair the crew's ability to return safety. Requiring ultra-high reliability drives up development cost and risk substantially.

Both automated and crew-controlled rendezvous and docking require ultra-high reliability and therefore cost and risk are again compounded.

Number of Control Mode/State Variables in the Controller's Process Model → [High, Medium, Low]

Every controller (whether automated or human) must use a model of the controlled process in order to provide effective control [78]. The complexity of the controller's process model will increase all cost and risk for both humans and automation. For automation, a large number of control modes and states in the process model increases the development cost of the software. The more complex the process model, the greater the number of required functions and interfaces to accomplish the task. Furthermore, the cost and risk associated with testing the different states of the controller's process model increases as the complexity level of the process model increases.

Similarly, for a human controller, a complex process model (usually called a mental model in humans) increases the mission risk associated with the task, because of the increased probability of mode confusion or other condition caused by mismatches between the controller's mental model and the actual process state.

The automated rendezvous and docking system and the crew-controlled system both have the same number of control modes in their process model, because they are controlling the same process.

<u>Reliability of the Source of Information Update</u> → [Yes, No]

Even if feedback is provided to the controller, the reliability of the feedback information must also be considered. Feedback is helpful to the controller if the source for information update is reliable; if not, feedback can be detrimental to the controller's efforts to successfully carry out the task.

As previously stated, sensors on board the CEV coupled with the crew-controlled backup docking window provide an excellent source for reliable information update, and therefore mission risk is not adversely affected.

<u>Environmental Considerations and their Impact</u> → [High, Medium Low]

Several aspects of the environment may impact the level of mission risk associated with a particular function implementation on a spacecraft. For example, "Proximity to Other Unmanned Vehicles" will increase mission risk, because there is a possibility of interference or collision, but it will not increase mission risk as much as "Proximity to Other Manned Vehicles" where a collision could result in loss of life as opposed to loss of mission. Other environmental considerations include surface and space environment and whether or not the controlled vehicle is currently docked with another vehicle. A full listing of the environmental considerations is found in the questionnaires in Appendix D.

Rendezvous and docking in Mars orbit, whether it be autonomous or crew controlled, must take into consideration two environmental factors: (1) Proximity to Other Unmanned Vehicles and (2) Low-Mars Orbit Space Environment. Proximity to Other Unmanned Vehicles has high impact on the cost and risk factors for autonomous rendezvous and docking, while for crew controlled rendezvous and docking it has low impact. The autonomous case may require some proximity detection to avoid collision, which will increase development cost and risk slightly, and mission risk greatly. The Low-Mars Orbit Space Environment has low impact for both the autonomous and crew controlled cases.

Criticality of Software and Human Actions → [High, Medium, Low]

The safety-related nature of most functions needed for human space exploration is a major driver of cost and risk for the software and human-computer interactions. Cost increases as a result of the testing, hazard analysis, and additional verification and validation required for certification of safety-critical software. Training a human controller to perform a task that is safety-critical may also be expensive. Providing the controller with an appropriate and safe user interface requires testing and additional verification and validation. However, it is important to note that incorporating design for safety techniques into the software lifecycle from the beginning has the potential to drastically reduce these additional costs, because errors and unsafe designs are not created. Furthermore, some hazards can be mitigated using non-software or simpler solutions. Depending on the method and type of mitigation strategy used, cost and risk may be further increased.

Several hazards are associated with rendezvous and docking including (1) uncontrolled approach attitude leading to collision and (2) inability to perform successful docking [37]. Depending on the method and type of mitigation strategy used, cost and risk may be further increased.

Potential Mitigation Strategies → [Elimination, Prevention, Control, Damage Reduction]

The identified hazards need to be mitigated in some fashion. This mitigation strategy may involve additional software, human procedures and/or software to support the human procedures. Furthermore, the type of mitigation strategy employed will impact the level of mission risk associated with the function. Mitigation strategies aimed at hazard elimination present far less mission risk than strategies only aimed at hazard control and damage reduction.

For the automated rendezvous and docking example, the hazards associated with the task are mitigated by creating the crew-controlled backup system described as the second part of the example. This system mitigates the hazard by allowing the crew to take over control in the event of an unstable attitude or a missed docking. Furthermore, the type of mitigation strategy employed will impact the level of mission risk associated with the function. Mitigation strategies aimed at hazard elimination present far less mission risk than strategies only aimed at damage reduction. In the case of automated rendezvous and docking, hazards cannot be eliminated but only controlled.

A full listing of the results of evaluating both the automated and crew-controlled rendezvous and docking functions with respect to the parameters is shown in Table 3, Appendix D. The next step is to evaluate all the functions beyond the baseline set for a variety of system architectures with respect to these parameters in order to provide the system architects with an impact analysis of their system architectures on the software. Functions outside the baseline set are identified for each of the system architectures and then each of these functions is evaluated (like the examples provided in this section) with respect to the assessment parameters. This impact analysis was applied to fourteen potential mission architectures out of almost 2000 architectures generated by the system engineers (ten transportation architectures and four surface operation architectures). Figures 4-2 and 4-3 depict the evaluation results from the Lunar and Mars analyses respectively.

The sum of the evaluation scores assigned to development cost, development risk and mission risk were normalized to an average cost/risk case for both the Lunar and Mars examples. The Lunar examples are normalized to OPN 43/Short Stay and the mars examples to OPN 938/600-day are shown in Figure 4-2, where the average cost/risk cases represent 100% cost and risk and the other missions are some percentage of that average case. Development cost, development risk and mission risk are depicted in three sections from left to right with the five Lunar missions shown in each section. Each bar shows what percentage of that cost or risk results from transportation (shown in blue) and surface (shown in purple) components. These components are further decomposed into software and HCI portions.

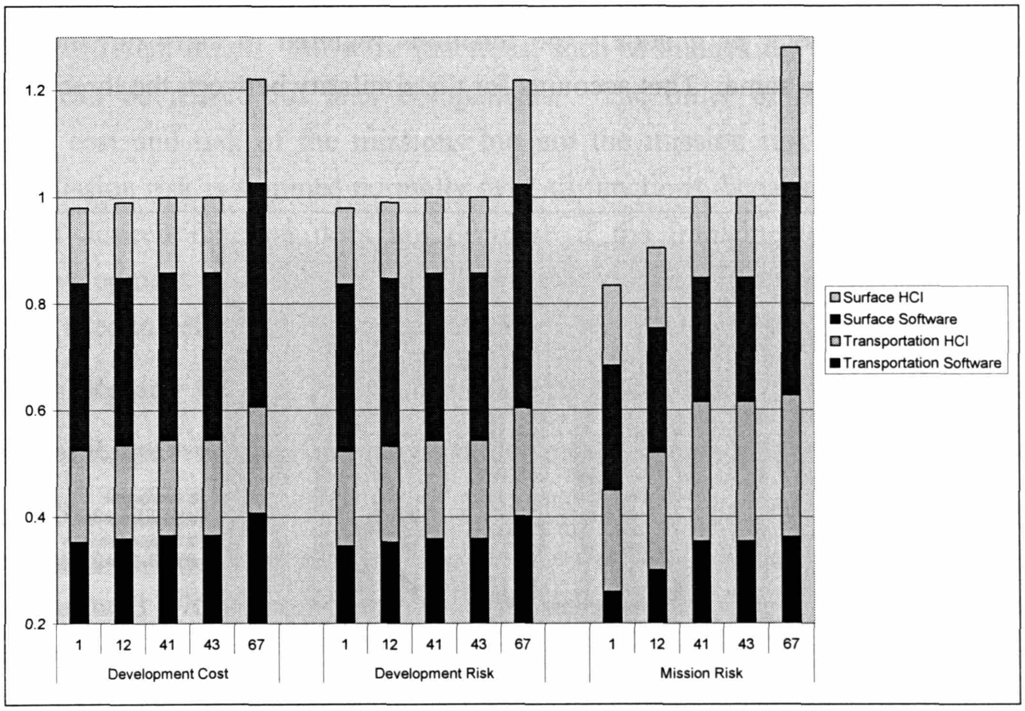


Figure 4-2. Lunar Architecture Evaluation Results

For the Lunar mission architectures (Figure 4-2), OPN architectures 1, 12, 41 and 43 show roughly equivalent development cost and risks, however OPN architectures 1 and 12 had significantly lower mission risk. OPN architecture 67 shows the highest development cost and risk and mission risk, primarily due to the additional assets required to accomplish the mission.

As shown in Figure 4-2, the jump from a Short Stay to a Long Stay on the Moon increases the cost and risk of the mission substantially. Furthermore, the system architectures with fewer vehicles and fewer vehicle interactions, the best example of which is labeled OPN 1, are lower in development cost and risk and especially in mission risk, simply because there is much less software required to support the mission. In addition, the majority of the cost and risk associated with each mission derives from the surface operations as opposed to the transportation architecture, which is another interesting aspect of this analysis. Surface operations will be a costly, and risky, portion of the Space Exploration Initiative.

Figure 4-3 shows the results of the Mars architecture analysis. For the Martian mission architectures, OPN 881 had the lowest development costs and risks and mission risks. Architectures 938, 969 and 970 had roughly the same level of costs and risks. Architecture 881 with a 60-day stay seems to be the best case in terms of cost and risk from a software and HCI perspective. Architecture 969 with a 600-day stay, the system architecture is quite similar in total cost and risk as compared to the other 600-day stay architectures. These three architectures have nearly the same transportation architectures aside from details such as the orbit in which various vehicles must dock or undock: the functions required to carry out these missions, however, are virtually the same. This accounts for the similarity between the three 600-day stay Mars architectures.

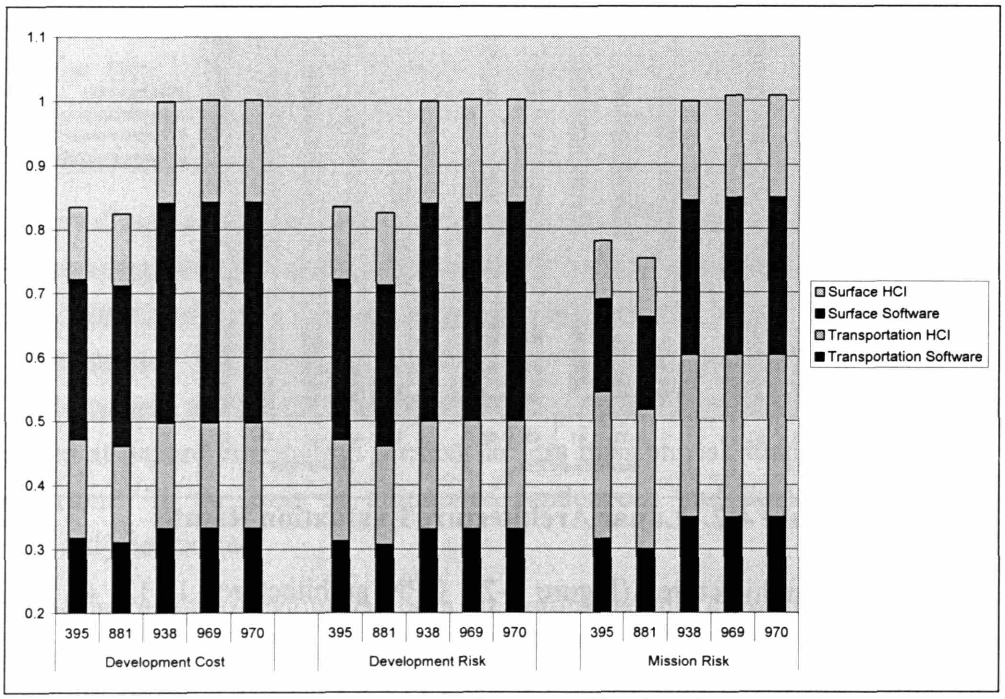


Figure 4-3. Mars Architecture Evaluation Results

The results of evaluating system architectures with respect to these parameters includes relative cost and risk rankings. These results can then be compared to one another to determine which system architectures are feasible with respect to the software system required to support those architectures. However, this analysis assumes that the functions are implemented without a product-line approach and not reused from one vehicle to the next. The following section describes the impact of applying a product-line reuse strategy to the results of the cost and risk analysis.

4.2.2.4 – The Impact of Software Product Line Reuse on Cost and Risk

Many of the functions developed for one vehicle may be reused on other vehicles. Functions that can be reused from one vehicle to another should be developed with the most stringent requirements from the beginning. For example, if autonomous rendezvous and docking algorithms are required for a manned vehicle and also required for two unmanned vehicles, the software should be developed to be man-rated from the beginning. The cost of adapting software that was not initially designed for safety will be prohibitive and in some cases more expensive than writing new software [74].

Appendix A depicts a typical development timeline for the system architecture described in this paper. The physical components of the architecture, listed in the left hand column, will be developed at different times. Software functions, such as undocking, developed for the earlier components can be reused for later components. The order of development impacts the development cost and risk of the missions but not the mission risk associated with various functions. Mission risk is summed normally over all functions, because the risk associated with performing a desired function does not decrease if the implementation strategy is reused. However, development cost and risk drop dramatically once a function has been implemented. The equation used to define the exponential decay of the development cost and risk is:

Cost of Reuse = Cost of Original Development * 0.3^x, where x = the instance of the reuse

Risk of Reuse = Risk of Original Development * 0.3^x, where x = the instance of the reuse

The factor 0.3 is derived from several studies of software product lines and the savings that the various products in the line experience throughout time [31]. Subsequent products in the line usually have around a 70% savings from the previous product. This estimate is sufficient for the high-level cost estimation performed in these early stages of development. Consequently, the evaluations take into account reuse of implementation strategy development throughout the timeline shown above. For example, OPN 969 with a 600-day stay saves roughly 61% in development cost and risk by reusing assets from one mission to the next.

Figures 4-2 and 4-3 change once reuse discounts are applied. Figure 4-4 shows the five Mars system architectures evaluated with respect to development cost before and then after the reuse discounts of the previous section were applied. Although the previous graph shows architecture 881 to be the best, after reuse is taken into account, architectures 395 and 881 are very similar with respect to software development cost and therefore perhaps other criteria should be used to decide between those two system architectures.

From a software perspective, simple mission architectures tend to have lower development costs, development risks and mission risks. As system architecture complexity increases, so do the software costs and risks. For architectures with vehicles and surface elements with similar functional requirements, product-line development approaches tend to reduce the effect that the additional complexity has on development costs and risks, but do not affect the additional mission risks.

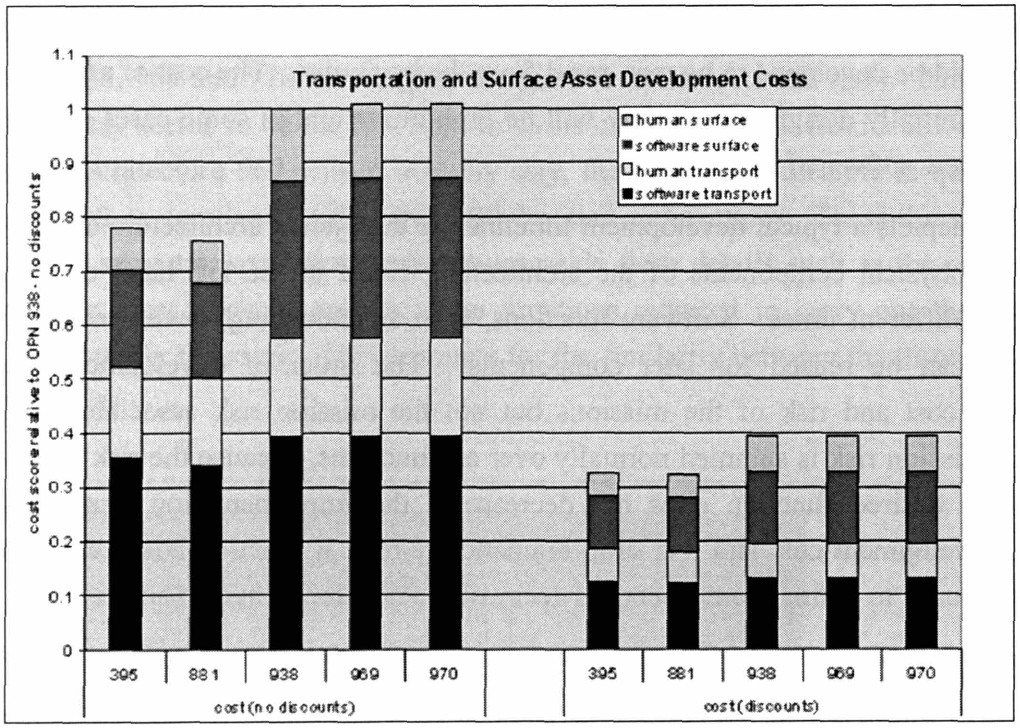


Figure 4-4. Mars Architecture Evaluation With and Without Discounts

4.3 – Requirements

Traditional requirements phases usually include high-level requirements, requirements analysis, and requirements specification. During the high-level requirements phase, the software engineers work with the customers to define the goals, requirements and constraints on the system software. During this phase of the software development lifecycle, the steps in developing a software architecture and establishing an approach to evaluation include the following:

1. Using preliminary concepts for system architectures, identify several options for software architectures that support the system architectures
2. Identify information important to the product-line architecture
3. Refine the set of quality attributes and identify preliminary set of criteria associated with these qualities
4. Identify views necessary to provide information about the set of quality attributes and requirements associated with the software architectures and any architecture description languages that will be used in concert with these views

4.3.1 – Step 1: Identification of Software Architecture Options

Four architecture styles were identified as possible for development and evaluation to support the system architecture chosen for the Exploration Initiative. It is important to note, however, that in this stage of development no architecture options should be initially discounted. These spacecraft software architecture styles include: traditional spacecraft software architecture (decomposition into system, subsystem, component and fault protection nodes), functional decomposition (decomposition into functional elements of the spacecraft such as maintain temperature or maintain attitude), state analysis (such as the Mission Data System architecture currently under development at JPL), and a distributed, net-centric architecture that employs an implicit invocation strategy (as described by the DoD).

Figures 4-5 – 4-8 describe the four prospective architecture styles chosen for possible use in constructing a software architecture for the Exploration Initiative. These four architecture styles were chosen because of their relevance to current trends in the spacecraft software engineering domain. The goal-based and net centric architecture styles were chosen because of the current research interest in these styles for space applications from NASA and the DoD, respectively. A traditional subsystem decomposition is also studied, because this has been the spacecraft software paradigm for many years and it will be interesting to determine the costs and benefits of switching from this traditional architecture to a new one. Finally, a pure functional decomposition is also studied, because of similarities to both the goal-based system and the traditional subsystem decomposition.

Style: Goal-Based (State Analysis)

Problem: Developed to support resource-limited, real-time systems that must operate in an asynchronous environment, in particular space systems. The main benefit of this style is that fault tolerance is no longer considered a separate entity. Because a failure mode or other anomalous condition is treated as just another possible set of system states, the spacecraft does not have to alter its nominal operations. It simply relegates the fault by attempting to fulfill its mission goals given its current state, a process no different than if the spacecraft was not in a fault detected state.

Context: This style requires that almost all aspects of the controlled system be represented as states. Goals are constraints on a state over a set time interval.

Solution:

System Model: All actions are directed by goals instead of command sequences. A goal is a constraint on a state over a time interval.

Components: Hardware Adapter, Controller, State Variable, and State Estimator. In the context of spacecraft control, a state variable is used to represent the following aspects of the controlled system: dynamics, environment, device status parameters, resources, data product collections, data management and transport policies, and externally controlled factors.

Connectors: Goals, State, Measurements, and Commands

Control Structure: Control is achieved through the set of goals uploaded to the controller and goal elaboration, which is the process by which a set of rules recursively expands a high level goal into a goal network.

Variants: Goal-based systems are closely related to more traditional artificial intelligence systems. In these architectures, planning algorithms direct agents to accomplish different, explicitly stated tasks. Based on a set of rules, instead of goals, the agents carry out the tasks.

Examples: Mission Data System

Figure 4-5. Goal-Based Architecture Style

Style: Traditional Subsystem Decomposition

Problem: The system is described as a hierarchy of system components. The system is decomposed into a series of subsystems, sub-subsystems, etc. until the individual components are reached. This style is especially useful for non-resource limited systems that can be hierarchically decomposed, not by function, but by structure and in which the functionality of the subsystems does not vary over time.

Context: In this architecture style, FDIR components are independent of the system, subsystem and component modules. FDIR components are often added as independent modules with interfaces into the subsystem software.

Solution:

System Model: Processes are invoked through the command sequences uploaded to the system level computer. In spacecraft, these sequences consist of a series of time-stamped commands from the ground controllers, usually written on-the-fly, that explicitly state control actions for the various subsystems.

Components: System, Subsystem, Sub-subsystem, Component, FDIR

Connectors: Control, Status

Control Structure: There is a single thread of control at the system level that executes command sequences uploaded to the command and data handling computer to achieve the mission objectives.

Variants: Traditional subsystem decomposition is one type of structured programming architecture style. This style is similar to functional decomposition.

Examples: Cassini, STS, etc.

Figure 4-6. Traditional Subsystem Decomposition Architecture Style

Style: Functional Decomposition

Problem: The system is described as a hierarchy of system functions. The system is decomposed into a series of functions, sub-functions, etc. until a set of functions is reached that can be implemented independently from one another. This style is especially useful for systems that can be hierarchically decomposed, and whose behavior drives the system instead of the structure. In addition, functionally decomposed systems can be easily distributed among multiple processors. Functional decomposition is often referred to as stepwise refinement and is one of the earliest forms of structured programming.

Context: Because functional decomposition is a form of top-down development, it can be easily analyzed with respect to safety. Most hazard analysis techniques, such as FTA and FMEA, required a model of behavior to identify hazards and therefore functional decomposition is especially useful for safety-critical systems.

Solution:

> **System Model**: Relationships between software modules are categorized by data flow. Functions are called by another function and data is passed between the functions to accomplish a task.
>
> **Components**: Functions, Sub-functions
>
> **Connectors**: Send/Receive Data, Function call
>
> **Control Structure**: One or more independent threads of control. Multiple threads used in distributed processing systems. Functions are called as needed.

Variants: Functional decomposition is one type of structured programming architecture style. This style is similar to the traditional subsystem decomposition of spacecraft. This architecture style can be used in conjunction with the layered architecture style, because both styles decompose the system with respect to behavior, or functions, and therefore complement one another.

Figure 4-7. Functional Decomposition Architecture Style

Style: Net-Centric (Implicit Invocation)

Problem: Consists of a loosely-coupled collection of components, each of which carries out some task and may enable other operations. The major characteristic of this style is that it does not bind recipients of signals to their originators. It is especially useful for applications that need to be able to be reconfigured, by changing a service provider or by enabling and disabling operations.

Context: This style usually requires an event handler that registers components' interests and notifies others. This can be achieved either through specialized operating system support or though specialized language features. Because of the intrinsically decentralized nature of systems designed this way, correctness arguments are difficult. For the same reason, building a mental model of such systems during program comprehension is difficult too.

Solution:

> **System Model**: Processes are independent and reactive. Processes are not invoked explicitly, but implicitly through the raising of an event.
>
> **Components**: Components are processes that signal events without knowing which component is going to react to them. Conversely, processes react to events raised somewhere in the system.
>
> **Connectors**: Components are connected through the automatic invocation of processes that have registered interest in certain events.
>
> **Control Structure**: Control is decentralized. Individual components are not aware of the recipients of the signals.

Variants: There are two major categories of systems exploiting implicit invocation. The first category comprises the so-called tool-integration frameworks as exemplified by many software development support environments. They consist of a number of 'toolies' running as separate processes. Events are handled by a separate dispatcher process, which uses some underlying operating system. The second category consists of languages with specialized notations and support for implicit invocation, such as Java Beans.

Examples: Java Beans, J2EE

Figure 4-8. Net-Centric Architecture Style

4.3.2 – Step 2: Identification and Support for Product Line Variation Points

In order for an architect to prepare a software architecture that allows for change, variation points between different products in the software product line need to be identified and then support for those variation points built into the architecture. To enable the adapters of a software product-line architecture to achieve their objectives from one product to the next, an explicit documentation of planned variability in the architecture is required. As stated in [26], "An architecture for a product line may encompass a collection of different alternatives for dealing with the variation among products. Capturing these alternatives and the rationale for each alternative enables the team constructing a product to have a list of potential solutions to choose from." The six types of variation are: function, data, control flow, technology, quality goals and environment [26]. Support for these types of variability can be built into any software product-line architecture. The variation can be optional, an instance out of several alternatives, or a set of instances out of several alternatives.

There are several areas of the software product-line architecture that must be constructed to support variation points for the Exploration Initiative. Since spacecraft can be easily and naturally decomposed according to their subsystems, this decomposition will also be used for identifying variation points. Figure 4-9 illustrates how the subsystem functions described in Appendix C can be grouped into similar vehicle and facility classifications.

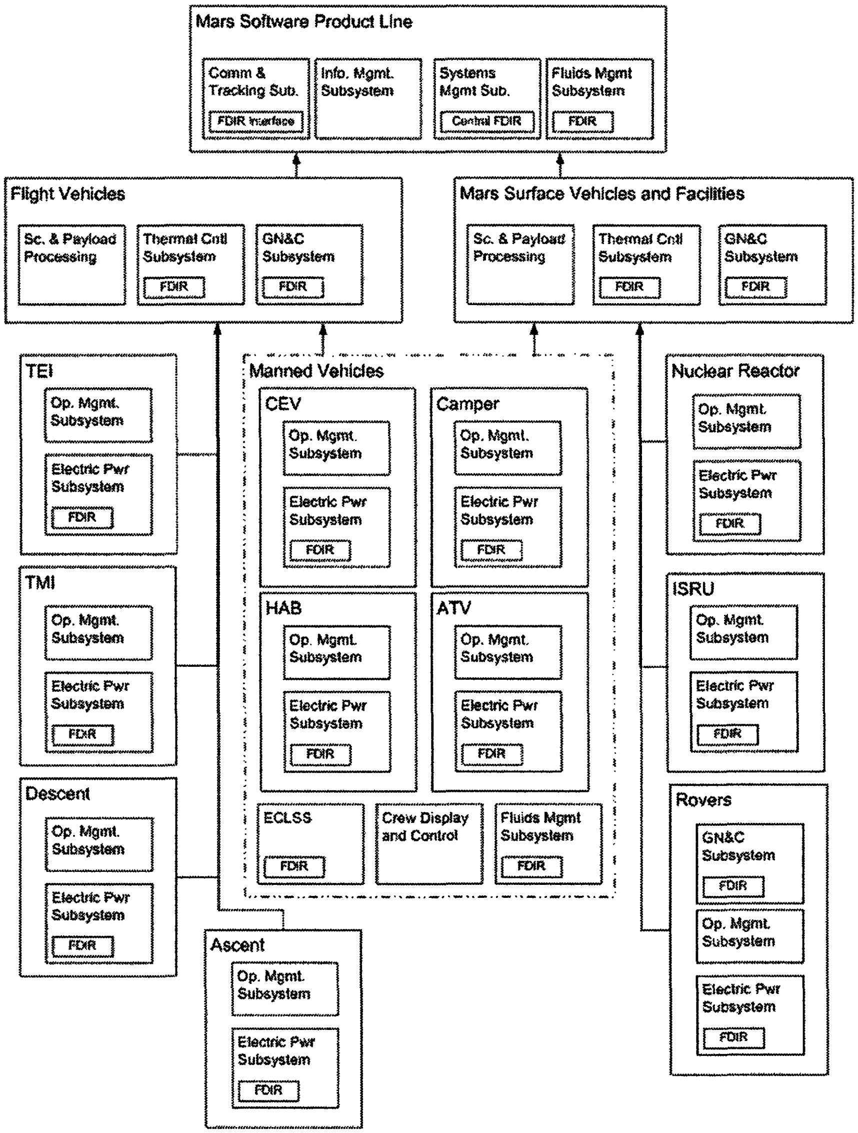


Figure 4-9. Mars Software Product Line Vehicle and Facility Groupings

As seen in Figure 4-9, several subsystems can be grouped at the software product line level. Generic models (described in Chapter 2) can be created for communication and tracking, information management, systems management and fluids management because each vehicle has these subsystems in some fashion as well as their Failure, Detection, Isolation and Recovery (FDIR) interfaces. The vehicles can then be grouped into flight vehicles and Mars surface vehicles and facilities, because these two groups of vehicles and facilities have very different requirements for science and payload processing, thermal control, and GN&C subsystems.

Furthermore, these vehicles and facilities can also be grouped according to whether or not they are manned or unmanned. All manned vehicles share an Environmental Control and Life Support Subsystem (ECLSS), crew display and controls, and additional fluids management needs outside of the standard fluids management on board unmanned vehicles. Operations management and electric-power systems are specific to the needs of the individual vehicle and therefore cannot be generalized for a class of vehicles.

Specialization of these generic models and tailoring the software to the specific needs of a particular vehicle and mission can occur in tandem with the model development. Through accretion of modular software assets, the software system evolves and extends throughout the duration of the Exploration Initiative, thereby aiding in limiting the obsolescence of the software system. The following section discusses a development schedule that supports the product line approach and the gradual accretion of software functionality over time based on the requirements of the missions.

Through this organization of vehicles and facilities, variation points between the vehicles and facilities used in the Exploration Initiative can be categorized at the vehicle/facility level or subsystem level. It is important to note that as the software architecture emerges and is defined and evaluated during the design phases, information about the variation points will become more specific.

Vehicle/Facility Level Variation Points → The main difference between software systems at the vehicle and facility level of the software product line involves whether or not the system will contain certain groupings of functionality, or subsystems. As seen in Figure 4-9, several of the vehicles and facilities are grouped into a manned category, while others are unmanned. The manned systems will all contain ECLSS functionality, while the unmanned vehicles need not have this software as part of their system. Furthermore, vehicles within the same category (i.e. the CEV and campers are both manned vehicles) may have slight variation points within subsystems common to both vehicles. This difference will undoubtedly lead to mostly function and data variation points. For example, although both the CEV and the campers require ECLSS, the CEV ECLSS must support six astronauts, while each camper's ECLSS only supports three. This difference in crew member support leads to a data variation point in the ECLSS portion of the software system architecture. The CEV will inevitably have to carry more Lithium Hydroxide canisters (used for scrubbing Carbon Dioxide out of the air) than the campers. The software system will have to cater to that difference in the portion of the ECLSS that deals with oxygen production and air quality maintenance.

Another difference at this level involves mission objectives. Depending upon the particular mission for which the vehicle or facility is intended, the requirements for the software may be strikingly different. For example, there are three major destinations for vehicles and facilities in the Exploration Initiative: low-Earth orbit (LEO), the Moon, and Mars. These varying mission objectives lead to all six types of variation points: function, data, control flow, technology, quality goals, and environment. To use the ECLSS example again, variation exists between the ECLSS of the CEV and campers not only because of the number of humans on board, but also because of the environment in which these vehicles must operate. The CEV operates in the space environment, while the campers operate in the Martian environment. Consequently, the camper ECLSS must take into consideration the Carbon Dioxide atmosphere, which the CEV ECLSS does not. This atmospheric difference affects the thermal properties of the vehicle and consequently some of the control algorithms that regulate the air scrubbing.

Subsystem Level Variation Points → At the subsystem level, differences between the vehicles themselves lead to variation points in all six categories. For example, all of the space vehicles will have similar GN&C subsystems. However, some of these space systems will require components within their GN&C subsystems that are not necessary in other vehicles. For example, although both the CEV and descent stages both require a GN&C subsystem, the functions that each of these subsystems performs will be very different. The CEV GN&C must have the capability of navigating from LEO to LMO (Low-Martian Orbit). The CEV must then maintain orbit around Mars. The descent vehicle, on the other hand, is towed into Martian orbit by the Trans-Martian Injection (TMI) stage. It therefore does not have its own propulsion system for traversing the space between the Earth and Mars and does not require software to support that function. However, the descent stage must perform the Entry, Descent and Landing (EDL) GN&C functions for the Hab / Descent Stage stack. Consequently, the descent stage GN&C subsystem contains an EDL component not required in the CEV GN&C subsystem.

Clearly, the largest variation points are found at the subsystem level and are the result of the specialized functions that are required from each vehicle. These variation points, as well as the more subtle, system-level variation points, must be taken into consideration by the software system architects from the beginning of development to help ensure that the product-line architecture is suitable for all vehicles and facilities within the product-line scope.

Step 3 in architecting during the high-level requirements phase involves refining and elaborating upon the quality attributes described during the concept generation phase as well as identifying a preliminary set of evaluation criteria associated with those attributes.

4.3.3 – Step 3: Refine Quality Attributes

During the requirements and design phases of the software development lifecycle, software architecting efforts are focused on constructing an appropriate software architecture and ensuring that the desired quality attributes are built into that architecture. One of the first steps in incorporating the quality attributes into the software architecture involves elaborating upon the quality attributes and then identifying several criteria for evaluating candidate software architectures with respect to these criteria.

According to Leveson in [74], the basic software system safety tasks include:

1. Trace the system hazards to the software-hardware interface
2. Translate the software-related hazards into requirements and constraints
3. Determine the consistency of the software constraints with the requirements
4. Demonstrate completeness with respect to the system safety parameters
5. Based on the software system safety constraints, develop system-specific software design criteria and requirements, testing requirements and Human-Computer Interaction (HCI) requirements
6. Trace safety requirements into the code
7. Identify software that controls safety-critical operations and concentrate safety analysis and test efforts on those functions and the safety-critical path that leads to their execution
8. Develop safety related software test plans, descriptions, procedures and case requirements
9. Perform HCI analysis and an analysis of the interfaces between critical and non-critical software
10. Review the test results
11. Review and trace software problems back to the system level
12. Assemble safety-related information

Based on these software system safety tasks, as well as information on real-time software systems [114] and spacecraft engineering [132], the seven desirable quality attributes for the Exploration Initiative (analyzability with respect to safety, ease of verification and validation, sustainability, affordability, buildability, ability to meet real-time constraints, and "monitor"-ability) are expanded upon.

4.3.3.1 – Analyzability with respect to safety

Based on these tasks, several criteria were identified for determining whether or not a particular software architecture is analyzable with respect to safety. These criteria include reviewability, traceability, isolation of critical functions and assurance of completeness and determinism. Most of these criteria have been incorporated into the SpecTRM system and software engineering development environment [76], [111].

Reviewability → Spacecraft in general, and spacecraft software in particular, are highly complex systems. In order for these systems to be analyzable with respect to safety, it is important that the semantic distance between the model of the system (in this case the software architecture) and the system itself be minimized. Engineering the software in this way will help spacecraft engineers, who are already experts in the spacecraft system, to better understand the system model they are working with and therefore make reviewing that system model easier.

One of the ways of supporting software review by engineers is the use of software architectures that promote model-based development. As described in Chapter 2, model-based development is a system development process organized around models of the desired, externally visible system behavior. These models form a complete system specification and can be used for analysis before implementation. Model-based software development plays an integral role in supporting reviewability in software architectures. Models are used to describe the externally visible behavior of the software components, thereby abstracting away design details and creating a formal requirements specification that can be tested, verified and validated before any design or implementation work has begun. These models can also be reused from one system to the next, which, as outlined in [77] and [131], helps to alleviate some of the problems commonly associated with code reuse in software engineering. Examples of model-based development include Intent Specifications and SpecTRM-RL used to model TCAS II, a conflict avoidance system, and ADAMS and POST used to model the Entry, Descent and Landing (EDL) system of the Mars Exploration Rovers (MER) [37].

Simply using model-based development does not ensure reviewability, however. The modeling language must be easily understood by the reviewers. Zimmerman and Leveson have identified and experimentally evaluated the features of modeling languages that promote readability and reviewability [139]. Many current model-based development languages are no easier to read and review then low-level code (and sometimes less so).

Traceability → Tracing safety requirements through design and into code as well as recording the design rationale behind the requirements, design, and coding decisions is essential for a software system to be analyzable with respect to safety. Traceability does not merely refer to the standard traceability matrix showing the mapping between high-level requirements and software modules, but also to the system design features and decisions in the software architecture that lead to implementation decisions. Not having this type of system knowledge and traceability can lead to accidents, especially when modifying reused software modules, as in the cases of both the Ariane and Milstar launch accidents. This becomes even more important when safety needs to be reevaluated due to changes in the software, which will inevitably occur [77].

Furthermore, design rationale also needs to be included in the software architecture and traceable from the requirements and to the implementation. The lack of such information has been cited in some of the most well-known spacecraft accidents. In the architecture development and selection process, software engineers must record not only what they are building, but also why they are building it in that way. In terms of the Exploration Initiative, it becomes even more essential for this information to be recorded in the software architecture description, because this artifact will undoubtedly be used throughout the long duration of the endeavor as well as changed and reused to fulfill new vehicle and facility requirements.

Isolation of Critical Functions → One of the most important tasks in software safety engineering involves modularizing the software with respect to what needs to be analyzed. Consequently, for spacecraft control software, it becomes imperative that the critical functions (those functions that control safety-critical operations) be isolated so that analysis can be focused on those modules.

One example of a software architecture style that does not support the isolation of critical functions is object-orientation. In an OO design, modules are abstracted from objects in the system. This type of abstraction in software leads to functions being dispersed over a wide range of objects. Safety-critical functions, therefore, become extremely difficult to analyze because they are not cohesively located in one module.

Furthermore, as described in [39], writing software in the spacecraft-engineering domain is quite different from writing software for traditional enterprise applications and other information management systems, because spacecraft software resides in a resource-limited environment. Consequently, mission activities must be designed with these limitations in mind; turning on a camera to take pictures uses power and storing those pictures consumes non-volatile memory. The physical side effects in this case are non-negligible and therefore must be managed; monitoring the effects of a heater on power consumption and the effect of temperature on a sensor measurement become paramount.

Architectures are needed that take into consideration both a systems view of the spacecraft as well as the complex physical interactions between the traditional subsystems. In other words, it is necessary that critical spacecraft software functions be isolated not only from a safety standpoint, but also from a physical standpoint, because OO-type, or structural-type, of decomposition leads to low cohesion, high-coupling and an added measure of difficulty for analysis.

Assurance of Completeness and Determinism → Completeness and determinism are two criteria desirable for safe software systems. A software system is incomplete if the system or software behavior is not specified precisely enough because the required behavior for some events or conditions is omitted or is ambiguous (is subject to more than one interpretation). Mathematically, from a state machine perspective, this means that for any given system state and set of inputs, each state and mode must have a transition. Determinism, on the other hand, refers to the property of the software system that for any given system state and set of inputs, there should be only one transition for each state and mode.

In order for a software system to be analyzable with respect to safety, the software engineers must be able to ensure that the software is both complete and deterministic. There are several software architecture options that are immediately eliminated by this requirement. For example, several artificial intelligence (AI) architectures cannot be used because they employ a non-deterministic and probabilistic approach to path planning [110]. In some cases, this precludes the architecture from being analyzed with respect to safety, because the state transitions are based on probability and not built into the system before runtime.

In addition, Leveson in [74] outlines over 60 criteria for requirements completeness of software. These criteria include properties on the transitions of the state machine such as reachability (the ability to reach every state in the system), path robustness and reversibility, all of which can be checked for in the software architecture description. Determining whether or not a software architecture can ensure completeness and determinism of the resulting software system can therefore easily be determined even during the design phase.

4.3.3.2 – Ease of verification and validation

Another quality attribute desirable for the software system of the Exploration Initiative similar to analyzability with respect to safety is ease of verification and validation. This quality attribute refers to the ability of the architecture to support engineers in the construction of software that allows the ease of both verification and validation tasks at every phase of the software lifecycle. As previously mentioned, verification refers to whether or not the system meets its requirements, while validation determines if the system meets the user's requirements. There are several criteria associated with the software architecture that determine whether or not it enhances ease of verification and validation. These criteria include reviewability, isolation of critical functions, simplicity, and testability. Again, many of these criteria come from Leveson's twelve software system safety tasks.

Reviewability → As in analyzability with respect to safety, reviewability is also an important aspect of ease of verification and validation. However, it is important to this quality attribute for slightly different reasons. In order for spacecraft system software to be easy to verify and validate, the architecture must support reviewability because it aids software engineers in both testing and quality assurance activities. Verification and validation activities are performed during each phase in the software engineering lifecycle to help assure the quality of the end software product. By the end of each phase, several software artifacts have been produced. These artifacts are verified and validated with respect to customer requirements and desired quality attributes to help ensure that proper feedback is given to the software developers concerning the quality of their software artifacts.

Reviewability of the software architecture aids in these testing and quality assurance activities because it increases the ability of software engineers to identify inconsistencies between the software artifacts and the customer requirements at earlier stages of the software development lifecycle. For example, Abowd *et. al.* in [2] identify some early warning signs of reviewability issues that stem from the software architecture, such as "top-level architecture components number more than 25." Having more than 25 conceptual architecture component types drastically decreases reviewability because of the complexity of the architecture. Consequently, verification and validation activities will be much more difficult.

Isolation of Critical Functions → Isolating the safety- and mission-critical functions in the software architecture also enhances the software engineers' ability to identify and perform quality assurance activities on the safety- and mission-critical software system components. By isolating the safety- and mission-critical functions into separate components, software engineers can easily identify the portions of the software that should be focused on when performing testing and quality assurance activities. Certain software architectures decompose the necessary software functionality into components and connectors that isolate critical functions, while other software architectures (as described in Chapter 5) do not.

Simplicity → Maintaining simplicity in the software architecture instead of adding unnecessary complexity is also essential to enhancing the ease of verification and validation activities. Software engineers who are battling the complexity of the software architecture are much less likely to accurately perform verification and validation tasks, than if the software architecture remains as simple as possible. Furthermore, there is a tradeoff associated with the design and implementation phase of the software development process that involves functional allocation. Increasing the capability of the software often increases its complexity and vice versa. What is the optimal balance between software capability and complexity?

There are several strategies for maintaining simplicity in a software architecture. These strategies include, but are not limited to, minimizing the number of states, modes and interface types; standardizing (through the use of information content and passing guidelines) interface communications; and minimizing dependencies. First, although many of today's complex systems are designed using a proliferation of states, modes and interfaces, this may introduce unnecessary complexity which hinders software engineers' ability to perform verification and validation activities. As discussed in [75], although mode-rich systems may provide flexibility and enhanced capability, they increase the need for and difficulty with maintaining an accurate mental model of the software process. Inaccurate mental models can lead to mistakes by the software engineers reviewing the software architecture.

Second, standardization of interface communications also aids in maintaining simplicity in a software architecture. Using information exchange standards, communication protocols, and documentation requirements as part of the software architecture provides software engineers with minimal variation between system interfaces, thereby decreasing complexity and increasing enterprise understanding. Ease of verification and validation is enhanced through these interface requirements, as long as those standards are appropriately matched to the software architecture and the characteristics of the problem domain. Finally, minimizing dependencies decreases the complexity of the software architecture [2], thereby increasing the simplicity of the design and the ease with which software engineers perform verification and validation activities.

Testability → Ease of verification and validation is also enhanced when the software architecture aids testers in developing safety-related test plans, test descriptions, test procedures and test case requirements. Different architectural styles provide various levels of support for software testing. As described in [44], how effectively a testing strategy fulfills its roles is dependant on (1) the match between the software architecture and the test strategy, (2) the ability of the test strategy to detect certain fault types, and (3) the interaction of the software architecture, test strategies and fault detection.

For example, pipe and filter architecture styles are most compatible with a thread integration test strategy, in which single threads of control are tested first, and then additional threads for the entire system are integrated. Critical module testing, on the other hand, which prioritizes software modules on a continuum of most critical to least critical, can be more easily performed on software architectures that isolate critical functions into separate components [44]. Furthermore, there is a correlation between test strategies and types of faults detected. For example, critical module testing is an appropriate software testing technique for finding timing faults that cause loss of data or improperly handled interrupts [44].

4.3.3.3 – Sustainability

One of the guiding principles of the entire Exploration Initiative involves the US maintaining a sustainable presence in space. This objective is contingent on a sustainable software system to support the goals of the Exploration Initiative. Sustainability is the ability of a software system to support meeting stakeholder needs in the present, while preserving the ability of the system to support these needs indefinitely as well as any new needs (both planned and unplanned) that may arise. Sustainability encompasses both extensibility (easily allows the addition of new features at a later date) and evolvability (allows the system to achieve ultimate purpose gradually). An approximation of how sustainable a software architecture is can be determined by looking at coupling within the architecture and the amount of technology forecasting that was used as part of the software architecture development process.

Coupling → Coupling refers to the amount of connectivity between modules in a software system. Measurements of coupling are some of the oldest and most well-tested software metrics and provide insight into the ease with which software can be modified. Since the software for the Exploration Initiative must be both extensible and evolvable in order to be sustainable, coupling metrics are able to provide insight into the sustainability of the software. During design, software engineers often contend with a tradeoff between coupling and modularity. In spacecraft software design, for example, there is usually a high degree of coupling between functions because of the limited resources on the spacecraft and the need for subsystems to work closely together to maximize resource allocation and efficiency. On the other hand, there is also a need for software reuse to decrease development costs, which would argue for decoupling components.

As described above, extensibility refers to allowing the addition of new features at a later date, while evolvability refers to achieving system purpose gradually. Insight into these quality attributes is achieved by analyzing the coupling from two different architecture viewpoints: inter-module coupling and intra-module coupling. Inter-module coupling refers to the number of references between modules while intra-module coupling refers to the connectivity within a module [80]. Intra-module coupling is important because when modifying the software system, it is important not to reduce inter-module coupling at the expense of significantly increasing intra-module coupling [80].

<u>Technology Forecasting</u> → The other aspect of sustainability has to do with the software architecture's ability to incorporate new technologies. The lifetime of the Exploration Initiative is upwards of 25 years and the software product line represents a large investment over the course of that project. Consequently, the software architects can help protect this investment through regularly examining new software and hardware technologies and building the architecture to handle them. There are three types of future technology fusion that needs to be taken into consideration when constructing the software architecture: new software onto legacy hardware, new hardware to support old software and new software to work with old software.

Many times it is possible to use technology forecasting to construct an avionics architecture that has hooks for future technologies. Technology forecasting in software engineering refers to exploring new development tools and techniques and new implementation products and strategies, as well as new system technology. Information about these products and technologies comes from end-user feedback, vendor trend reports, and advanced technology laboratories [31]. "Technology forecasting involves activities that attempt to understand what the future will bring to the software product line, in terms of new technologies and trends in the relevant application area(s) or domain(s)" [26]. The responsibility of preliminary technology forecasting with respect to the software product line architecture initially falls on the software engineers; however, for sustainability reasons, extended technology forecasting throughout the remainder of the Exploration Initiative must be under the perview of the System Engineering and Integration organization. This group should be responsible for continually keeping abreast of technology developments in order to make informed technology decisions regarding the core asset base.

4.3.3.4 – Affordability

Another important objective of the Exploration Initiative is that the goals are achieved within the available budget. Every thorough software architecture evaluation should consider the cost consequences of the proposed design. In this example, the US government has made it clear that little to no additional funds will be allocated to NASA from what they already receive to support the Exploration Initiative. Therefore, NASA must make the mission goals possible under the current budget. From a software perspective, affordability presents a large challenge, because the cost of the software will probably be a large portion of the total cost of the Initiative.

Affordability from a software standpoint refers to the ability of the software development enterprise to construct software and perform downstream software engineering activities such that the amount of resources required to build the software does not exceed the available resources. Through comparing different software architectures, engineers can determine if there are any features in the architectures that may be sources of undue cost or risk during the various lifecycle phases. The cost model described earlier in this chapter can be used to evaluate the software architectures with respect to cost.

4.3.3.5 – Buildability

Another important quality attribute, especially from the perspective of the software engineering stakeholders, is buildability, or the ability of software engineers to construct software from the software architecture. The software architecture should be intellectually manageable by the software engineers given the available resources. Buildability is comprised of intellectual manageability, availability and appropriateness of programming languages to support the architecture, hardware availability, and coupling.

<u>Intellectual Manageability</u> → The design and implementation phases of the software lifecycle are greatly influenced by the description of the software architecture. When selecting the software architecture for a specific project, there are often important issues concerning buildability and intellectual manageability. Can the system be built using the staff, budget, legacy software, and time available? Is building the proposed system intellectually manageable given the resources available for the project? Will the selected architecture make the system easier or harder to build? Does the software architecture style accurately match the engineers' mental model of the spacecraft system?

The intellectual manageability of a software architecture is closely related to the Technology Readiness Level (TRL) as described in Chapter 3. Architectural styles that have been widely used for many years, such as "Layered" and "Pipes and Filters" have well-known characteristics and properties. For example, a layered style, first introduced by Dijkstra in [35], has been frequently used as the architectural basis for operating systems for decades. Layered architectures have known advantages and disadvantages (advantages include hierarchical abstraction, while disadvantages include coupling between high- and low-level functions) and operating system classes at the university level are often taught using this style. In other words, the layered architecture style has a very high TRL and is an intellectually manageable design for most software engineers. Conversely, the TRL for agent-based architectures is quite low, especially in the spacecraft domain, where there has only been one experiment with onboard agent-based software [64].

Programming Languages → Different programming languages are more or less appropriate for implementing different software architecture styles. For example, there are some programming languages that are appropriate for software built using Object-Oriented Analysis and Design (OOAD), such as Java and C++. These languages were built to especially support object-oriented programming. However, these programming languages are less appropriate for real-time control systems, due to built-in services such as automatic garbage collection [114]. In those systems, Ada is a more appropriate programming language selection, because of the support it provides for timing constructs. The availability and maturity of appropriate programming languages for implementing a particular software architecture is important to ensuring the buildability of the system.

Hardware Availability → Computations and memory are a limited resource onboard spacecraft. Unlike computers on Earth, computers in space have to be radiation hardened, which is an extremely expensive process. Consequently, spacecraft computers lag several generations behind commercial desktop or laptop computers [132]. Software engineers must be able to implement the chosen architecture given the limited computational and memory capacities of the spacecraft computer.

Coupling → Inter-module and intra-module coupling measurements are also indicators of buildability of software architectures. There are several advantages to reducing connectivity in terms of buildability. Low connectivity allows for independent development of modules, because decisions are made locally and do not interfere with the correctness of other modules. Furthermore, correctness proofs are easier to devise, because they are on a much smaller scale. Decoupling also increases the comprehensibility of the various modules, because the module can be understood independently of the environment it is used, which is especially important in a software product line approach.

4.3.3.6 – Ability to meet real-time constraints

Real-time software systems are those systems that are subject to constraints in time; these systems fail if the software does not complete its intended purpose in the time-period after an event and before the deadline relative to the event. This quality attribute refers to the ability of the software engineers to ensure that deadlines will be met and the software will operate within the available resources through software architecture constructs. The criteria identified for determining whether or not a software architecture will provide a foundation for achieving real-time constraints are predictability and the ability to respond to the asynchronous environment of space, or concurrency.

Predictability → The software architecture must allow for a high degree of determinism or confidence that the software will operate as expected. Many embedded controllers are real-time and safety-critical, and it is often imperative that the software behavior be highly predictable. There are several strategies for enhancing the predictability of real-time, safety-critical systems such as those used in the Exploration Initiative. Soft and hard failure modes should be eliminated for all hazard-reducing outputs, while hazard-increasing outputs should have both soft and hard failure modes; there should be multiple paths through the software for state changes that increase safety and multiple triggers required to transition from a safe state to a hazardous state; finally, there must be no paths to unplanned hazardous states, which can be proven using certain architectures, such as functional decomposition [74].

Ability to respond to the asynchronous space environment → Spacecraft are real-time systems that operate in an asynchronous environment. Therefore, the software must be able to deal with unexpected events while ensuring that hard, real-time deadlines and timing constraints are still met. In addition, multiple, external signals sometimes arrive simultaneously and need to be handled in parallel. The ability to deal with these types of situations is known as concurrency. For deep space missions, autonomy is needed to deal with the large communication delays that can hinder a spacecraft waiting for ground control commands from accomplishing mission objectives. Although autonomy is traditionally associated with AI technologies, autonomy can also refer to any software that allows a system to mitigate faults without the interference of ground controllers. Architectures that support this type of autonomy are critical to the success of the Exploration Initiative.

Furthermore, the software should provide the ability to function in a degraded state of performance. In other words, instead of entering a safe mode, the software should be able to perform Fault Detection, Isolation and Recovery (FDIR) independently of ground control. This type of onboard FDIR requires new software architectures that incorporate FDIR software development and integration into the rest of the system software from the beginning of the software lifecycle. In other words, the FDIR software should not be a separate entity developed independently of the other subsystem software; it should be developed as part of the subsystem software. This approach to FDIR software development is described in [100].

4.3.3.7 – "Monitor"-ability

One of the quality attributes desirable for the Exploration Initiative that is specific to the spacecraft software engineering domain is the "Monitor"-ability of the software. This quality attribute refers to the characteristic of the software that allows operators to gain insight into the software's operations during runtime. The operators must be able to monitor the performance and telemetry of the various subsystems and devices on board the spacecraft at all times, especially during safety- or mission-critical operations. Traditionally, software engineering has promoted information hiding and encapsulation, but state visibility is necessary for spacecraft software if fault diagnosis and recovery is to be performed: onboard and ground-based diagnosis engines need complete telemetry and state information to determine the nature and cause of the fault as well as possible recovery scenarios. "Monitor"-ability is primarily a measurement of the degree of encapsulation and information hiding used in the software architecture. This software quality attribute has been extensively researched by the State Analysis and MDS team at JPL; the idea of elevating state information to gain insight into runtime software operations is described in Section 2.2 and [40], [61], [62].

Encapsulation and Information Hiding → Encapsulation refers to the software design strategy of enclosing programming elements inside a larger, more abstract entity. In other words, encapsulation refers to the bundling of data with the methods that operate on that data. Information hiding, although often confused with encapsulation, is a subtly different design tactic. Information hiding refers to the design strategy of hiding design decisions in a program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed. In other words, employing information hiding techniques isolates clients from requiring intimate knowledge of the design to use a module, and from the effects of changing those decisions. In summary, information hiding conceals how a module implements its functionality, while encapsulation ensures that the module only interacts with other modules through its interface.

Encapsulation and information hiding are important to measuring the "monitor"-ability of a software architecture, because they provide an indication of what type of information is accessible to operators and controllers at runtime. As described in [39], architecture styles that encourage encapsulation and information hiding actually complicate the design of high-risk control systems. Some information should not be encapsulated in spacecraft control software because no single subsystem has full responsibility for the variable's value. Furthermore, operators and controllers may need to know the value of the variable during safety- or mission-critical flight phases or when a fault occurs. Information hiding may prevent the operators from having that vital knowledge. Consequently, the software architecture must take into consideration encapsulation and information hiding and ensure that it appropriately employed for high-risk control systems.

4.3.4 – Step 4: Software Architecture Viewpoints and ADLs

Finally, viewpoints need to be identified in order to capture the proper information necessary for analyzing the architecture with respect to the quality attributes defined in the previous step. A viewpoint is a specification of the conventions for constructing and using a view. It is a pattern or template from which to develop individual views of a software architecture by establishing the purposes and audience for a view and the techniques for its creation and analysis. Viewpoint definitions should include (as specified in IEEE standard 1471-2000) [60]:

- Stakeholders addressed by the viewpoint
- Concerns to be addressed by the viewpoint
- The language, modeling techniques, analytical methods to be used in constructing a view based upon the viewpoint
- The source

Views (representation of a whole system from the perspective of a related set of concerns) of a particular instance of an architecture type can then be defined and should include the following information:

- Identifier and other introductory information
- Representation of the system constructed with the languages, methods and modeling or analytical techniques of the associated viewpoint
- Configuration information

Furthermore, each architecture description should include the rationale behind the selection of the viewpoint, to help maintain sustainability throughout the Exploration Initiative.

One of the most well-known set of software architecture viewpoints was described by Krutchen in [71] and is called the 4+1 view model. It was developed to support object-oriented analysis and design and is therefore based on UML diagrams and scenarios. Figure 4-10 presents a modified 4+1 viewpoint model for spacecraft software that is architecture independent and focuses on data and information flow. A description of each of the viewpoints used in this study can be found in Appendix C.

In addition to identifying necessary viewpoints for architecture definition, it is also necessary to provide a formal architecture description language (ADL) for creating these definitions. Current, popular ADLs include UML [19], UniCon [137], and many others described in [87]. In addition, a visual architecture description language that still has a formal basis (thereby allowing mathematical analysis) is preferred because it helps to limit the semantic distance between the user of the model and the model itself. Furthermore, an ADL which does not limit its context to a particular domain or design style is also preferred, because we wish to be able to accurately model various architecture styles in a design-independent framework. The ADL chosen, because of its clear adherence to these guidelines, is ACME, which is an Architecture Description

Interchange Language developed by Garlan [52]. ACME was developed as part of the ABLE (Architecture-Based Languages and Environments) Project to provide a tool in which software engineers can specify their own architecture styles as well as their representations and then construct architectures using these self-defined styles. In other words, ACME is a design-independent framework for describing and developing software architecture families and software architectures themselves [25].

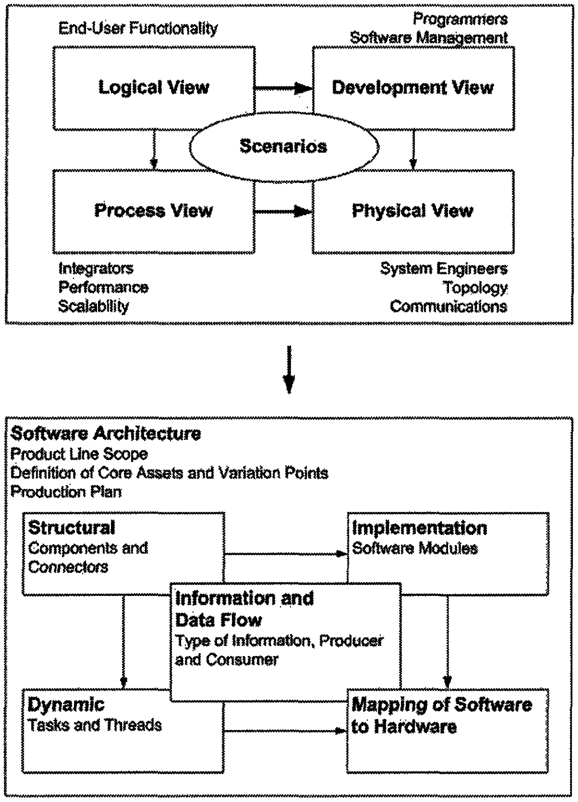


Figure 4-10. Modified 4 + 1 Viewpoint Model for Spacecraft Software

ACME will be used to support architecture evaluation for the Exploration Initiative in the following manner. First, architecture style, or family, definitions will be written using the ACME tool to describe the four architectures chosen for this study: traditional spacecraft software decomposition, functional decomposition, state analysis and net-centric designs. Next, a product line architecture (the architecture from which all vehicle and facility software will be derived) is described for the Exploration Initiative based on these style definitions. It is important to note that there may be multiple product lines that comprise the initiative based on the variety of destinations, vehicles and facilities. Finally, individual software architectures can be constructed for the various components (CEV, TMI, etc.) that comprise the Exploration Initiative system architecture. After completing the conceptual architecture descriptions, trades between quality attribute metrics and/or requirements need to be identified and a qualitative analysis of the product-line information should be performed by experts in the field. It is important not to discount expert opinion and analysis; this information should be combined with the mathematical analysis to perform the down-select step.

4.5 – Summary

Multi-Attribute Software Architecture Trade Analysis consists of software architecting activities throughout every phase of the software lifecycle. This chapter described the steps that comprise MASATA during each the concept generation and requirements portions of the lifecycle. During the concept generation and feasibility study phase, software engineers focus on identifying stakeholders and the quality attributes they desire from the system. They also determine the impact various system architectures have on the cost and risk associated with creating a software system to support that architecture. During the requirements specification phase, the software architecting process begins and engineers elaborate upon the quality attributes identified in the previous step and begin identifying conceptual architecture styles that may be appropriate given the characteristics of the application domain. The results of performing the first two phases of the MASATA process on the Exploration Initiative were also presented. Stakeholders, cost model results, architecture style options and variation points for the vehicles and facilities of the Exploration Initiative were presented. The next chapter provides the necessary MASATA steps that comprise the design portion of the software development lifecycle as well as the results of performing these steps on a system architecture created to support the goals and requirements of the Exploration Initiative.

Chapter 5:

Multi-Attribute Software Architecture Trade Analysis – Design

The previous chapter discussed the first two phases of the MASATA process: concept generation and feasibility study and requirements. This chapter presents the remaining steps of MASATA that take place during the design portion of the software lifecycle. The results of the preliminary design phase as applied to the Exploration Initiative are also presented.

5.1 – Design

After completing the steps in the software architecting process for concept generation and requirements, the software engineering efforts can be geared toward design. Traditional software design is comprised of preliminary and detailed design phases and focuses on the development of conceptual and realizational architectures respectively. Conceptual architecture descriptions involve the traditional box and line diagrams customarily associated with the term architecture. These are fairly high-level descriptions of the architecture and can have many lower-level (or realizational architecture) interpretations. Realizational architecture descriptions are at a much lower-level and are directly visible at the coding level. During preliminary design, the following steps are taken to define and evaluate the software architecture options:

1. Generate conceptual architecture options
2. Identify methods for evaluating the software architecture (checklists, metrics, etc.) with respect to the quality attribute criteria
3. Identify any trades between quality attribute criteria/metrics and/or requirements

4. Perform qualitative and quantitative analyses of product-line information and preliminary conceptual architecture options
5. Down select based on qualitative and quantitative evaluation results

Once the architecture is defined using an ADL, it is in an analyzable form that can be evaluated with respect to the metrics selected to define the various quality attributes desired by stakeholders. Some of these criteria may be quantitative, but most are qualitative. Based on the results of both the qualitative and quantitative analyses, a down-selection of the software architecture trade space is performed. At this point, we want to select possibly two or three options to take into the detailed design phase.

During the detailed design phase, realizational architectures (lower-level designs) are generated and evaluated based on the conceptual architectures identified in the previous phase. The detailed design phase includes the following steps:

1. Generate preliminary designs from each conceptual software architecture option in the down-selected group using the views and architectural description languages identified in the previous step
2. Perform qualitative and quantitative analyses of the models to determine their adherence to requirements and desirable quality attributes
3. Select conceptual software architecture
4. Generate realizational software architecture based on the conceptual architecture chosen
5. Ensure that the final design adheres to the architecture selected at the end of the preliminary design phase

Again, these architectures are modeled using an architecture description language, a form in which they can be analyzed more accurately both qualitatively and quantitatively. The architecture is again analyzed and finally a conceptual architecture is chosen for the product line. All detailed designs (realizational architectures) generated from the chosen conceptual architecture must also be analyzed to make sure that they adhere to the constraints and intent of the selected conceptual architecture.

5.1.1 – Preliminary Design

During preliminary design, the following steps are taken to define and evaluate the software architecture options:

1. Generate conceptual architecture options
2. Identify methods for evaluating the software architecture (checklists, metrics, etc.) with respect to the quality attribute criteria
3. Identify any trades between quality attribute criteria/metrics and/or requirements
4. Perform qualitative and quantitative analyses of product line information and preliminary conceptual architecture options
5. Down select based on qualitative and quantitative evaluation results

5.1.1.1 – Step 1: Generate Conceptual Architectures

During the preliminary design phase, software engineers construct conceptual architectures (represented by box and line diagrams) based on the architecture styles identified during the requirements phase and their knowledge of the system at hand. This section presents these conceptual architectures and the means by which they are evaluated with respect to the quality attribute criteria outlined in the previous chapter. Based on the results of evaluating these conceptual architectures, the software engineers then decide which conceptual architectures are appropriate for taking into the detailed design phase.

Goal-based Conceptual Architecture: The goal-based conceptual architecture was described in Section 2.2 and depicted in Figure 2-13.

Traditional Subsystem-Based Decomposition Conceptual Architecture: Figure 5-1 depicts the typical software architecture for today's spacecraft systems. The spacecraft is decomposed with respect to its subsystems, which are grouped according to the functions they provide for the spacecraft. Often confused with functional decomposition, subsystem-based decomposition is a combination of both structural and functional decompositions. First, the system level consists of the command and data handling software (CDH), which controls the actions of all the subsystems, provides some of the "operating system"-type software, and interfaces with the central Fault Detection, Isolation and Recovery (FDIR) subsystem. The CDH software interfaces with the various subsystems that comprise a spacecraft, including Thermal Control, Attitude Determination and Control, Guidance, Navigation & Control and Communications.

These subsystem software modules report their status to the CDH software, and, in turn, are controlled by that software. In the same way, the sub-subsystem and component software reports status to their parent subsystem and is controlled by that subsystem.

Fault Detection, Isolation and Recovery module is separate from the other subsystems and the CDH software. The Central FDIR interfaces directly with the CDH software and with the subsystems through FDIR components in each subsystem. Unpredictable events outside nominal variations are dealt with by this high-level fault protection software. This software may be inadequate if time or resources are constrained and recovery actions interfere with satisfying mission objectives. If the FDIR software is insufficient to handle a fault, the spacecraft enters a safe mode in which all systems are shut down except for those needed to communicate with Earth. The spacecraft then waits for instructions (a specialized command sequence for dealing with the fault) from the ground controller.

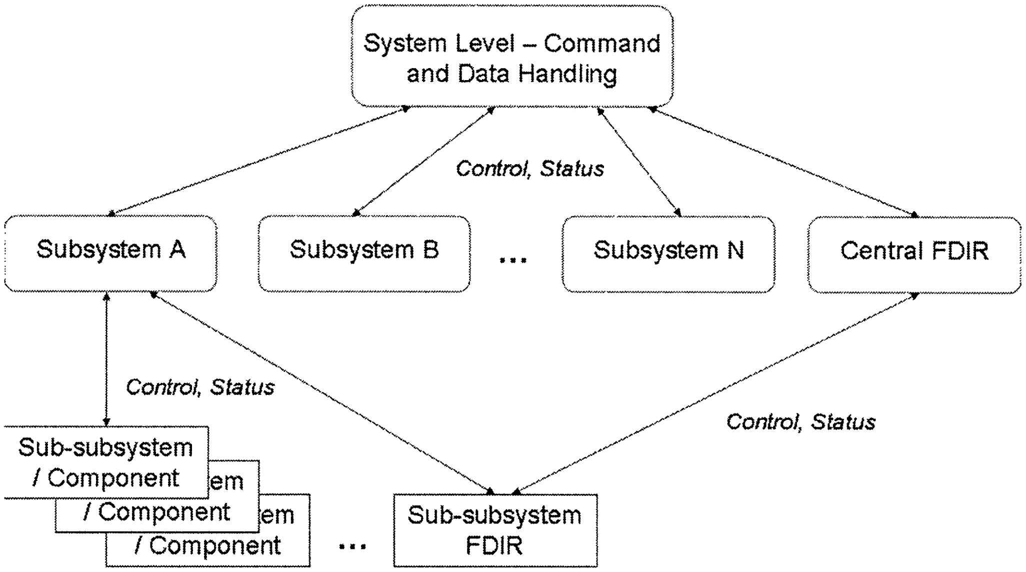


Figure 5-1. Traditional Subsystem-Based Decomposition Conceptual Architecture

The decomposition of the spacecraft into its composite subsystems represents a type of structural decomposition. These subsystems are then decomposed into the functions that make up that subsystem. As shown in Figure 5-2, the Thermal Control Subsystem, for example, is comprised of a variety of functions all related to maintaining thermal characteristics of the spacecraft such as heat sensing, thermal balancing and caution/warning generation.

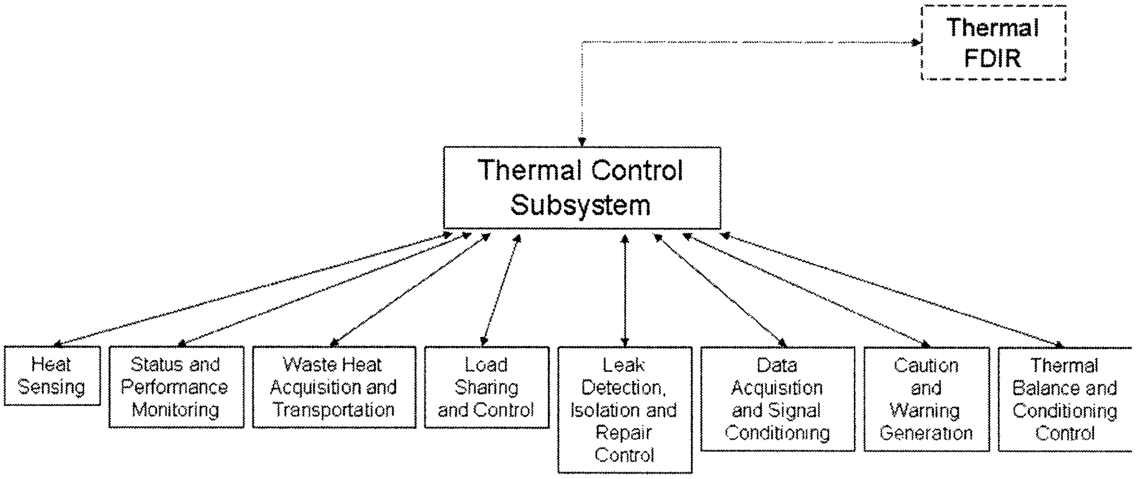

Thermal FDIR
Thermal Control Subsystem
Heat Sensing
Status and Performance Monitoring
Waste Heat Acquisition and Transportation
Load Sharing and Control
Leak Detection, Isolation and Repair Control
Data Acquisition and Signal Conditioning
Caution and Warning Generation
Thermal Balance and Conditioning Control

Figure 5-2. Traditional Subsystem-Based Decomposition Example

In addition, in traditional, subsystem-based spacecraft decomposition, control lies in a single thread comprised of time-stamped commands to the CDH computer: a time or an action triggers another action in software. Consequently, the spacecraft must adhere closely to its predefined operational model to assure that mission objectives are achieved. These sequences are extremely specific to each spacecraft and mission and are often written on-the-fly.

Net-Centric Conceptual Architecture: The layered and implicit invocation architecture styles can be used in combination to create a net-centric software architecture for the Exploration Initiative, where the term net-centric refers to distributed software that takes advantage of available resources despite where these resources reside. Prior to configuration, spacecraft modules exist independently of one another. Each module has software built upon and from the software product-line core assets. The module's internal software architecture depends upon the module's type, i.e. subsystem or avionics. These architectures are depicted in Figure 5-3 within the subsystem and avionics boxes.

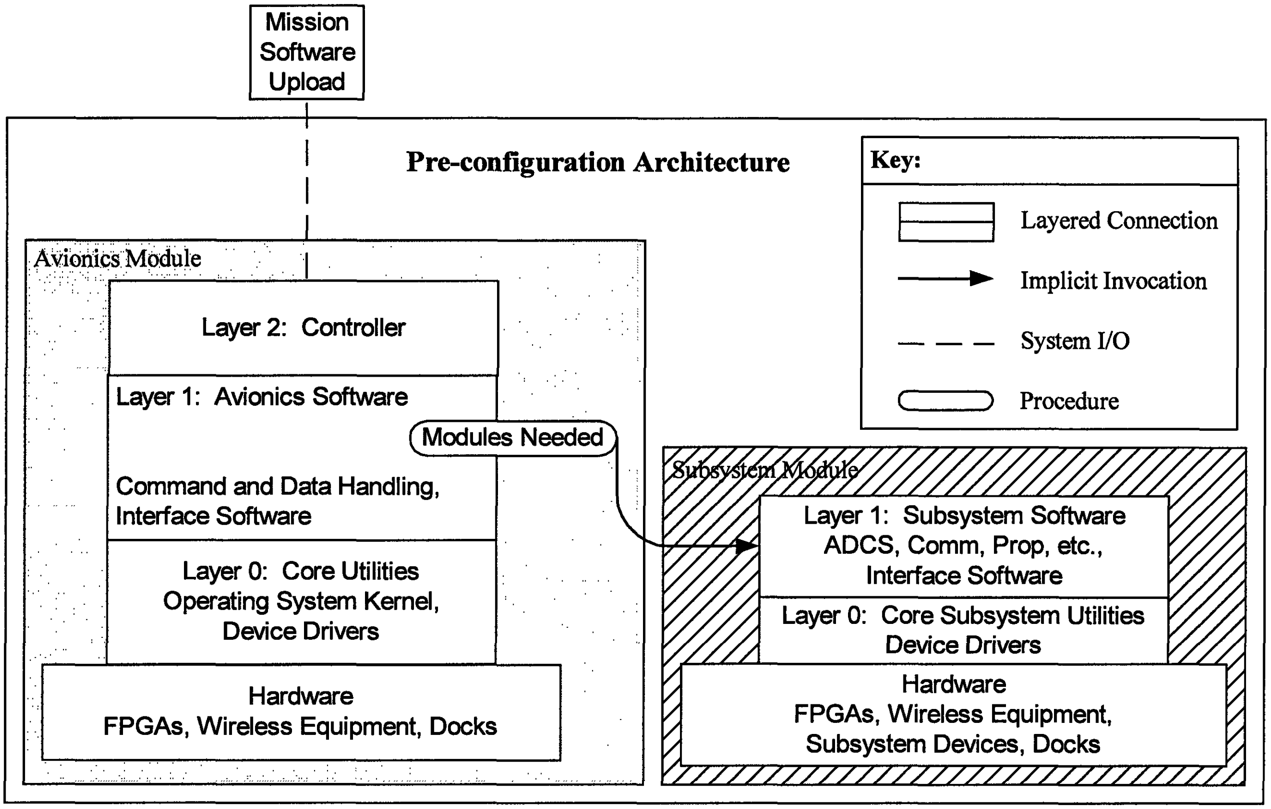

Mission Software Upload
Pre-configuration Architecture
Key:
Layered Connection
Implicit Invocation
System I/O
Procedure
Avionics Module
Layer 2: Controller
Layer 1: Avionics Software
Modules Needed
Command and Data Handling, Interface Software
Layer 0: Core Utilities Operating System Kernel, Device Drivers
Hardware FPGAs, Wireless Equipment, Docks
Subsystem Module
Layer 1: Subsystem Software ADCS, Comm, Prop, etc., Interface Software
Layer 0: Core Subsystem Utilities Device Drivers
Hardware FPGAs, Wireless Equipment, Subsystem Devices, Docks

Figure 5-3. Pre-Configuration Implicit Invocation-Based Net-Centric Architecture

As seen in the subsystem module box in Figure 5-3, the subsystem module software follows a layered architecture style built on top of subsystem hardware, which includes FPGAs, wireless equipment, devices particular to that subsystem and at least one docking device. At the lowest layer in the software architecture reside the core subsystem utilities. These include the device drivers for the particular pieces of hardware used by that subsystem as well as for the docking device(s) and the wireless communication protocols for receiving and transmitting information between modules. The second layer in the software architecture contains the subsystem software; it uses the facilities provided by layer 0 to provide the module with typical subsystem functionality. For example, a TMI stage contains hardware for providing GN&C functionality; consequently, the TMI stage contains software that converts gyroscope and accelerometer data into the spacecraft's attitude and uses this information in conjunction with a desired destination to operate the propulsion equipment. Another example involves battery modules that contain software for charging and discharging power and monitoring the temperature of the spacecraft, which all stages must contain. Consequently, if the battery in one vehicle fails, it can use the resources of other vehicles to perform battery functions. Each module also contains the interface software used in the implicit invocation architecture style that governs the interactions between the various modules.

As seen in the avionics module box in Figure 5-3, the avionics module software is slightly different than the individual subsystem module software. The designated avionics module controls the post-configuration spacecraft stack. Although it also follows a layered architecture style built on top of command and data handling hardware, the software functionality contained in each layer is different from the subsystem modules. The command and data handling hardware includes the spacecraft processor (probably just a more powerful FPGA), wireless equipment and docking devices. At the lowest layer in the software architecture reside the core system utilities. These include the services delivered by the operating system kernel (such as scheduling, memory reading and writing, etc.), device drivers for the particular pieces of hardware used for command and data handling as well as the for the docking device(s) and the wireless communication protocols for receiving and transmitting information between modules.

The second layer in the software architecture contains the command and data handling, avionics software; it uses the facilities provided by layer 0 to provide the module with typical CD&H functionality. There are two important pieces of software that reside at this level besides the typical CD&H functions: the interface software for use in the implicit invocation architecture style that governs the interactions between the various modules and configuration evaluation software for determining whether or not the configuration meets the needs of the mission software. Finally, the topmost layer (layer 2) of the avionics software architecture contains the control software that contains a set of mission objectives. The control software is (presumably) uploaded to a particular avionics module from a ground station.

As previously mentioned, all modules contain software interfaces. In an implicit invocation architecture, these interfaces contain a collection of procedures and events. The interface of the avionics module broadcasts several events, one of which is important during the pre-configuration phase. This event is the "Modules-Needed" event. The avionics module broadcasts this event whenever a new control program is uploaded from the ground. Depending on the types of modules needed by the control program, different instances of the modules-needed event will be broadcast. For example, if only propulsion and attitude determination are needed, the "Modules-Needed" event broadcast will contain signatures unique to propulsion and attitude determination subsystem modules. The control program can also call other avionics modules if additional computational power is needed. Each module also contains procedures that respond to the broadcast events. The procedures registered to the "Modules-Needed" event are "Docking" procedures. Once it has been established that a module is needed by an avionics module, the procedures associated with docking begin so that the various modules can attain the avionics module's desired configuration.

Once a controller is uploaded to an avionics module the process of configuration begins. For software the sequence of events leading up to configuration proceed as follows:

1. The controller relays to the command and data handling software a message that the following modules are needed for the particular configuration detailed in the mission software.
2. This "Modules Needed" event is broadcast (layer 1 service) from the avionics module through the wireless network (layer 0 service).
3. This event is transmitted to the various modules within the vicinity. The interfaces (layer 0 service) may or may not be registered with this event. In the modules that are registered with the event, a flag is raised and the high-level event handling software takes control and autonomous docking begins. Modules that are not registered with the event remain inactive.
4. Autonomous docking is performed to create the specified configuration.

Once all of the modules have been docked, the spacecraft transitions into post-configuration phase. The post-configuration conceptual architecture is depicted in Figure 5-4. Although the layered portion of the architecture remains unchanged, there are several aspects of the overall architecture that do not manifest themselves until after a configuration has been reached. These aspects are discussed below.

Because multiple avionics nodes can be connected in a configuration to provide extra computational power to the command and data handling activities, it is important to designate a single control node after reconfiguration.

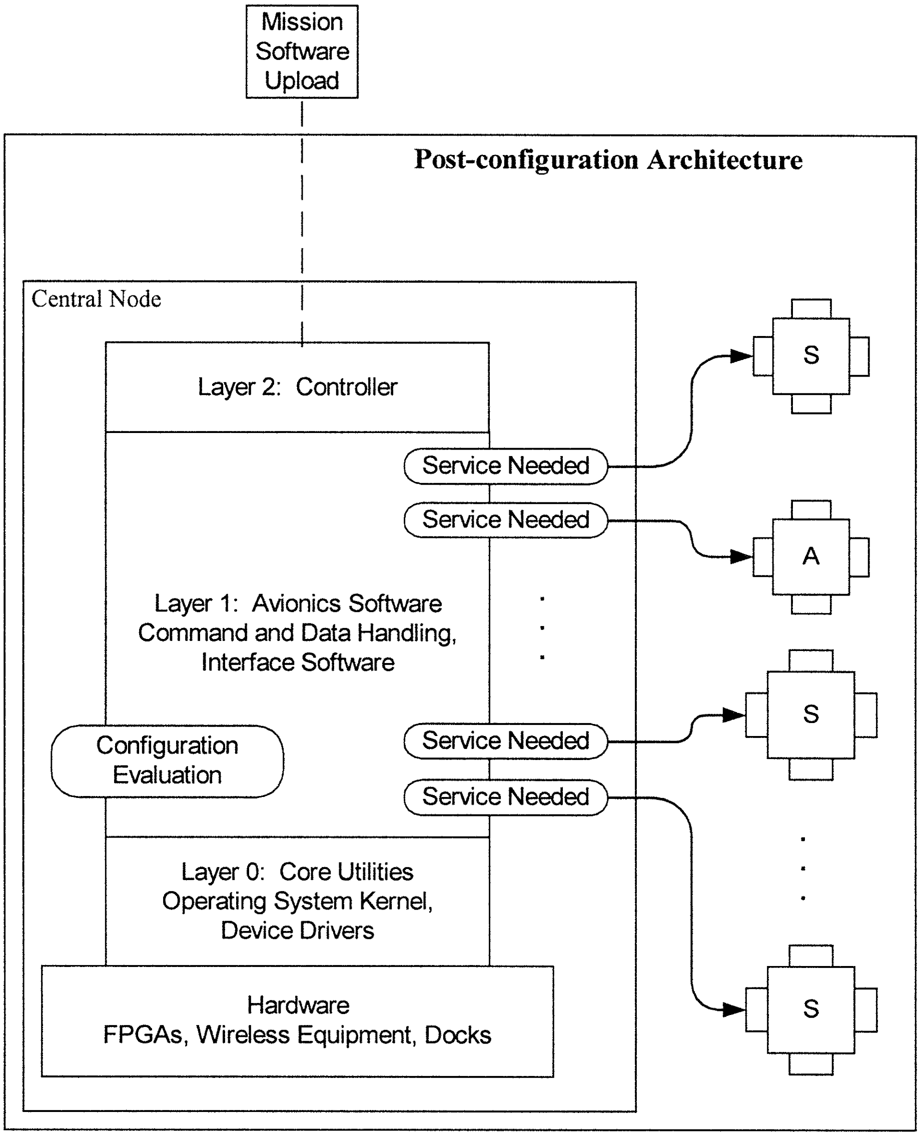

Mission Software Upload
Post-configuration Architecture
Central Node
Layer 2: Controller
Service Needed
Service Needed
Layer 1: Avionics Software Command and Data Handling, Interface Software
Configuration Evaluation
Service Needed
Service Needed
Layer 0: Core Utilities Operating System Kernel, Device Drivers
Hardware FPGAs, Wireless Equipment, Docks
S
A
S
S

Figure 5-4. Post-Configuration Implicit Invocation-Based Net Centric Architecture

The first activity performed after all "Modules Needed" have been docked with the central node is "Configuration Evaluation." Configuration evaluation is a piece of software that resides on the central node and is part of the command and data-handling package. This software procedure tests the integrity of the reconfigured system by performing a series of checks as to whether or not the new vehicle has assembled properly. Based on the results of these checks, the configuration evaluation software then makes a go/no-go decision as to whether or not to enable the operation of the spacecraft. The checks include (1) confirmation from the central node hardware drivers that a successful docking has indeed taken place and (2) confirmation from the assembled subsystem and avionics module hardware drivers that a successful docking has been achieved and (3) a sample event broadcast that makes a small alteration in the attitude of the new spacecraft, (4) a sample event broadcast that measures the new attitude of the spacecraft, and (5) a comparison of the predicted attitude with the actual attitude. If the docking handshake has been confirmed by each node in the system and the difference in attitude caused by the small alteration event is within a predefined tolerance, then the configuration evaluation software notifies the controller to begin nominal spacecraft operation. If not, diagnosis software takes control of the central node and the process of fault diagnosis and recovery takes place.

Once the spacecraft is operating nominally, several services may be needed from the various subsystem and avionics nodes to achieve the mission objectives as determined by the software controlling the central node. The central node broadcasts "Service-Needed" events through the wireless communication device. Each subsystem and avionics node has particular procedures registered with these events. These procedures reside on the second layer of their architectures and include typical subsystem utilities such as propulsion, attitude determination, power, communication, CD&H, etc. The central node achieves its mission objectives by announcing a series of "Service-Needed" events to the spacecraft system. The procedures in the subsystem and avionics module software that are registered with these events execute, thereby achieving the objectives of the spacecraft controller. For example, if the spacecraft needs to perform a Hohmann transfer to achieve a higher orbit as prescribed by the control software, the central node might announce a series of events in the following order:

1. Determine Orbit
2. Calculate Burn
3. Activate Thruster X for Y Seconds
4. Determine Orbit
5. Confirm Transfer

The attitude determination module would respond to the "Determine Orbit" event and respond by delivering the attitude of the spacecraft to the central node. As described in the previous section on architecture styles, implicit invocation can be applied to the module generating the event. When a "Calculate Burn" or "Determine Orbit" event is announced, the CD&H utilities provided by the central node are required and a procedure associated with this event is activated. Finally, the propulsion module will respond to the "Activate Thruster X for Y Seconds" event, thereby providing actuation to the spacecraft.

<u>Functional Decomposition Conceptual Architecture</u>: Figure 5-5, the functional decomposition conceptual software architecture diagram, is one example of an architecture that can be derived from the product line vehicle and facility organization diagram in Figure 4-9. Figure 5-5 illustrates how a layered architecture can be used to provide the overall organization of the software for the product line, while Figure 5-6 shows an example of the software architecture within each layer, which is based on functional decomposition.

The software for the product line is built upon a common avionics hardware platform. Having a common avionics hardware architecture aids in creating an effective software product line architecture because the hardware interfaces are then also a part of the core assets and can be used and upgraded across multiple vehicles and facilities. Furthermore, a common operating system, shown in Figure 5-5 as Systems and Information Management, can also be utilized across the entire product line.

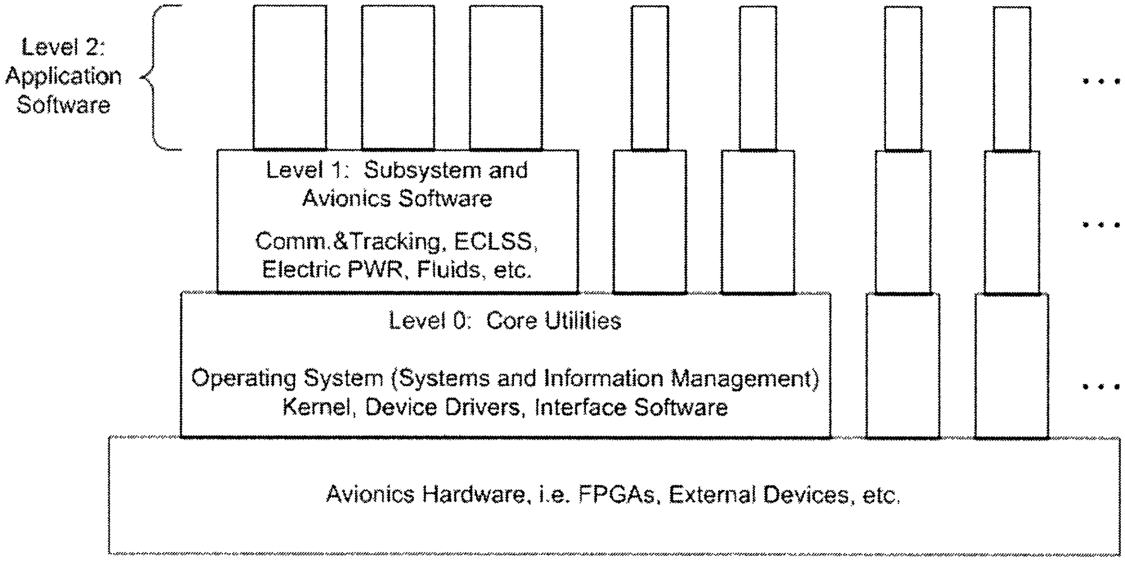

Level 2: Application Software
Level 1: Subsystem and Avionics Software
Comm.&Tracking, ECLSS, Electric PWR, Fluids, etc.
Level 0: Core Utilities
Operating System (Systems and Information Management)
Kernel, Device Drivers, Interface Software
Avionics Hardware, i.e. FPGAs, External Devices, etc.
. . .
. . .
. . .

Figure 5-5. Functional Decomposition Conceptual Architecture

Level 0 of the layered software architecture includes the core utilities of the software, which include both operating system services and device drivers. Because not every grouping of vehicles and/or facilities have the exact same devices, software components can also be directly added onto the hardware separate from the Level 0 software platform used as a core asset across the entire product family. These separate software assets are denoted by the empty rectangles next to the Level 0 core utility software asset.

Level 1 of the layered software architecture encompasses the software traditionally captured in subsystem modules, i.e. ECLSS, Power System, ACS, etc. software. Like Level 0, some of this software can be used across all vehicles and facilities and shown in the Figure 4-2 product line organization chart. However, there is more software at this level that needs to be added for particular vehicle and facility uses. This software is designated by the rectangles built upon both the Level 0 core asset and the Level 0 additional software rectangles.

The architecture of the software at this level varies from a traditional approach to spacecraft software organization. Traditionally, spacecraft software is organized along a structural decomposition of the system into various subsystems and their components. In this architecture, a purely functional decomposition is used to organize the subsystem software. Consequently, instead of having subsystem software modules such as attitude determination and control or environmental control, the software is organized along the functions that need to be accomplished, such as the function illustrated in Figure 5-6, "Set Temperature."

This type of software organization takes into consideration the physics of the spacecraft, in which various aspects of the subsystems interact with other subsystems. As outlined in [39], taking these interactions into consideration from the beginning helps to ensure that fault protection is not an add-on feature but an integral part of the spacecraft software. Determining the current temperature involves checking the temperature of various assemblies throughout the spacecraft. Next, the actuation values to the various subsystem components that generate heat or cool the spacecraft can be determined. For example, the interaction between the thermal and power subsystems is captured because the read actual temperature and determine temperature actuation value functions takes the battery itself into consideration from the beginning, not just in a fault protection mode.

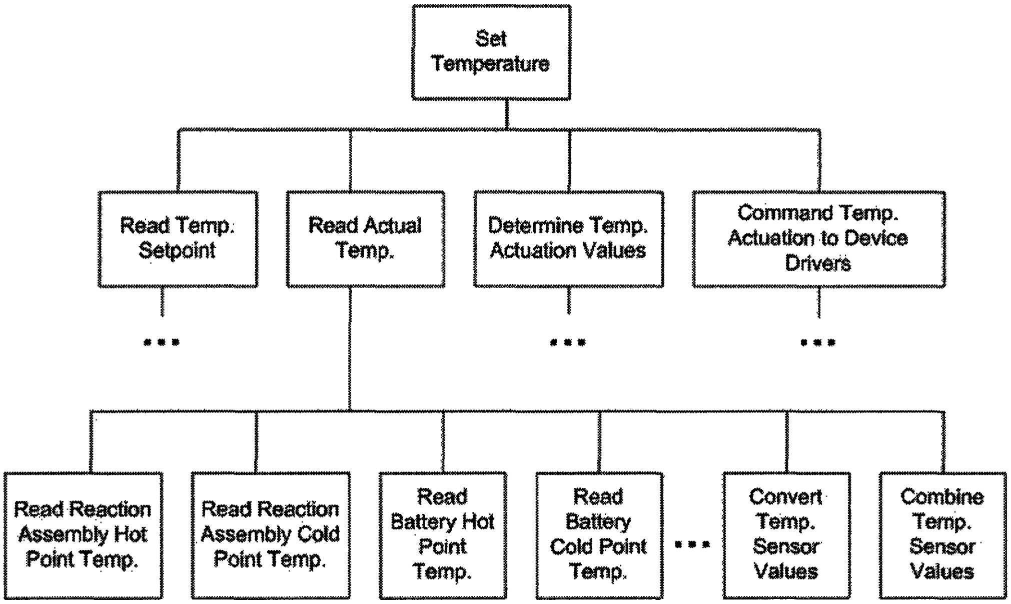

Set Temperature
Read Temp. Setpoint
Read Actual Temp.
Determine Temp. Actuation Values
Command Temp. Actuation to Device Drivers
...
...
...
Read Reaction Assembly Hot Point Temp.
Read Reaction Assembly Cold Point Temp.
Read Battery Hot Point Temp.
Read Battery Cold Point Temp.
...
Convert Temp. Sensor Values
Combine Temp. Sensor Values

Figure 5-6. Functional Decomposition Application Level

The final level of software in the layered software architecture is Layer 2, or the application software. This level encompasses software specific to unique control laws required to achieve mission goals or software specific to an individual vehicle or facility. For example, certain vehicles will need to perform autonomous rendezvous and docking at the Moon or Mars. However, these functions will not need to be immediately available. Consequently, an autonomous rendezvous and docking function can be added at the Level 2 software application level as it is needed in the timeline of the Exploration Initiative.

5.1.1.2 – Step 2: Quality Attribute Measurement Techniques

Now that the conceptual architecture options have been described, they can be evaluated both qualitatively and quantitatively with respect to the quality attribute criteria outlined in the previous chapter. Below is a list of the quality attribute criteria, how they will be measured in the architectures, and any trade-offs that exist between the criteria. Appendix C contains a descriptive listing of the viewpoints used in modeling the software architecture options. Part of these descriptions includes the concerns addressed by the particular viewpoint, in this case which criteria are evaluated based on a certain software architecture view. It is also important to note that some of these measurement techniques cannot be utilized at the conceptual architecture level, because of the lack of detail. However, they can be applied once a realizational architecture is constructed.

Reviewability → Measuring the ease with which stakeholders can review a software architecture is largely qualitative; therefore, based on expert opinion and stakeholder assessment, the reviewability of a software architecture can be rated on a scale of High, Medium and Low. The application of model-based development methods may greatly increase the reviewability of the software architecture if the modeling techniques and languages employed complement the spacecraft engineers' mental model of the system. Consequently, if there is a model-based development technique that can be used to support a particular architecture, and the language and tools were built with reviewability and usability in mind, reviewability increases. Finally, the number of different components and connectors has been proposed as an indication of reviewability especially with respect to ease of verification and validation. It is hypothesized that reviewability is enhanced if the number of conceptual architecture types is below 25.

Traceability → As outlined in the previous chapter, traceability is an important aspect of analyzability with respect to safety. Traceability is mostly the result of documentation and software architecture description practices. Documentation of design rational and pointers between requirements and quality attributes and architecture components and connectors is necessary to promote traceability. Therefore, traceability can also be assessed by experts and system stakeholders on a scale of High, Medium and Low.

Isolation of Critical Functions → The evaluation of the software architectures with respect to this criteria is quite straightforward – critical functions can either be identified and isolated within a software architecture or not.

Assurance of Completeness and Determinism → Like isolation of critical functions, the completeness and determinism of a software architecture can be easily determined. The requirements and quality attributes that drive the software architecture can be evaluated with respect to Leveson's completeness criteria [74]. The externally visible behavior of the software can be modeled as a state machine and determinism can be assessed mathematically [76]. Any architectures that use probabilities or knowledge bases can be immediately ruled out as being deterministic – the behavior of the system is not determined until runtime. Consequently, a software architecture is also deterministic or not.

Simplicity → There are several indicators of simplicity in a software architecture including number of states, modes and interface types. Like the cost model parameter outlined in the previous chapter these indicators are High, Medium or Low. Dependencies should also be identified. By analyzing the dynamic and structural views of the software architecture, the static and runtime dependencies between software components and connectors can be determined and ranked High, Medium or Low. Finally, interface communications are either standardized or not.

Testability → Testability is mostly a function of whether or not there is a test strategy that complements the software architecture style, the types of faults that can be detected using that strategy, and whether or not it is easy or difficult to detect faults important to the success of the spacecraft mission.

Coupling → Two sets of coupling metrics are used and were derived from [80]. Inter-module coupling refers to the number of references between modules while intra-module coupling refers to the connectivity within a module. There are two metrics used to represent inter-module coupling: coupling-between-modules (CBM) and coupling-between-module-nodes (CBMN). CBM represents the number of non-directional, distinct inter-module references, while CBMN represents the number of non-directional, distinct, inter-module, node-to-node references. There are also two metrics used to represent intra-module coupling: coupling-between-nodes (CBN) and coupling-inside-modules (CIM). CBN represents the number of non-directional, distinct, inter-node references within a module and CIM the average of the coupling metrics for the nodes within the module. These measurements can be taken directly from the structural and implementation views to help determine the extensibility and evolvability of the software architecture.

Technology Forecasting → The technology forecasting activities were either used in the development of the software architecture or not. Indicators of technology forecasting include both hooks into the software and hardware architectures as well as infrastructure throughout the software product-line management to support the activities.

Development Cost → Measured through the cost model described in Chapter 4.

Intellectual Manageability → As described in the previous chapter, intellectual manageability relates to (1) whether or not the architecture is buildable given the current resources allocated to the project and (2) how closely the software architecture matches the spacecraft engineers' mental model of the system. One indicator of whether or not the system will be buildability given the current resources allocated to the project is the TRL of the architecture styles and tactics employed by the software engineers in constructing the software product-line architecture. A higher the TRL level corresponds to greater familiarity with the software architecture, giving an indication of buildability. In addition, some software architecture styles more closely match the engineers' mental model of the system. For example, architectures based on a subsystem- or functional-decomposition resemble how spacecraft engineers traditionally think about spacecraft systems and therefore will be more intellectually manageable than a new paradigm.

Programming Languages → Certain programming languages were constructed to support particular types of software architecture styles. For example, C++ and JAVA were constructed exclusively to support object-oriented analysis and design. Consequently, these two languages are excellent for implementing OO-based software architectures. However, these languages are may be less appropriate for developing software for architectures that abstract with respect to functions as opposed to objects. Consequently, the familiarity of the software developers with the programming language appropriate for implementing a particular architecture style and infrastructure to support development in that language also contributes to the buildability of the system.

Hardware Availability → This criterion is measured primarily by the availability of hardware to support the software constructed from the architecture. The software must fit within the bounds of available memory and processing speed provided by the avionics architecture.

Predictability → The predictability of software refers to the confidence of the software engineers that the software will repeatedly perform as expected, especially with respect to safety-critical or mission-critical functions. Focusing on the safety- and mission-critical portions of the software, the following checklist can be used to provide an indication of predictability associated with software developed from a particular architecture:

- For hazard-reducing outputs, soft and hard failure modes have been eliminated
- Hazard increasing outputs should have both soft and hard failure modes
- Multiple software paths for increasing safety
- Multiple triggers to enter a hazardous state
- No paths to unplanned hazardous states

Based on the degree to which the objectives in this checklist are achieved, the software architecture is considered to provide a greater or lesser degree of predictability with respect to safety- and mission-critical functions. The predictability can then be ranked High, Medium or Low.

Ability to respond to asynchronous space environment → The ability of software to respond to the asynchronous environment of space is largely dependent on (1) its ability to handle concurrency, (2) its ability operate in a degraded state, and (3) the breadth of fault types that can be handled by the FDIR software. This criteria is especially important during the Martian phase of the Exploration Initiative (or any deep-space mission), because the communication delay precludes ground operators from dealing with unforeseen occurrences or failures during the time-critical phases of the mission. Based on how the software architecture caters to handling concurrency, degraded states and failures, it can be evaluated with respect to its ability to respond to the asynchronous space environment.

Encapsulation → Encapsulation is either utilized in the software architecture construction or not. Minimizing encapsulation will generally increase the "Monitor"-ability of software, because programming elements needed for performing FDIR tasks by ground controllers are not hidden within an abstracted module. A high degree of encapsulation often coincides with information hiding, but this is not necessarily always true. It may be possible to encapsulate certain aspects of the software functionality without hiding vital information, thereby increasing modularity while maintaining runtime "Monitor"-ability.

Information Hiding → Like encapsulation, information hiding is either used or not in the construction of the low-level software architecture. Hiding information may decrease the "Monitor"-ability of the runtime software, if information important to fault diagnosis is hidden within an abstracted module. Consequently, careful thought must be placed into what information is hidden within a particular module and what information is made available to the rest of the software and to the operators.

5.1.1.3 – Step 3: Analysis of Conceptual Architectures

Based on the measurement techniques outlined in the section above, the conceptual architectures are evaluated to determine their ability to support the desired quality attributes. Table 5-1 presents a summary of the evaluation results, and the paragraphs below describe the results in greater detail. At this point of architecture development, several of the metrics used for evaluation may not be applicable because there is not enough information about the architecture to make an assessment. These criteria have a "NI" in the evaluation summary box indicating that there is not enough information at this point and these criteria will be used during the detailed design phase.

Goal-based Conceptual Architecture:

Analyzability with Respect to Safety → *Neutral*

The four criteria used to evaluate an architecture with respect to its analyzability with respect to safety are reviewability, traceability, isolation of critical functions and assurance of completeness and determines. The goal-based conceptual architecture described in this dissertation (the Mission Data System) is neutral with respect to the software engineers' ability to create software that is analyzable with respect to safety. MDS utilizes model-based development (the models

	Goal-based	Traditional Subsystem-based	Net-Centric (Implicit Invocation)	Functional Decomposition
Analyzability with respect to safety	Neutral	Supports	Neutral	Supports
Reviewability	High	High	Medium	High
Traceability	High	High	Medium	Medium
Isolation of Critical Functions	Detracts	Neutral	Neutral	Supports
Assurance of Completeness and Determinism	Detracts	Supports	Neutral	Supports
Ease of Verification and Validation	Neutral	Supports	Neutral	Supports
Reviewability	High	High	Medium	High
Isolation of Critical Functions	Detracts	Neutral	Neutral	Supports
Simplicity	NI	High	Medium	High
Testability	Medium	Medium	Medium	High
Sustainability	Supports	Neutral	Supports	Supports
Coupling	NI	NI	NI	NI
Technology Forecasting	High	Low	High	High
Affordability	--	--	--	--
Development Cost	NI	NI	NI	NI
Buildability	Neutral	Supports	Neutral	Supports
Intellectual Manageability	Medium	High	Medium	High
Programming Languages	NI	NI	NI	NI
Hardware Availability	NI	NI	NI	NI
Coupling	NI	NI	NI	NI
Ability to meet real-time constraints	Neutral	Neutral	Neutral	Neutral
Predictability	Low	High	Low	High
Ability to respond to asynchronous space environment	Supports	Detracts	Supports	Neutral
“Monitor”-ability	Supports	Detracts	Supports	Supports
Encapsulation	Low	High	High	High
Information Hiding	Low	High	Low	Low

Table 5-1. Conceptual Architecture Analysis Results Summary

can be developed using whatever modeling tools the engineers typically employ and can then be translated to fit into the MDS framework), which enhances the reviewability and traceability of the requirements and the subsequent design and implementation. Consequently, MDS is ranked high with respect to reviewability and traceability.

The MDS conceptual architecture does not abstract with respect to functions, but with respect to controllers, estimators and states. Consequently, individual software functions can not be isolated. Instead, the software determines the functions the spacecraft should perform to achieve certain goals. This approach makes the software extremely difficult to analyze with respect to safety and therefore detracts from this quality attribute.

In general, goal-based architectures are nondeterministic – it is impossible to determine the performance of the software prior to runtime. Furthermore, the software is not complete (with respect to the functions it must perform) at the time of deployment. MDS uses a process called goal-elaboration to deal with new situations. Although this provides additional functionality in terms of FDIR, it detracts from the quality attribute criteria that the software should be assured complete and deterministic before deployment. Consequently, the failure to fulfill this quality attribute criteria also detracts from this architecture style's analyzability with respect to safety.

Ease of Verification and Validation → *Neutral*

The four criteria used to evaluate an architecture with respect to its ease of verification and validation are reviewability, isolation of critical functions, simplicity and testability. This goal-based conceptual architecture detracts from the software engineers' ability to perform verification and validation activities. The evaluation of reviewability and isolation of critical functions are the same for ease of verification and validation as they are for analyzability with respect to safety: high and detracts respectively.

The simplicity of this conceptual architecture is low. Simplicity is determined by the number of states, modes and interface types, dependencies and interface standards. In MDS, the complexity of the system depends on the complexity of the model. The architecture as a whole has only six different components: Mission Planning and Execution, State Knowledge, State Control, Hardware Adapter, State Estimation and the Models. Therefore, without knowing the model of the system, there is not enough information to judge simplicity.

The testability of this conceptual architecture is medium. As outlined in [43], several legitimate V&V techniques have been identified as plausible testing strategies for the MDS architecture. The paper describes appropriate V&V techniques that can be used to prevent or mitigate software risks. However, these techniques have not been demonstrated on the architecture, so testability is ranked medium: V&V techniques have been outlined, but not demonstrated.

Sustainability → *Supports*

There are two criteria used in assessing an architecture with respect to sustainability: coupling and technology forecasting. At this point there is not enough information to evaluate the architecture with respect to the four coupling metrics described in the previous chapter. These measurements can be performed after detailed designs have been constructed. However, based on technology forecasting, this goal-based conceptual architecture supports sustainability. The MDS architecture forms the foundation for the spacecraft software; goals are changed, added and elaborated upon to achieve new objectives.

Affordability → *NI – Not Enough Information*

Buildability → *Neutral*

The four criteria used to evaluate an architecture with respect to its buildability are intellectual manageability, programming languages, hardware availability and coupling. At this point in the architecture definition, there is not enough information to choose the programming language or hardware to support the software. In addition, the description is not detailed enough to evaluate with respect to the four coupling metrics. Consequently, at this point, intellectual manageability is the sole indicator of buildability. For a goal-based architecture style, the intellectual manageability is low, because the TRL of MDS is quite low and architecture does not correspond closely to spacecraft engineers' mental model of their systems. Currently, MDS is at TRL 4 – validation in a laboratory environment. In order to increase the TRL, MDS must be validated in the relevant environment – in this case space. TRL 4 corresponds to a low familiarity with the software architecture style.

Ability to Meet Real-Time Constraints → *Neutral*

There are two criteria used to evaluate an architecture with respect to the software's ability to meet real-time constraints are predictability and the ability to respond to the asynchronous environment of space. The goal-based conceptual architecture neither supports nor detracts from the software engineers' ability to meet real-time constraints. From a qualitative standpoint, the predictability of the MDS software is low, because actions are determined during runtime based on goals and goal elaboration. However, when more detailed information about the architecture is available, the checklist described in the previous section should be used to determine predictability of safety-critical functions. On the other hand, goal-based architectures support responding to the asynchronous space environment, because they easily handle concurrency, degraded states, and a wide-breadth of fault types through its goal elaboration system.

"Monitor"-ability → *Supports*

The "monitor"-ability of software is determined by the use of encapsulation and information hiding. The goal-based conceptual architecture described in this dissertation (the Mission Data System) supports the software engineers' ability to monitor the software during runtime, because the architecture style was created with low encapsulation and information hiding in mind; states become the central portion of the architecture and state information is available to all parts of the software, enhancing its ability to deal with faults.

Traditional Subsystem-Based Decomposition Architecture:

Analyzability with Respect to Safety → *Supports*

The four criteria used to evaluate an architecture with respect to its analyzability with respect to safety are reviewability, traceability, isolation of critical functions and assurance of completeness and determines. The traditional subsystem-based architecture described in this dissertation supports the software engineers' ability to create software that is analyzable with respect to safety. As demonstrated in [131], traditional subsystem-based architectures can be easily created using highly readable model-based development, which enhances the reviewability and traceability of the requirements and the subsequent design and implementation. Consequently, this architecture style is ranked high with respect to reviewability and traceability.

The traditional spacecraft software conceptual architecture does not abstract with respect to functions, but with respect to subsystems and subsystem components. Within the subsystems and subsystem components, the architecture consists of a series of functions that can be executed by commands from the controller, some of which are critical and can be isolated. Consequently, the traditional subsystem-based architecture style neither detracts from nor supports the isolation of critical functions and is therefore ranked neutral.

The completeness and determinism of traditional subsystem-based software architectures can be easily demonstrated, especially through the use of executable requirements specifications as shown in [131]. Therefore, this architecture style supports assurance of completeness and determinism.

Ease of Verification and Validation → *Supports*

The four criteria used to evaluate an architecture with respect to its ease of verification and validation are reviewability, isolation of critical functions, simplicity and testability. The traditional spacecraft software architecture supports the software engineers' ability to perform verification and validation activities. The evaluation of reviewability and isolation of critical functions are the same for ease of verification and validation as they are for analyzability with respect to safety: medium and detracts respectively.

The simplicity of this conceptual architecture is high. The number of states and modes is low; these values are kept in the central controller and describe the operating state and mode of the spacecraft as a whole. Typical modes include position hold, startup and momentum management. Dependencies are also low, because each module is a part and only interacts with its parent subsystem.

The testability of this conceptual architecture is also medium. Individual subsystem functionality is tested independently, integrated and then tested as part of the spacecraft as a whole. However, the type of and number of faults that can be detected using this architecture and testing strategy are low; FDIR is largely relegated to ground controllers.

Sustainability → *Neutral*

There are two criteria used in assessing an architecture with respect to sustainability: coupling and technology forecasting. This conceptual architecture style detracts from sustainability, because traditional spacecraft architectures are created individually and specifically for each mission and mission objectives. Because the software is so mission-specific, it cannot be adapted as technology evolves.

Affordability → *NI – Not Enough Information*

Buildability → *Supports*

The four criteria used to evaluate an architecture with respect to its buildability are intellectual manageability, programming languages, hardware availability and coupling. The intellectual manageability of this architecture style is high because the TRL is at level 9 – the actual system has been flight proven through many successful mission operations. Furthermore, the decomposition of the spacecraft into subsystems and their corresponding functions accurately matches spacecraft engineers' mental models of the spacecraft.

Ability to Meet Real-Time Constraints → *Neutral*

There are two criteria used to evaluate an architecture with respect to the software's ability to meet real-time constraints are predictability and the ability to respond to the asynchronous environment of space. This conceptual architecture is neutral with respect to the software engineers' ability to meet real-time constraints. Although highly predictable, the traditional spacecraft software architecture style does not provide the spacecraft with the ability to independently handle the asynchronous environment of space. Unexpected conditions and faults are handled by ground controllers; command sequences that mitigate these anomalous situations are uploaded to the spacecraft in real time.

"Monitor"-ability → *Detracts*

The "monitor"-ability of software is determined by the use of encapsulation and information hiding. As discussed in [39], the traditional spacecraft software architecture style detracts from the "monitor"-ability of the software, because of its use of encapsulation and information hiding.

Net-Centric Conceptual Architecture:

Analyzability with Respect to Safety → *Neutral*

The four criteria used to evaluate an architecture with respect to its analyzability with respect to safety are reviewability, traceability, isolation of critical functions and assurance of completeness and determines. The net-centric conceptual architecture described in this dissertation (uses implicit invocation) neither detracts from nor supports the software engineers' ability to create software that is analyzable with respect to safety. Although not currently used, the components in the net-centric architecture style can be developed using model-based techniques, which enhances the reviewability and traceability of the requirements and the subsequent design and implementation. Each of the different service, avionics and control modules can be modeled and then assembled in different configuration to evaluate the new configuration. Consequently, the net-centric architecture style is ranked medium with respect to reviewability and traceability.

Like the traditional subsystem architecture, the implicit invocation architecture style is neutral with respect to isolation of critical functions. Functionality is relegated to the subsystem, avionics and controller modules, similar to the traditional architecture style. However, critical functions can be isolated within each of the various spacecraft components. This architecture style is also neutral with respect to assuring the software with respect to completeness and determinism. Although the individual modules can be assured complete and deterministic, the implicit invocation engine may not produce results that ensure safety. An implicit request for services, may not be answered if the modules that can deliver that service are faulty or unavailable.

Ease of Verification and Validation → *Neutral*

The four criteria used to evaluate an architecture with respect to its ease of verification and validation are reviewability, isolation of critical functions, simplicity and testability. This net-centric conceptual architecture neither detracts nor supports the software engineers' ability to perform verification and validation activities. The evaluation of reviewability and isolation of critical functions are the same for ease of verification and validation as they are for analyzability with respect to safety: medium and neutral respectively.

The simplicity of this conceptual architecture is medium. Although the number of states and modes is similar to that of the traditional spacecraft software architecture, there are more complex interfaces between the various spacecraft components, i.e. controller, service and avionics modules. The testability of this conceptual architecture is medium. The testing strategy is similar to that of the traditional spacecraft architecture; individual components are tested and then different configurations of components.

Sustainability → *Supports*

There are two criteria used in assessing an architecture with respect to sustainability: coupling and technology forecasting. The net-centric architecture supports sustainability, because new hardware and software components can be added to the architecture as needed and become new assets in the service modules available.

Affordability → *NI – Not Enough Information*

Buildability → *Neutral*

The four criteria used to evaluate an architecture with respect to its buildability are intellectual manageability, programming languages, hardware availability and coupling. The intellectual manageability of the net-centric architecture style is medium. Although the software is decomposed structurally like the traditional spacecraft software architecture, the implicit invocation strategy has not been used before in the space environment. Consequently, that portion of the software system has a low TRL level leading to a lower buildability level that the traditional architecture. However, the decomposition of the spacecraft into various modules that perform different subsystem functions does coincide with how spacecraft engineers traditionally represent spacecraft operations. Therefore, until more information becomes available, the net-centric architecture is considered neutral with respect to buildability.

Ability to Meet Real-Time Constraints → *Neutral*

There are two criteria used to evaluate an architecture with respect to the software's ability to meet real-time constraints are predictability and the ability to respond to the asynchronous environment of space. Predictability is low, because there are a number of ways the various available components can respond to a service request. However, this negative aspect of the architecture is counter-balanced by the enhanced ability to deal with new and unforeseen situations. Modules can be used in unexpected, yet effective ways. For example, if extra processing power is needed, multiple avionics modules can combine to provide the extra services.

"Monitor"-ability → *Supports*

The "monitor"-ability of software is determined by the use of encapsulation and information hiding. Like the traditional spacecraft software architecture, the net-centric style uses encapsulation of functionality in the decomposition into software modules. However, information hiding is low, because many of the variables, states and modes must be available to the controller module that is requesting services. Consequently, the net-centric architecture style supports "monitor"-ability.

<u>Functional Decomposition Conceptual Architecture</u>:

Analyzability with Respect to Safety → *Supports*

The four criteria used to evaluate an architecture with respect to its analyzability with respect to safety are reviewability, traceability, isolation of critical functions and assurance of completeness and determines. The functional decomposition supports software engineers' ability to create software that is analyzable with respect to safety. Functional decomposition components can be easily developed using model-based development, which enhances the reviewability and traceability of the requirements and the subsequent design and implementation. Because the system is decomposed with respect to functions, isolating critical functions is quite simple. Safety analyses can then focus on these safety-critical portions of the code more easily. Again, because modules in a functional decomposition can be developed using a model-based development environment it is easy to check for completeness and determinism.

Ease of Verification and Validation → *Supports*

The four criteria used to evaluate an architecture with respect to its ease of verification and validation are reviewability, isolation of critical functions, simplicity and testability. Functional decomposition style based architectures support engineers' ability to perform V&V activities. The evaluation of reviewability and isolation of critical functions are the same for ease of verification and validation as they are for analyzability with respect to safety: high for functional decomposition.

The simplicity of this conceptual architecture is high. Layered architectures, as well as functional decompositions, are software design practices that have been used and studied for many years. These architecture styles have well-known properties, are taught in many computer science and software engineering schools, and the TRL is high due to its application is many safety- and mission-critical systems.

The testability of this conceptual architecture is high. Layers can be tested as developed. Supports incremental testing of kernel functions, utilities and finally processes. Regression testing should be completed as each new layer is added, which is especially important due to the modular accretion of assets over time.

Sustainability → *Supports*

There are two criteria used in assessing an architecture with respect to sustainability: coupling and technology forecasting. Functional decomposition supports sustainability, because new hardware and software functions can be added as needed, as long as they conform to interface standards. In addition, because a layered operating system is used, changes in the avionics hardware only effect the lowest software level. Changes can be made to the hardware-software interface without drastically impacting the rest of the software.

Affordability → *NI – Not Enough Information*

Buildability → *Supports*

Layered and functional decomposition software architectures are common architectural styles that have been used frequently for many years in both the operating system domain and for embedded controllers. The buildability of these architecture styles is reflected by their widespread use in industry, especially on safety- or mission-critical systems. Furthermore, the abstraction of low-level detail to low-level layers makes the construction of high-level programs that use functional decomposition architecture style much easier to build.

Ability to Meet Real-Time Constraints → *Neutral*

Process and task schedulers are contained at the operating system level. There are several well-known techniques for writing predictable schedules for hard real-time software systems. Predictability is high, but the ability to deal with the asynchronous space environment is largely dependent on the FDIR approach used. Consequently, if FDIR is added as an independent function, concurrency capability is low. FDIR needs to be integrated into each functional component to be effective.

"Monitor"-ability → *Supports*

Because the low-level details of the hardware are abstracted from the high-level processes, the visibility down to kernel tasks and device information is less than in an architecture that elevates this information to a much more important level such as state. However, requests for information from the operating system can be incorporated into the subsystem utility level as it is in many commercial operating systems.

5.1.1.4 – Step 4: Down-Selection

Based on the results of the conceptual architecture style evaluation, the traditional spacecraft software architecture should be replaced with newer, more effective spacecraft software architectures. This conclusion coincides with the original hypothesis that the traditional spacecraft software architecture is no longer suitable because of its inability to autonomously respond to new and unforeseen situations and the lack of visibility into runtime performance and telemetry. The net-centric and functional decomposition architecture styles provide enhanced capabilities, both in terms of runtime "monitor"-ability and sustainability, while a goal-based FDIR engine may provide improved autonomy capabilities. Consequently, the architecture style options should be either (1) down-selected to include net-centric and functional decomposition architecture styles, and possibly a goal-based architecture style exclusively for fault protection or (2) integrated to create a hybrid architecture style that provides the desirable capabilities of the net-centric, functional decomposition and goal-based architecture styles.

The net-centric, functional decomposition, and goal-based architectures can be evaluated independently of one another during detailed design. During the detailed design phase more information will become available about the requirements and characteristics of the desired software system, which aids engineers in performing a more detailed assessment of the suitability of the various architecture styles to achieve quality attributes. On the other hand, aspects of the various designs can be combined to exploit the effective and valuable portions of the architecture style for spacecraft software engineering, while eliminating some of the more negative aspects of the style in favor of the strengths of another style. For example, the architecture might employ a functional decomposition architecture style for the application software, built upon a layered operating system that contains implicit invocation and goal-based architecture style techniques for fault isolation, detection and recovery. Both of these options can be explored during the detailed design phase.

5.1.2 – Detailed Design

During the detailed design phase, software engineers construct realizational architectures (software architecture descriptions that are at a lower-level of abstraction than those of conceptual architectures) based on the down-selection results identified during the preliminary design phase. These realizational architectures are modeled using ACME, which is an architecture description language, and the modified 4+1 view model described in the previous chapter. The results of the detailed design phase include:

- five views of the realizational architectures as depicted in ACME based on the down-selected architecture style options,
- a mapping between the five architecture views,

- an analysis of these diagrams with respect to their adherence to quality attribute criteria and how they manage attribute trades, and
- the selection of a software architecture upon which the implementation is based.

In this section, an example portion of a realizational architecture for a traditional spacecraft software architecture style is presented. The style was chosen as the example, because of the amount of information available about this software architecting technique. Full descriptions and analyses of the down-selected detailed realizational architecture options cannot be constructed because of a lack of detailed information. As outlined in the Conclusion, the modeling and analysis of the realizational architectures is a key part of the future work based on this dissertation.

5.1.2.1 – Architecture Description Example

Figures 5-7 and 5-8 depict the structural and implementation views of a traditional spacecraft decomposition conceptual architecture as it is represented using the ACME software architecture description tool. In particular, the thermal and fault protection software are highlighted. During the detailed design phase, these conceptual architectures are refined into more detailed ACME models representing the derived realizational architecture. Although a traditional spacecraft decomposition architecture was not one of the down-selected results, it was used as an example, because of the historical information available about this style. Furthermore, the lack of detailed information precludes the author from modeling a more detailed, realizational architecture. Consequently, the example is a conceptual architecture. However, figures 5-7 and 5-8 provide an excellent indication of how to represent different viewpoints using ACME and how these viewpoints are essential in providing different sets of information to the engineers about the software architecture and its properties.

Figure 5-7 depicts a typical architecture representation in ACME. In this architecture style, component types include system level, subsystem level, component level and fault detection components. As seen in the diagram, the system level component, in this case the Command and Data Handling Computer (CDHC), is represented with a dark green oval. The various subsystems such as Thermal, Power, GN&C and Payload are represented with medium green ovals. Finally, subsystem component functions (for the Thermal Subsystem examples include Heat Sensing and Thermal Balance and Conditioning Control) are represented with light green ovals. Fault Detection, Isolation and Recovery modules are represented as dark pink boxes. As seen in the diagram, and described in the traditional subsystem decomposition style description, the Central FDIR is a separate, add-on module that interfaces into each of the subsystems. Connectors in ACME are represented by directed arrows with the small yellow circle in the middle of the connector. This circle along with the color of the line indicates the characteristics of that particular connector. For example, in this diagram there are two types of connectors: control and data transfer.

Figure 5-7 shows an example of the structural viewpoint of the thermal subsystem, while Figure 5-8 the implementation viewpoint. As seen in the diagrams, these viewpoints represent very different aspects of the same thermal subsystem. The structural viewpoint depicts the static structure of the system, or how the system is decomposed into predefined types of components and connectors. The structural view of the traditional subsystem-based decomposition architecture clearly illustrates the hierarchical nature of this style. Thermal functions interface with their parent subsystem, subsystem interface with the central computer, and the FDIR components are added on as a separate subsystem with interfaces into the other subsystems.

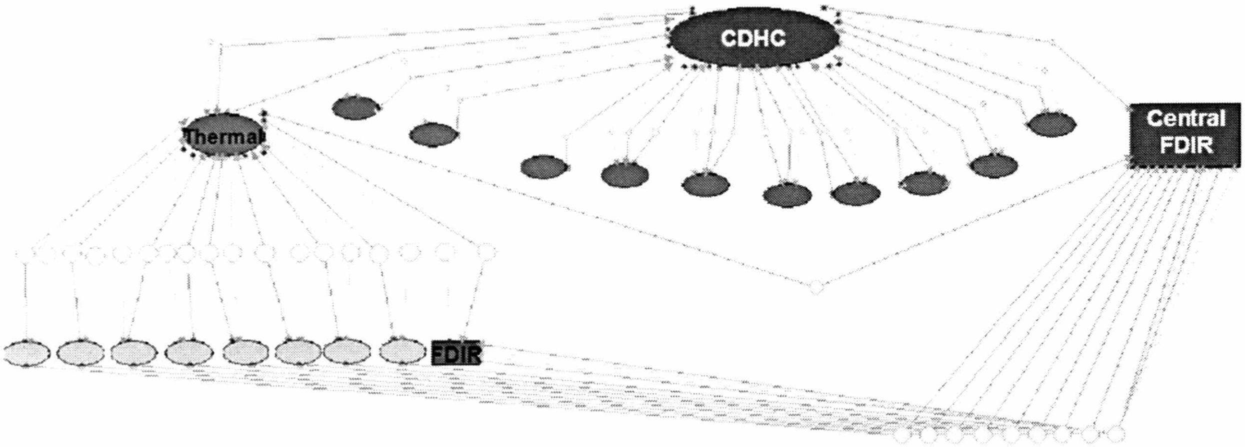


Figure 5-7. Structural View Example in ACME

Figure 5-7 addresses certain quality attribute concerns including sustainability, ease of verification and validation and "monitor"-ability. For example, as discussed in Appendix C, one of the stakeholders that utilizes the structural viewpoint to gain insight into the software architecture are the software programmers. The coupling metrics, CBN (coupling-between-nodes) and CIM (coupling-inside-modules), are determined using this diagram and provide some insight into the extensibility and evolvability of the software system. However, another viewpoint is necessary to evaluate the architecture with respect to the two remaining coupling metrics, CBM (coupling-between-modules and CBMN (coupling-between-module-nodes).

Figure 5-8 illustrates the implementation view of the thermal and FDIR subsystems using the traditional spacecraft software architecture style. As seen in the diagram, the Central FDIR is developed alongside the FDIR modules that interface into the various subsystems. In addition, each of the subsystems is implemented as a unit; the Thermal subsystem is built as a stand-alone module that interfaces both with the CDHC and FDIR.

The implementation view depicted in Figure 5-8 describes the grouping of static entities into modules, or chunks, that are assigned to various programmers or programming groups. This diagram addresses the quality attributes such as sustainability and buildability. Programmers also utilize this view to determine the extensibility and evolvability of the architecture, but they gain additional insight into the software architecture not provided by the structural view. Because this view illustrates the groupings of software into programmable modules, this view aids in determining the software's buildability.

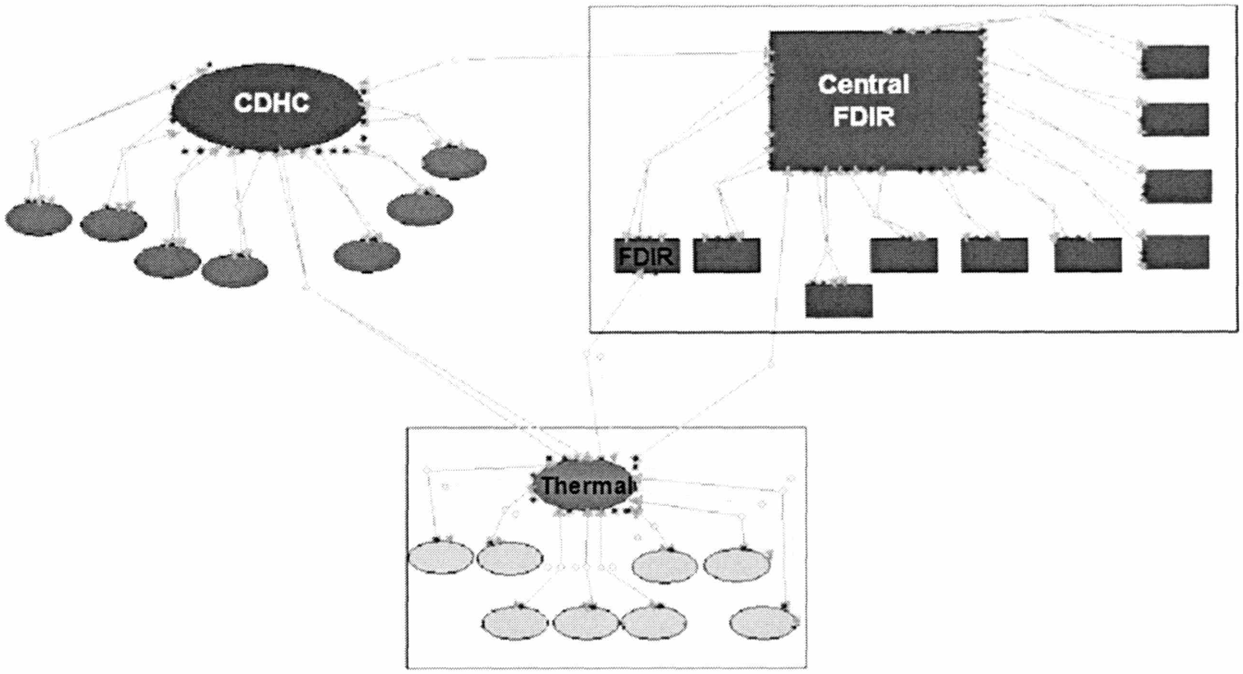


Figure 5-8. Implementation View Example in ACME

As previously mentioned, a full set of all five viewpoints (described in Appendix C) are necessary to create a full, detailed description of a software architecture. The various stakeholders require the different views to observe and measure the various proprieties of the software architecture that stem from different aspects of that architecture. As described in the Conclusion, further work is needed in the area of defining these viewpoints, integrating them into the ACME environment and making seamless and formal transitions between the viewpoints. This research will help ensure that the detailed design descriptions are consistent with one another and provide the necessary information without redundancy or discrepancies.

5.2 – Summary

This chapter presented the results of performing the design activities of the MASATA development approach on the Exploration Initiative software engineering effort. Measurement techniques, whether qualitative or quantitative, were identified with respect to each of the quality attribute criteria to determine whether or not the architecture style is suitable for creating software that caters to the needs of the system stakeholders. After a down-selection of conceptual architectures based on the results of the evaluation, realization architectures, architecture descriptions at a lower level of abstraction are generated and evaluated to determine which is more suitable to serve as the software product line architecture. In this analysis, it was shown that a functional-decomposition software architecture, supported by some goal-based fault detection, isolation and recovery software is the most appropriate architecture for achieving the requirements and desired quality attributes of the US Space Exploration Initiative stakeholders. Finally, the software architecture takes form in the software design phase. The realizational architecture is evaluated throughout the entire development process in order to ensure that the architecture caters to the needs of the software system stakeholders.

[This page intentionally left blank.]

Chapter 6:

Conclusion

This dissertation presented a new methodology for creating large-scale, safety-critical software for spacecraft that utilizes applicable development techniques outlined in both the current software and spacecraft systems research domains. The development strategy is based on a software product line approach that is supported by model-based development, safety-driven design and software architecture-centric engineering. The methodology was demonstrated on a spacecraft engineering case study – NASA's Exploration Initiative, whose objectives involve creating a sustainable human presence in space.

First, the background information and related research that lays the foundation for the new approach to spacecraft software development was presented. Advancements made in the commercial software engineering domain as well as state-of-the-art research from spacecraft engineers are integrated into spacecraft software enginering. Furthermore, by incorporating the successful aspects of current and proposed spacecraft software architectures, such as State Analysis and the Mission Data System, with legacy information from the traditional subsystem-decomposed spacecraft architecture and software-based spacecraft accidents, a new formula for guiding the process by which spacecraft software is developed can be easily constructed. Based on the background information and experience described in this dissertation, a software product-line approach in conjunction with model-based development, safety-driven design, new software architectures and software architecture evaluation are proposed as key to creating a sustainable software development effort to support the US Space Exploration Initiative.

Next, the details of developing a software product line for the Exploration Initiative, which forms the basis of the new approach, is outlined. Creating a software product line for spacecraft in general and the Exploration Initiative in particular involves the development of infrastructure in three different areas: (1) technical, or software engineering, activities, including core asset development, individual product development and product-line scoping; (2) technical management activities, including defining a production plan, lifecycle model and approach to safety; and (3) organizational activities such as developing a management structure. These three practice areas provide the foundation for creating an effective software product-line development approach for the Exploration Initiative. One of the most important technical activities involves core asset development, in particular the development of the software product line architecture. The appropriateness of this architecture to fulfilling the requirements and desirable quality attributes is essential to the success of the software development endeavor for the Exploration Initiative, because it provides the foundation from which the various products in the line are constructed. To support the construction and selection of an appropriate software product line architecture, an architecture-centric development process known as Multi-Attribute Software Architecture Trade Analysis, or MASATA, which complements and is integrated with the software process model used to support the software product line approach.

MASATA consists of software architecting activities throughout every phase of the software development lifecycle (concept generation and feasibility study, requirements and design). During the concept generation and feasibility study phase, software engineers focus on identifying stakeholders and the quality attributes they desire from the system. They also determine the impact various system architectures have on the cost and risk associated with creating a software system to support that architecture. During the requirements specification phase, the software architecting process begins and engineers elaborate upon the quality attributes identified in the previous step and begin identifying conceptual architecture styles that may be appropriate given the characteristics of the application domain. Finally, the software architecture takes form in the software design phase. In this approach, the realizational architecture is evaluated throughout the entire development process in order to ensure that the architecture caters to the needs of the software system stakeholders. In future work building upon this research, a realizational architecture can be developed and evaluated in greater detail than shown in this dissertation.

The artifacts developed as a part of the software product-line approach as well as the results of performing MASATA were demonstrated on one of the system architecture options created to support the goals and requirements of the US Space Exploration Initiative. First, an analysis of the impact of a set of lunar and mars transportation and surface operation architectures on the cost and risk associated with the software needed to support the architectures was performed. A set of both software and human-computer interaction (HCI) evaluation parameters were used to assign values to the cost and risk associated with providing the necessary software functionality to support the different architectures. The cost and risk values were summed to obtain total cost and risk rankings for the various evaluated system architectures. The system architectures were then ranked relative to one another and with respect to a nominal case, or average, system architecture in terms of these measured software metrics. Finally, discounts were applied to development cost and risk based on how the production plan allows for product line reuse among the various vehicles and facilities. The main conclusion that can be drawn from the evaluations performed during the concept generation and feasibility studies is that simple mission architectures tend to have lower development costs, development risks and mission risks and as system architecture complexity increases, so do the associated software costs and risks. In addition, for architectures with vehicles and surface elements with similar functional requirements, product-line development approaches tend to reduce the affect the additional complexity has on development costs and risks, but do not affect the additional mission risks.

Stakeholders and desirable quality attributes were also described during this phase. Along with the functional requirements for the software, several quality attributes that support the objectives of the Exploration Initiative were also identified, including analyzability with respect to safety, ease of verification and validation, sustainability, affordability, buildability, ability to meet real-time constraints, and “monitor”-ability.

During the requirements specification phase of the software development lifecycle several architecture styles were identified as possible candidates to serve as guidelines for the software product-line architecture. For the Exploration Initiative conceptual software architectures based on the traditional subsystem-based decomposition, state analysis, implicit invocation-based net-centric and functional decomposition architectural styles were created. These architecture styles were chosen because of their timeliness and applicability to the spacecraft software domain. It is important during actual software architecture development not to discount any architecture styles prematurely. Software engineers also identify information important to the product line architecture during the requirements phase including scope and variation points. The quality attributes are also refined at this point and a preliminary set of criteria associated with these qualities are identified. For the Exploration Initiative, analyzability with respect to safety was expanded into reviewability, traceability isolation of critical functions, and assurance of completeness and determinism; ease of verification into reviewability, isolation of critical functions, simplicity, and testability; sustainability into coupling and technology forecasting; affordability into development cost; buildability into intellectual manageability, programming languages, hardware availability, and coupling; ability to meet real-time constraints into predictability and ability to respond to asynchronous space environment; and “monitor”-ability into encapsulation and information hiding.

During preliminary design, conceptual architectures were generated with guidance from each of the four architecture styles chosen for the example. Evaluation methods, whether they be quantitative or qualitative, were then identified for each of the quality attribute criteria. Finally, the conceptual architectures were evaluated with respect to the quality attribute criteria. The preliminary design phase ends with a down-selection to two or three conceptual architectures which guide the design of the realizational software architectures.

Finally, during the detailed design phase software engineers construct realizational architectures (software architecture descriptions that are at a lower-level of abstraction than those of conceptual architectures) based on the down-selection results identified during the preliminary design phase. These realizational architectures are modeled using ACME, which is an architecture description language, and the modified 4+1 view model described in the previous chapter. The results of the detailed design phase are five views of the realizational architectures as depicted in ACME based on the down-selected architecture styles, an analysis of these diagrams with respect to their adherence to quality attribute criteria and how they manage attribute trades, and culminates in the selection of a software architecture upon with the implementation is based.

In conclusion, several software engineering and management artifacts were produced as a part of this exercise and can be used to support a software enterprise for the Exploration Initiative. These artifacts are the result of adapting a software product-line approach to spacecraft software engineering and performing the MASATA process on one of the system architectures created for the Exploration Initiative. These artifacts include descriptions of the core assets, the product line scope, a product development approach and production plan, a safety-driven software lifecycle model, a common system engineering and integration organization structure, a new architecture-evaluation based development strategy known as MASATA.

The contributions of this work have application in both the software engineering and spacecraft engineering domains. First and foremost, this dissertation demonstrated the benefits and applicability of a software product line approach as applied to spacecraft software engineering in general and the US Space Exploration Initiative in particular. By creating a framework within which and through which the various vehicles and facilities necessary to achieve the mission objectives of the Exploration Initiative, NASA and its contractors will be able to achieve the ambitious goals within the allotted budget and on time. This approach decreases the cost of creating subsequent products in the line by focusing on the software product line architecture as the basis for all future development. As the core asset base grows and evolves around the product line architecture, the cost and time associated with each additional product in the line will continue to decrease, while maintaining, and even improving the quality of those products.

Second, the integration of modern software engineering techniques into the spacecraft software engineering domain will bring spacecraft engineering "up to speed" with the current practices and state-of-the-art research prevalent in the software engineering community. The dichotomy between the practices and techniques used in the spacecraft and software engineering communities precludes spacecraft engineers from achieving the full potential of the mechanical systems. By slowly, but steadily, integrating modern software engineering techniques into the spacecraft domain, engineers will be able to verify and validate these new approaches and accomplish the new, more challenging mission objectives that could not be previously achieved.

Finally, MASATA is the first step toward creating a science of design for software engineering. For many years computer scientists and engineers have debated whether or not the development and deployment of software truly be considered an engineering discipline. By creating a science of design for software engineering, the field moves closer to becoming a true engineering discipline. Integrating systems engineering practices, traditional engineering trade analyses, and system architecting practices with the software engineering lifecycle will help guide software engineers to make educated and reasonable decisions with respect to the software architecture, design, and implementation. Instead of merely employing the newest strategies from current software engineering literature, engineers are aided in identifying the software needs of the system and formulating an appropriate solution that caters to those needs. This approach to software engineering represents a new trend within the field and brings it closer to other, more traditional, engineering disciplines.

6.1 – Future Work

There are several avenues for future research and development based on the results of this dissertation, several of which involve expanding the less detailed portions of the software product line approach and the MASATA process. These research areas include (but are not limited to) the following:

- Application of MASATA at the detailed design level
- Integration of modified 4+1 view model with ACME
- Exploration of compliance with the modified 4+1 view model with other architecture description standards
- Construction of a reusable, model-based core asset base
- Investigation of additional quality attribute metrics

In order to test the applicability of MASATA at the detailed design level, it is necessary to perform the methodology with detailed requirements specifications from a real project. This additional information would provide software architects with the information necessary for developing the lower-level software architecture and perform evaluation at the detailed design level. At this level, different, more detailed evaluation metrics may exist because of the more detailed information available. For example, in a realizational software architecture, how the software handles concurrency is represented in detail. Consequently, it become easier to measure the software's responsiveness to the asynchronous space environment. Applying MASATA at the detailed design level is essential to determining its applicability at successfulness for spacecraft software architecture evaluation.

In support of applying MASATA at the detailed design level, it is important that a formal connection between the modified 4+1 view model described in this dissertation and the ACME Studio software architecture description tool be defined. Each view should have a representation in ACME and these views should be able to be linked to one another in ACME Studio. This integration will help enhance software and system engineers understanding of the software architecture and ensure that the resulting product adheres to the requirements and constraints described in the various architecture views.

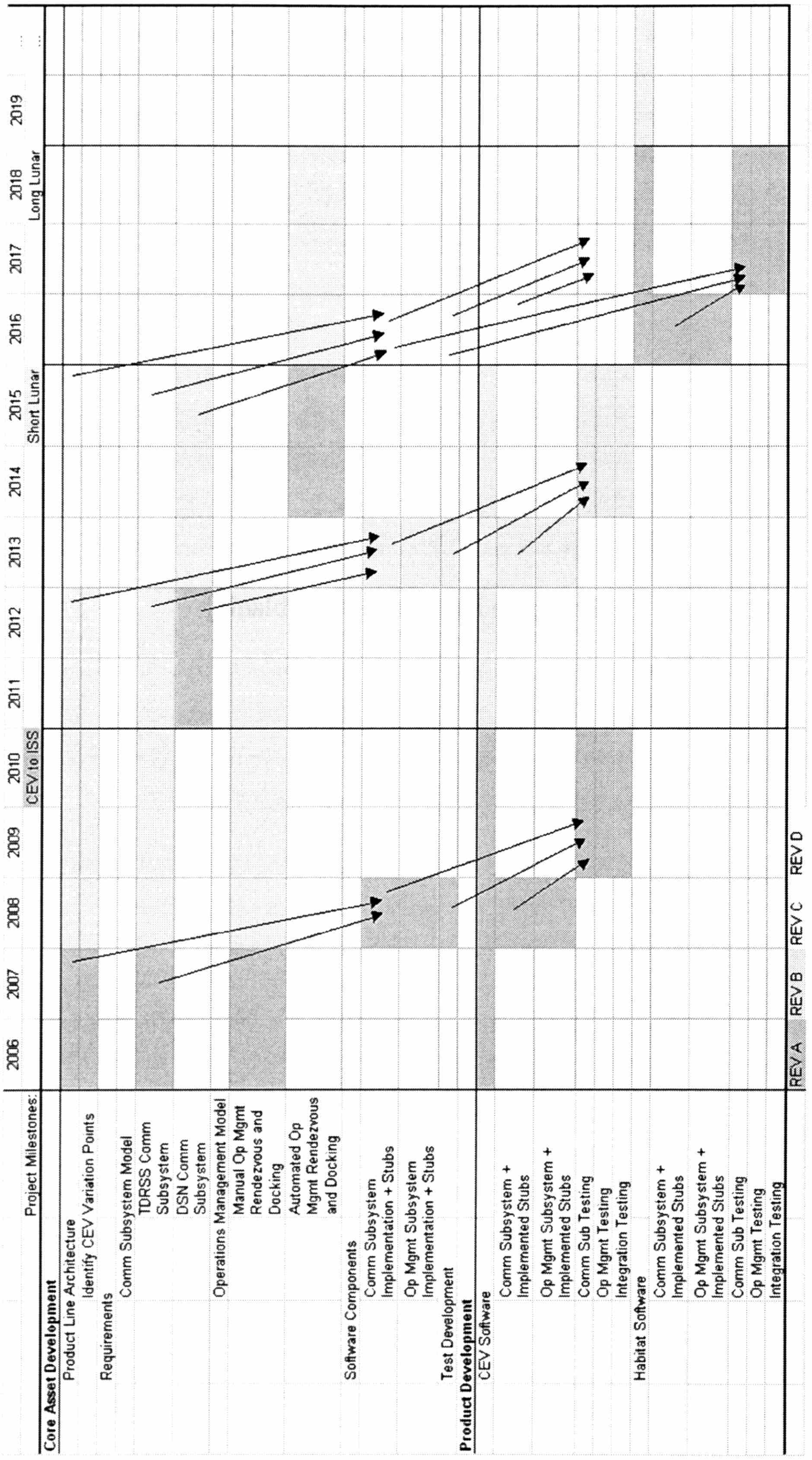
Furthermore, there are several standards today for creating architecture descriptions. For example, the 4+1 view model have become a "de facto" standard in the software engineering community. DoDAF (the Department of Defense Architecture Framework) provides guidelines for generating system architecture descriptions and has become the standard system architecture description framework for many aerospace applications. Consequently, it will be useful to demonstrate how the modified 4+1 view model adheres to the precepts of these standards so that it can be used on project that require observance to these standards.

Another important aspect of this software product-line research involves creating a reusable core asset base of model-based components. In order to determine the effectiveness of creating a model-based core asset base, it is important to create these models and test their applicability on a variety of products in the line. For example, several subsystem-level components such as communications and guidance could be created and then modified for testing in very different vehicles such as the Crew Exploration Vehicle and the Trans Lunar Injection stage. By creating and testing a sample core asset base in this manner, the costs and benefits of this approach can be determined before the entire approach is adopted. Consequently, any problems encountered in the core asset base development or integration can be alleviated before full development begins.

Finally, it would be highly useful to create a type of quality attribute database that contains a fuller listing of quality attributes, their descriptions, the criteria used to describe them, and metrics that can be used to evaluate software architectures with respect to these criteria. By creating this type of quality attribute database, software stakeholders will have an enhanced ability to identify the desirable quality attributes for their systems and select which criteria and metrics are most appropriate for evaluating the software architecture for their particular system with respect to these attributes. Furthermore, matrices can be created to explicitly identify trades between quality attributes and quality attribute criteria. These matrices will provide a simple yet detailed reference for identifying where tradeoff points will occur in the software architecture and possible strategies for mitigating these tradeoffs.

[This page intentionally left blank.]

Appendix A: Example Production Plan



[This page intentionally left blank.]

Appendix B: Subsystems and Functions

Flight Vehicles										
Vehicle Control Subsystems					Communication and Tracking Subsystem	Systems Mangement Subsystem	Operations Management Subsystem	Information Management Subsystem	Science and Payload Processing Subsystem	Crew Display and Control Subsystem
GN&C Subsystem	Thermal Control Subsystem	Electric Power Subsystem	Environmental Control and Life Support Subsystem	Fluids Management Subsystem						
State Maintenance	Heat Sensing	Sensing and Switch Control	Temperature and Humidity Control	Fluids Monitoring and Management (N2, H2O, Waste)	S/G Comm	Operating System	Rendezvous and Docking Control	Maintenance Management	Payload Status Monitoring	Crew Data and Display
Orbit Adjust Control	Status and Perform Monitoring	Redundancy (Routing) Control	Fire Detection and Suppression Control	Storage and Transfer Control	S/G Telemetry Comm	Fault and Redundancy Management	Proximity Operations Monitoring and Control	Real-Time Command and Control (Crew, Operations, EVA and Information Systems Management)	Science Experiment Data Collection and Recording	Caution and Warnings Display
Attitude Control	Waste Heat Acquisition and Transportation	Power Flow and Loading Monitor, Evaluation and Control	Atmosphere Monitoring and Control	Leak Detection, Isolation and Control	S/S Comm	Network Operating System	Area Traffic Management and Control (Manned and Unmanned)	Simulation and Training	Experiment Data Processing and Control	Crew Control of Platform Systems
Momentum Control	Load Sharing and Control	Detect, Isolate and Clear Power Faults	Water Recovery Management	Waste Fluid Handling, Vending and Dumping	S/G Detection and Tracking Acquisition	FDIR	Flight Planning and Control	Payload Management	Payload Operations Control	Crew User Services (WP, Planning, Spreadsheets, etc.)
Flex Body Control	Leak Detection, isolation and Repair Control	Caution and Warning Generation	EMU Service and Checkout	Caution and Warning Generation	Sensor Control	System Service (DEMS, Network Control, Time Mangement/Control, Data Distribution, Transaction Management)	Caution and Warning Generation	Test Management	Video/Imagery Data Management	Avionics Subsystem I/O and Control
Propulsion Control	Data Acquisition and Signal Conditioning	Power Transient Security Control	Airlock Pressure Control	Fuel and Propellant Management and Control	Track Management and Distribution	Access and Security Control	Ascent Control	EVA Planning	Experiment Managmenet and Control	Crew Override and Control of GN&C Functions
FDIR I/F	Caution and Warning Generation	Solar Array Management	Gasses Control		Video Comm	Caution and Warning Control	Orbit Entry Control	Resource Management and Service (Crew and Equipment)		
On-Orbit Test Support	Thermal Balance and Conditioning Control	Thermal Reactor Management	Special Environment Control		Audio Management and Comm	System Configuration Management	Airlock and Doors Control	Quarantine Monitoring and Management		
Re(Boost) Managemental and Control	FDIR	Ion Reactor Management	Contamination Detection and Removal Control		C&T Monitoring and Status	Re(Init) and Shutdown	De-Orbit/Descent Control	Equipment Stowage Management		
Monitor and Support to Operations Control		Ion Solar Management	Caution and Warning Generation		Orbital Debris Monitoring	User I/F Management	Landing and Post Landing Control	Perishible Inventory Management		
Caution and Warning Generation			Environment Monitoring and Management		Caution and Warning Generation	(Over)Load Control	OMA Control	Housekeeping and Trash Management		
GN&C Coordination Control					FDIR Interface	I/O Handling	EVA Control	Crew Health Management		
Trajectory (Transit and Return) Adjust Control					Trajectory Monitor	Health Monitoring	Stage Ejection	Software Configuration Management		
Position Control					Comm Data Traffic Managmenet			Platform Configuration Management		
					Crew/Equipment Retrieval Support			Onboard Information System		
					Space Weather Monitoring			Logicstics and Supply Management		
								Maintenance/Service Management		

Ground Vehicles										
Vehicle Control Subsystems					Communication and Tracking Subsystem	Systems Mangement Subsystem	Operations Management Subsystem	Information Management Subsystem	Science and Payload Processing Subsystem	Crew Display and Control Subsystem
GN&C Subsystem	Thermal Control Subsystem	Electric Power Subsystem	Environmental Control and Life Support Subsystem	Fluids Management Subsystem						
State Maintenance	Heat Sensing	Sensing and Switch Control	Temperature and Humidity Control	Fluids Monitoring and Management (N2, H2O, Waste)	G/S Comm	Operating System	Rendezvous Control	Maintenance Management	Payload Status Monitoring	Crew Data and Sisplay
FDIR I/F	Status and Perform Monitoring	Redundancy (Routing) Control	Fire Detection and Suppression Control	Storage and Transfer Control	G/S Telemetry Comm	Network Operating System	Proximity Operations Monitoring and Contorl	Real-Time Command and Control (Crew, Operations, EVA and Information Systems Management)	Science Experiment Data Collection and Recording	Caution and Warnings Display
Monitor and Support to Operations Control	Waste Heat Acquisition and Transportation	Power Flow and Loading Monitor, Evaluation and Control	Atmosphere Monitoring and Control	Leak Detection, Isolation and Control	V/V Comm	FDIR	Area Traffic Management and Control (Manned and Unmanned)	Simulation and Training	Experiment Data Processing and Control	Crew Control of Platform Systems
Caution and Warning Generation	Load Sharing and Control	Detect, Isolate and Clear Power Faults	Water Recovery Management	Waste Fluid Handling, Vending and Dumping	G/S Detection and Tracking Acquisition	Fault and Redundancy Management	Ground Traversal Planning and Control	Payload Management	Payload Operations Control	Crew User Services (WP, Planning, Spreadsheets, etc.)
GN&C Coordination Control	Leak Detection, isolation and Repair Control	Caution and Warning Generation	EMU Service and Checkout	Caution and Warning Generation	Sensor Control	System Service (DEMS, Network Control, Time Mangement/Control, Data Detection, Transaction Management)	Caution and Warning Generation	Test Management	Video/Imagery Data Management	Navigations Subsystem I/O and Control
Ground Position Control	Data Acquisition and Signal Conditioning	Power Transient Security Control	Airlock Pressure Control	Fuel and Propellant Management and Control	Track Management and Distribution	Access and Security Control	Airlock and Doors Control	EVA Planning	Experiment Managmenet and Control	Crew Override and Control of GN&C Functions
Ground Navigation Planning and Control	Caution and Warning Generation	Solar Array Management	Gasses Control		Video Comm	Caution and Warning Control	OMA Control	Resource Management and Service (Crew and Equipment)		
Motor Control	Thermal Balance and Conditioning Control	Thermal Reactor Management	Special Environment Control		Audio Management and Comm	System Configuration Management	EVA Control	Quarantine Monitoring and Management		
	FDIR	Ion Reactor Management	Contamination Detection and Removal Control		C&T Monitoring and Status	Re(Init) and Shutdown		Equipment Stowage Management		
		Ion Solar Management	Caution and Warning Generation		Surface Weather Monitoring	User I/F Management		Perishible Inventory Management		
		Surface Reactor Management	Environment Monitoring and Management		Caution and Warning Generation	(Over)Load Control		Housekeeping and Trash Management		
			ISRU Management and Control		FDIR Interface	I/O Handling		Crew Health Management		
					Comm Data Traffic Managmenet	Health Monitoring		Software Configuration Management		
					Crew/Equipment Retrieval Support			Platform Configuration Management		
					Space Weather Monitoring			Onboard Information System		
								Logicstics and Supply Management		
								Maintenance/Service Management		

Appendix C: Software Architecture Viewpoints

Viewpoint Name: Structural

Stakeholders: Software Engineers: Programmers, Testers; Operators

Concerns: Describes the static structure of the system; how the system is decomposed into predefined types of components and connectors. Quality attribute concerns include: Sustainability, Ease of Verification and Validation, "Monitor"-ability

Language, Modeling Techniques, Analytical Methods: ACME

Reviewability → Number of components and connectors
Isolation of Critical Functions
Sustainability → Coupling (CBN, CIM)
Simplicity → Number of States, Modes and Interfaces; Static Dependencies

Viewpoint Name: Dynamic

Stakeholders: Software Engineers: Programmers, Testers; Operators

Concerns: Describes the behavior of the system during run-time. Includes processes, threads, synchronization and scheduling with respect to time.

Language, Modeling Techniques, Analytical Methods:

Assurance of Completeness and Determinsm
Simplicity → Dynamic Dependencies
Real-Time Capabilities → Predictability, Concurrency Handling

Viewpoint Name: Implementation

Stakeholders: Software Engineers, Programmers, Testers

Concerns: Describes the grouping of static entities into modules, or chunks, that are assigned to various programmers or programming groups.

Language, Modeling Techniques, Analytical Methods:

Sustainability → Coupling (CBM, CBMN)
Buildability
"Monitor"-ability → Encapsulation, Information Hiding

Viewpoint Name: Mapping of Software to Hardware

Stakeholders: Software Engineers, Programmers, Avionics Engineers, Commercial Hardware Partners

Concerns: Describes how the implemented software modules will operate on computing system hardware; there exists a mapping between the software and hardware designs.

Language, Modeling Techniques, Analytical Methods:
Hardware Technology Forecasting
Hardware Availability → Memory, Processing Speed
Programming Language – Hardware Interface

Viewpoint Name: Information and Data Flow

Stakeholders: Testers, Operators, Enterprise Stakeholders

Concerns: Describes how information and data flow through the system to and from external devices (human, machine or environment), which is the key to value delivery.

Language, Modeling Techniques, Analytical Methods:
"Monitor"-ability → Information Hiding
Sustainability

Appendix D: Cost and Risk Evaluation Criteria

Are there hard or soft real time constraints?	Hard	Is the software computationally intensive?	High	How many interfaces are required?	High
	Soft		Medium		Medium
	None		Low		Low
Is a human override or backup function used?		What are the volatile storage requirements?		What is the average breadth of these interfaces?	
	Yes		High		Large
	No		Medium		Medium
			Low		Small
How precedented is the software? What TRL Level is the software?		What are the non-volatile storage requirements?		What are the environmental considerations?	
	9		High		Proximity to other unmanned vehicles
	8		Medium		Proximity to other manned vehicles
	7		Low		Lunar surface environment
	6	How many control modes are in the process model?			Mars surface environment
	5		High		LEO space environment
	4		Medium		In-transit to Moon space environment
	3		Low		In-transit to Mars space environment
	2	How many state variables are in the process model?			Low-moon orbit space environment
	1		High		Low-mars orbit space environment
Is Ultra-High Reliability required?			Medium		Docked to other unmanned vehicles
	Yes		Low		Docked to other manned vehicles
				For each environmental consideration, what is the impact?	
	No				
Is the software deterministic?		How many functions are required?			High
	Yes		High		Medium
	No		Medium		Low
				Are there software actions that are safety-critical?	
			Low		
Are there multiple algorithms that can be used for a backup system?		How many decisions are required?			
					Yes
	Yes		High		No
	No		Medium	Does the mitigation strategy involve additional software?	
			Low		Yes
		Is there a reliable source for information update?			No
Is the software distributed?			Yes	If the software actions can impact hazards, are the mitigation strategies aimed at elmination, prevention, control, damage reduction?	
	Yes		No		Elimination
	No				Prevention
					Control
					Damage Reduction

Table D-1. Software Questionnaire

Is the human the primary controller of the system?	Yes	If a human override or backup function is used, is incremental control possible?	Yes	Is there a user interface?	Yes
	No		No		No
What is the margin between the propagation delay between the controller and the controlled system and the deadline?		Does the system provide feedback?		Is there a reliable source for information update?	
	Small		Yes		Yes
	Medium		No		No
	Large				
What type of response is needed by the controller?	Simple	What is the workload (Time Load) associated with the task?	High		Proximity to other unmanned vehicles
	Choice		Medium		Proximity to other manned vehicles
	Decision-making		Low		Lunar surface environment
					Mars surface environment
		What is the workload (Mental Effort Load) associated with the task?	High	What are the environmental considerations?	LEO space environment
			Medium		In-transit to Moon space envirornment
			Low		In-transit to Mars space environment
					Low-moon orbit space environment
		What is the workload (Stress Load) associated with the task?	High		Low-mars orbit space environment
			Medium		Docked to other unmanned vehicles
Is Ultra-High Reliability required?	Yes		Low		Docked to other manned vehicles
	No	What is the length of the task?		For each environmental consideration, what is the impact?	
			Prolonged		High
What is the situation awareness level required by the human in the loop?	Level 1 - Perception of Elements in Current Situation		Medium		Medium
	Level 2 - Comprehension of Current Situation		Short		Low
	Level 3 - Projection of Future Status				
		How many control modes are in the process model?	High	Are there human actions that are safety-critical?	Yes
If a human override or backup function is used, what amount of infrastructure is needed to support the controller?	Substantial - New infrastructure. Major adaptations to existing infrastructure		Medium		No
	Moderate - No new infrascturcture. Adaptations to existing infrastructure		Low	Does the mitigation strategy involve additional human procedures or actions?	
	Minimal - Minor adaptations to existing infrastructure	How many state variables are in the process model?			Yes
	None		High		No
What is the skill level required to perform the task? How much training is required?			Medium	If the human actions can impact hazards, are the mitigation strategies aimed at elmination, prevention, control, damage reduction?	
	High		Low		Elimination
	Medium				Prevention
	Low				Control
					Damage Reduction

Table D-2. HCI Questionnaire

Question	Rendezvous and Docking Control				Crew Override and Control of GN&C Functions			
		Development Cost	Development Risk	Mission Risk		Development Cost	Development Risk	Mission Risk
Are there hard or soft real time constraints?	Hard	6	6	8	Hard	6	6	8
Is a human override or back-up function used?	Yes	6	6	0	No	6	6	0
How precedented is the software?	TRL 3	9	9	9	TRL 9	0	0	0
Is the software deterministic?	Yes	0	0	0	Yes	0	0	0
Are there multiple algorithms that can be used for a back-up system?	Yes	10	10	0	No	0	4	0
Is the software distributed?	No	0	0	0	No	0	0	0
What are the CPU cycle requirements?	High	4	4	4	Medium	2	2	2
What are the volatile storage requirements?	High	4	4	4	Medium	2	2	2
What are the non-volatile storage requirements?	Medium	2	2	2	Medium	2	2	2
How many functions are required?	High	6	6	6	Medium	3	3	3
How many decisions are required?	Medium	3	3	3	Low	0	0	0
Is there a reliable source for information update?	Yes	0	0	0	Yes	0	0	0
How many interfaces are required?	High	5	5	5	Medium	2	2	2
What is the average breadth of these interfaces?	Medium	10	10	10	Medium	10	10	10
Is the human the primary controller of the system?	N/A	N/A	N/A	N/A	No	0	0	0
What is the margin between the propagation delay between the controller and the controlled system and the deadline?	N/A	N/A	N/A	N/A	Large	0	1	0
What type of response is required from the controller?	N/A	N/A	N/A	N/A	Decision-Making	0	2.5	0
What is the situation awareness level required by the human in the loop?	N/A	N/A	N/A	N/A	Level 3	7	0	7
What amount of infrastructure is needed to support the controller?	N/A	N/A	N/A	N/A	Major	8	4	0
What is the skill level required to perform the task?	N/A	N/A	N/A	N/A	High	8	6	8
Is incremental control possible?	N/A	N/A	N/A	N/A	No	0	0	4
Does the system provide feedback?	N/A	N/A	N/A	N/A	Yes	0	0	0
What is the workload (Time Load) associated with the task?	N/A	N/A	N/A	N/A	High	0	0	6
What is the workload (Mental Effort Load) associated with the task?	N/A	N/A	N/A	N/A	Medium	0	0	2
What is the workload (Stress Load) associated with the task?	N/A	N/A	N/A	N/A	Medium	0	0	2
What is the length of the task?	N/A	N/A	N/A	N/A	Medium	0	0	4
Is there a user interface?	N/A	N/A	N/A	N/A	Yes	6	2	0
Is there a reliable source for information update?	N/A	N/A	N/A	N/A	Yes	0	0	0
Is Ultra-High Reliability required?	Yes	10	10	0	Yes	10	10	0
How many control modes are in the process model?	Medium	3	3	3	Medium	3	3	3
How many state variables are in the process model?	High	6	6	6	Medium	3	3	3
What are the environnmental considerations?	Proximity to other unmanned vehicles	N/A	N/A	N/A	Proximity to other unmanned vehicles	N/A	N/A	N/A
	Low-Mars orbit space environment	N/A	N/A	N/A	Low-Mars orbit space environment	N/A	N/A	N/A
For each environmental considreation, what is the impact?	High	2	2	6	Low	0	0	0
	Low	0	0	0	Low	0	0	0
Are there software or human actions that are safety-critical?	Yes	10	10	10	Yes	10	10	10
Do the mitigations strategies involve additional software or human actions or procedures?	Yes	10	10	10	N/A	N/A	N/A	N/A
If the software or human actions can impact hazards, are the mitigation strategies aimed at elimination, prevention, control or damage reduction?	Control	4	4	4	N/A	N/A	N/A	N/A

Table D-3. Scores for Example Functions in OPN 969

[This page intentionally left blank.]

References

[1]. Abdurazik, Ayour. "Suitability of the UML as an Architecture Description Language with Applications to Testing." ISE-TR-00-01. Feb. 2000.

[2]. Abowd, Gregory, Len Bass, Paul Clements, Rick Kazman, Linda Northrup and Amu Zaremski. "Recommended Best Industrial Practice for Software Architecture Evaluation." CMU Technical Report CMU/SEI-96-TR-025. Jan. 13, 1997.

[3]. Alexander, Christopher. *The Timeless Way of Building*. Oxford University Press, 1979.

[4]. Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Center for Environmental Structure Series. Oxford University Press, 1977.

[5]. Allen, Robert, David Garlan. "Formalizing Architectural Connection." International Conference of Software Engineering, 1994.

[6]. Avritzer, Alberto and Elaine J. Weyuker. "Investigating Metrics for Architectural Assessment." International Symposium on Software Metrics, Bethesda, MD, 1998.

[7]. Bachmann, Felix, Len Bass, Gary Chastek, Patrick Donohoe and Fabio Peruzzi. "The Architecture Based Design Method." CMU Technical Report CMU/SEI-2000-TR-001. Jan. 2000.

[8]. Baragry, J. And Karl Reed. "Why is it So Hard to Define Software Architecture?" Proceedings of the Asia Pacific Software Engineering Conference, IEEE, CA, 1998.

[9]. Bass, Len, Paul Clements and Rick Kazman. *Software Architecture in Practice*. 2nd Ed. Addison-Wesley, Boston, MA, 2003.

[10]. Bass, Len, Mark Klein and Felix Bachmann. "Quality Attribute Design Primitives and the Attribute Driven Design Method." CMU/SEI-2000-TN-017.

[11]. Bic, Lubomir F. and Alan C. Shaw. *Operating Systems Principles*. Prentice Hall, Upper Saddle River, NJ, 2003.

[12]. Blanchard, Benjamin S.and Wolter J. Fabrycky. *Systems Engineering and Analysis*. 3rd Ed. Prentice Hall, 1998.

[13]. Boehm, Barry. *Software Engineering Economics*. Prentice Hall, 1981.

[14]. Boehm, Barry W. "Software Risk Management: Principles and Practices." IEEE Software, Jan. 1991.

[15]. Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." IEEE Computer, Vol 21, No. 5, May 1988.

[16]. Boehm, Barry and Frank Belz. "Experiences with the Spiral Model as a Process Model Generator." IEEE 1990.

[17]. Boehm, Barry, Alexander Egyed, Julie Kwan, Dan Port, Archita Shah and Ray Madachy. "Using the WinWin Spiral Model: A Case Study." IEEE Computer, July 1998.

[18]. Booch, Grady. *Object-Oriented Analysis and Design with Applications*. Second Ed. Addison-Wesley, Boston, MA, 1994.

[19]. Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, Boston, MA, 1999.

[20]. Bosch, Jan. *Design and Use of Software Architectures.* Addison-Wesley, 2000.

[21]. Bratthall, Lars and Per Runeson. "A Taxonomy of Orthogonal Properties of Software Architectures." 1999.

[22]. Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering.* Anniversary Edition. Addison Wesley Longman, Boston, MA, 1995.

[23]. Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley & Sons, 1996.

[24]. Budgen, David. "Software Design Methods: Life Belt or Leg Iron?" IEEE Software, Sept./Oct. 1999.

[25]. Carnegie Mellon University. "The Acme Architectural Description Language." http://www-2.cs.cmu.edu/~acme. 1998.

[26]. Carnegie Mellon University, Software Engineering Institute. "Software Product Lines." http://www.sei.cmu.edu/productlines/index.html. 2005.

[27]. Carnegie Mellon University, Software Engineering Institute. "Software Architecture for Software-Intensive Systems." http://www.sei.cmu.edu/architecture. 2005.

[28]. Carnegie Mellon University, Software Engineering Institute. "Architecture Evaluations." http://www.sei.cmu.edu/architecture/ata_eval.html. 2005.

[29]. Checkland, Peter. *Systems Thinking, Systems Practice.* John Wiley & Sons, 1981.

[30]. Clements, Paul, Rick Kazman and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley, Boston, MA, 2002.

[31]. Clements, Paul and Linda Northrup. *Software Product Lines: Practices and Patterns.* Addison-Wesley, Boston, MA, 2002.

[32]. Crawley, Edward, Olivier de Weck, Steven Eppinger, Christopher Magee, Joel Moses, Warren Seering, Joel Schindall, David Wallace and Daniel Whitney. "The Influence of Architecture in Engineering Systems." Engineering Systems Monograph, Mar. 2004.

[33]. Daouk, Mirna, Nicolas Dulac, Kathryn Anne Weiss, David Zipkin and Nancy Leveson. "A Practical Guide To STAMP-Based Hazard Analysis." 22nd International System Safety Conference, Providence, RI, Oct. 2004.

[34]. Debaud, Jean-Marc, Klaus Schmid. "A Systematic Approach to Derive the Scope of Software Product Lines." International Conference of Software Engineering, Los Angeles, CA, 1999.

[35]. Dijkstra, Edsger W. "The Structure of the 'T.H.E.'- Multiprogramming System." Communications of the ACM, Vol. 11, Iss. 5, 1968.

[36]. Dorbica, Liliana, Eila Niemelä. "A Survey on Software Architecture Analysis Methods." IEEE Transactions on Software Engineering, Vol. 28, No. 7, July 2002.

[37]. Draper-MIT Team. Concept Exploration and Refinement Final Report for the Exploration Initiative. Oct. 2005.

[38]. Dulac, Nicolas and Nancy Leveson. "An Approach to Design for Safety in Complex Systems." International Conference on System Engineering (INCOSE '04), Toulouse, June 2004.

[39]. Dvorak, Daniel. "Challenging Encapsulation in the Design of High-Risk Control Systems." 17th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, Seattle, WA, Nov. 2000.

[40]. Dvorak, Daniel, Robert Rasmussen, Glenn Reeves, Allan Sacks. "Software Architecture Themes in JPL's Mission Data System." IEEE Aerospace Conference, Mar. 2000.

[41]. Fairley, Richard E. *Software Engineering Concepts*. Mcgraw-Hill Series in Software Engineering and Technology. McGraw Hill, 1985.

[42]. Fayad, Mohamed and Douglas C. Schmidt. "Object-Oriented Application Frameworks." Communications of the ACM, Vol. 40, No. 10, Oct. 1997.

[43]. Feather, Martin S., Lorraine M. Fesq, Michel D. Ingham, Suzanne L. Klein and Stacy D. Nelson. "Planning for V&V of the Mars Science Laboratory Rover Software." IEEEAC Paper #1386, Ver. 4, Dec. 2003.

[44]. Eickelmann, Nancy S. and Debra J. Richardson. "What Makes One Software Architecture More Testable Than Another?" ACM Sigsoft Workshop, San Francisco, CA, 1996.

[45]. Euler, Edward A., Steven D. Jolly, and H.H. 'Lad' Curtis. "The Failures of the Mars Climate Orbiter and Mars Polar Lander: A Perspective from the People Involved." *Advances in the Aeronautical Sciences, Guidance and Control.* Volume 107, 2001.

[46]. Fukuzawa, Kimijuki and Motoshi Saeki. "Evaluating Software Architectures by Coloured Petri Nets." International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, 2002.

[47]. Gabriel, Richard P. *Patterns of Software : Tales from the Software Community*. Oxford University Press, 1998.

[48]. Gamble, Edward B., Jr. and Reid Simmons. "The Impact of Autonomy Technology on Spacecraft Software Architecture: A Case Study." IEEE Intelligent Systems, September/October 1998, pp. 69 - 75.

[49]. Garlan, David. "Research Directions in Software Architecture." ACM Computing Surveys, June 1995, Vol. 27, No. 2.

[50]. Garlan, David. "Software Architecture: A Roadmap." ACM Future of Software Engineering, Limerick, Ireland. 2000.

[51]. Garlan, David and Mary Shaw. "An Introduction to Software Architecture." 1994.

[52]. Garlan, David, Robert Monroe, and David Wile. "Acme: An Architecture Description Interchange Language." Proceedings of CASCON, Toronto, Nov. 1997.

[53]. Garlan, David, Robert Allen, and John Ockerbloom. "Architecture Mismatch, or Why it's hard to build systems out of existing parts." International Conference on Software Engineering, Seattle, WA, Apr. 1995.

[54]. Gat, Erann. "The MDS Autonomous Control Architecture." Jet Propulsion Laboratory. Pasadena, CA.

[55]. Ghezzi, Carlo, Mehdi Jazayeri and Dino Mandrioli. *Fundamentals of Software Engineering*. 2nd Ed.

[56]. Harsu, Maarit. "A Survey of Product-Line Architectures."

[57]. Hatton, Les. "Does OO Sync with How We Think?" IEEE Software, May/Jun. 1998.

[58]. Hayhurst, Kelly J. and C. Michael Holloway. "Considering Object Oriented Technology in Aviation Applications." Proceedings of the 23rd Digital Avionics Systems Conference. 2003.

[59]. Honeywell Corporation. "Languages and Tools for Embedded Software Architectures." http://www.htc.honeywell.com/projects/dssa/dssa_tools.html.

[60]. IEEE. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std. 1471-2000. Sept. 2000.

[61]. Ingham, Michel D., Robert D. Rasmussen, Matthew B. Bennett, Alex C. Moncada. "Generating Requirements for Complex Embedded Systems Using State Analysis." 55th Congress of the International Astronautical Federation, Vancouver, Canada, October 2004, IAC-04-IAF-U.3.A.05.

[62]. Ingham, Michel D., Robert D. Rasmussen, Matthew B. Bennett, Alex C. Moncada. "Engineering Complex Embedded Systems with State Analysis and the Mission Data System." 1st AIAA Intelligent Systems Technical Conference, Chicago, IL, Sept. 2004.

[63]. Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Boston, MA, 1999.

[64]. Jet Propulsion Laboratory. "Remote Agent Experiment – Deep Space 1 Technology Validation Symposium." Pasadena, CA, Feb. 2000.

[65]. Kazman, Rick, Paul Clements and Len Bass. "Classifying Architectural Elements as a Foundation for Mechanism Matching." Proceedings of the 21st Annual International Computer Software and Applications Conference, Los Alamitos, CA, 1997.

[66]. Kazman, R., M. Klein and P. Clements. "Evaluating Software Architectures for Real-Time Systems." Carnegie Mellon University and the Software Engineering Institute, 1998.

[67]. Kazman, Rick, Mario Barbacci, Mark Klein and S. Jeromy Carriere. "Experience with Performing Architecture Tradeoff Analysis." International Conference on Software Engineering, Los Angeles, CA, 1999.

[68]. Kazman, Rick, Jai Asundi, and Mark Klein. "Quantifying the Costs and Benefits of Architectural Decisions." International Conference on Software Engineering, Toronto, May 2001.

[69]. Kerth, Normal L. and Ward Cunningham. "Using Patterns to Improve Our Architectural Vision." IEEE Software, Jan. 1997, pp. 53 - 59.

[70]. Koopman, Phillip J., Jr. "A Taxonomy of Decomposition Strategies Based on Structures, Behaviors, and Goals." Design Engineering Technical Conferences, Vol. 2, 1995.

[71]. Krutchen, Philippe. "Architectural Blueprints - The "4+1" View Model of Software Architecture." IEEE Software, Nov. 1995, Vol. 12, No. 6.

[72]. Kuloor, Chethana, Armin Eberlein. "Requirements Engineering for Software Product Lines."

[73]. Leopold, Ray. Personal Communication. May 2004.

[74]. Leveson, Nancy G. *Safeware: System Safety and Computers*. Addison-Wesley Publishing Company, Reading, MA, 1995.

[75]. Leveson, Nancy G., Denise Pinnel, Sean David Sandys, Shuichi Koga, Jon Damon Reese. "Analyzing Software Specifications for Mode Confusion Potential." Proceedings of the Workshop on Human Error and System Development, Glasgow, March 1997.

[76]. Leveson, Nancy G. "Intent Specifications: An Approach to Building Human-Centered Specifications." IEEE Transactions on Software Engineering, Jan. 2000.

[77]. Leveson, Nancy and Kathryn Anne Weiss. "Making Embedded Software Reuse Practical and Safe." Proceedings of Foundations of Software Engineering, Nov., 2004.

[78]. Leveson, Nancy. *A New Approach to System Safety Engineering*, Incomplete Draft (downloadable from http://sunnyday.mit.edu/book2.html).

[79]. Leveson, Nancy, Mirna Daouk, Nicolas Dulac, and Karen Marais. "Applying STAMP in Accident Analysis." Workshop on Investigation and Reporting of Incidents and Accidents (IRIA), Sep. 2003.

[80]. Lindvall, Mikael, Roseanne Tesoriero Tvedt and Patricia Costa. "An Empirically-Based Process for Software Architecture Evaluation." Empirical Software Engineering: An International Journal, Kluwer, 2002.

[81]. Lions, J.L., et al. *Ariane 5 Flight 501 Failure: Report by the Inquiry Board.* Paris, France. 19 July 1996.

[82]. Maier, Mark W. and Eberhardt Rechtin. *The Art of Systems Architecting.* 2nd Ed. CRC Press, Boca Raton, 2002.

[83]. Mankins, John C. "Technology Readiness Levels: A White Paper." Advanced Concepts Office, NASA, 6 Apr 1995.

[84]. Matinlassi, Mari. "Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA." International Conference on Software Engineering, Edinburgh, UK, May 2004.

[85]. Matinlassi, Mari. "Evaluation of Product Line Architecture Design Methods." 7th International Conference on Software Reuse, Young Researchers Workshop, Austin, TX, Apr. 2002.

[86]. Matinlassi, Mari. Eila Niemelä, Liliana Doborica. "Quality-Driven Architecture Design and Quality Analysis Method – A Revolutionary Initiation Approach to a Product Line Architecture." Technical Research Centre of Finland, 2002.

[87]. Medvidovic, Nenad and Richard N. Taylor. "A Classification and Comparison Framework for Software Architecture Description Languages." IEEE Transactions on Software Engineering, Vol. 26, No. 1, Jan. 2000.

[88]. Mellor, Stephen J. and Ralph Johnson. "Why Explore Object Methods, Patterns, and Architectures?" IEEE Software, Jan. 1997.

[89]. Meyer, Bertrand. "A Really Good Idea." IEEE Computer, Dec. 1999.

[90]. Meyer, Marc H. and James M. Utterback. "The Product Family and the Dynamics of Core Capability." Sloan Management Review, Spring 1993.

[91]. Meyer, Marc H., Robert Seliger. "Product Platforms in Software Development." Sloan Management Review, Vol. 40, No. 1, Fall 1998.

[92]. Moriconi, Mark. "Correct Architecture Refinement." IEEE Transactions on Software Engineering Special Issue on Software Architecture." Apr. 1995.

[93]. Moynihan, Tony. "An Experimental Comparison of Object-Oriented and Functional-Decomposition as Paradigms for Communicating System Functionality to Users." J. Systems Software 1996.

[94]. Mustapic, Goran. "Architecting Software For Complex Embedded Systems – Quality Attribute Based Approach to Openness." Mälarden University Licentiate Thesis, No. 38. Sweden, 2004.

[95]. NASA/ESA Investigation Board. "SOHO Mission Interruption." 31 Aug 1998.

[96]. National Aeronautics and Space Administration. "The Vision for Space Exploration." http://www.nasa.gov. Feb. 2004.

[97]. Niemelä, Eila, Tuomas Ihme. "Product Line Software Engineering of Embedded Systems." ACM SIGSOFT Software Engineering Notes, Vol. 26, No. 3, May 2001.

[98]. Niemelä, Eila, Mari Matinlassi and Anne Taulavuori. "A Practical Evaluation of Software Product Family Architectures." 3rd International Conference on Software Product Lines. Aug. 2004.

[99]. Oman, Charles. "Spatial Orientation and Situation Awareness." Class Notes, !6.400 Human Factors Engineering, MIT, Nov 2002.

[100]. Ong, Elwin and Nancy G. Leveson. "Fault Protection in a Component-Based Spacecraft Architecture." Proceedings of the International Conference on Space Mission Challenges for Information Technology, Pasadena, July 2003.

[101]. Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules." Communications of the ACM, Vol. 15, No. 12, Dec. 1972.

[102]. Perry, Dewayne E. and Alexander L. Wolf. "Foundations for the Study of Software Architecture." ACM Software Engineering Notes, Oct. 1992, Vol. 17, No. 4.

[103]. Pimmler, Thomas U. and Steven D. Eppinger. "Integration Analysis of Product Decompositions." Design Theory and Methodology, 1994.

[104]. Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. Multiple Versions.

[105]. Rasmussen, Robert D. "Goal-Based Fault Tolerance for Space Systems Using the Mission Data System." *IEEE*. 2001.

[106]. Rechtin, Eberhardt. *Systems Architecting*. Prentice Hall, 1990.

[107]. Reeves, Glenn, Tracy Neilson and Todd Litwin. "Mars Exploration Rover Spirit Vehicle Anomaly Report." Jet Propulsion Laboratory MER 420-6-785. May 12, 2004.

[108]. Regan, Patrick, and Scott Hamilton. "NASA's Mission Reliable." IEEE Computer, Jan. 2004, pp. 59 - 68.

[109]. Robertson, David, Karl Ulrich. "Planning for Product Platforms." Sloan Management Review Vol. 39, No. 4, Summer 1998.

[110]. Russel, Stuart and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Saddle River, NJ, 1995.

[111]. Safeware Engineering Corporation. "SpecTRM User Manual." 2001.

[112]. Schmidt, Douglas C., Mohamed Fayad and Ralph E. Johnson. "Software Patterns." Communications of the ACM, Oct. 1996, Vol. 39, No. 10.

[113]. Schmidt, Douglas C. and Frank Buschmann. "Patterns, Frameworks, and Middleware: Their Synergistic Relationships." International Conference on Software Engineering, Toronto, Canada, 2003.

[114]. Shaw, Alan C. *Real-Time Systems and Software.* John Wiley & Sons, Inc., New York, 2001.

[115]. Shaw, Mary. "Comparing Architectural Design Styles." IEEE Software, Nov. 1994.

[116]. Shaw, Mary, et. al. "Candidate Model Problems in Software Architecture." http://www.cs.cmu.edu/~ModProb/. Version 1.3, Jan. 1995.

[117]. Shaw, Mary. "Abstractions for Software Architecture and Tools to Support Them." IEEE Transactions on Software Engineering Special Issue on Software Architecture. Apr. 1995.

[118]. Shaw, Mary and Paul Clements. "Toward Boxology: Preliminary Classification of Architectural Styles." ACM Sigsoft Workshop, San Francisco, CA, 1996.

[119]. Shaw, Mary and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, Upper Saddle River, NJ, 1997.

[120]. Shaw, Mary. "The Coming-of-Age of Software Architecture." International Conference of Software Engineering, Toronto, Canada, 2003.

[121]. Shereshevsky, Mark, Habib Ammari, Nicholay Gradetsky, Ali Mili, and Hany H. Ammar. "Information Theoretic Metrics for Software Architectures." International Computer Software and Applications Conference, Chicago, IL, 2001.

[122]. Sommerville, Ian. "Software Process Models." ACM Computing Surveys, Vol. 28, No. 1, Mar. 1996.

[123]. Soni, D., Robert L. Nord and Christine Hofmeister. "Software Architecture in Industrial Applications." International Conference of Software Engineering, Seattle, WA, 1995.

[124]. Stolper, Steven A. "Streamlined Design Approach Lands Mars Pathfinder. IEEE Software, Vol. 16, Issue 5, September/October, 1999, pp. 52 - 62.

[125]. Suh, Nam P. *Design with Systems.* Oxford University Press, NC, 2001.

[126]. Taulavuori, Anne, Eila Niemelä and Päivi Kallio. "Component Documentation – A Key Issue in Software Product Lines." Information and Software Technology Vol. 46, No. 8, 2004.

[127]. Tepfenhart, William M. and James J. Cusick. "A Unified Object Topology." IEEE Software, Jan. 1997, pp. 31 - 35.

[128]. Ulrich, Karl T. and Steven D. Eppinger. *Product Design and Development.* 3rd Ed. McGraw Hill, Boston, 2004.

[129]. Van der Linden, Frank J. and Jürgen K. Müller. "Creating Architectures with Building Blocks." IEEE Software, Nov. 1994.

[130]. Van Vliet, Hans. *Software Engineering Principles and Practice.* 2nd Ed. Wiley, New York, NY, 2001.

[131]. Weiss, Kathryn Anne. "Component-Based Systems Engineering for Autonomous Spacecraft." MIT Master's Thesis, 2003.

[132]. Wertz, J.R. and W.J. Larson. *Space Mission Analysis and Design*. Microcosm Press, 1999.

[133]. White, John R. and Taylor L. Booth. "Towards an Engineering Approach to Software Design." International Conference on Software Engineering, San Francisco, CA, Oct. 1976.

[134]. Wickens, Christopher D. and Justin G. Hollands. *Engineering Psychology and Human Performance*. 3rd Ed. Prentice Hall, Upper Saddle River, NJ, 2000.

[135]. Williams, Lloyd G. and Connie U. Smith. "Performance Evaluation of Software Architectures." WOSP, Santa Fe, NM, 1998.

[136]. Wray, Richard B. and John R. Stovall. "Space Generic Open Avionics Architecture (SGOAA) Reference Model Technical Guide." Lockheed Engineering and Science Company, Houston, TX, Apr 1993.

[137]. Zelesnik, Gregory. "The UniCon Language Reference Manual." http://www-2.cs.cmu.edu/afs/cs/project/vit/www/UniCon/reference-manual/Reference_Manual_1.html. May 1996.

[138]. Zhao, Jianjun. "On Assessing the Complexity of Software Architecture." Proceedings of the Third International Workshop on Software Architecture, Orlando, FL, 1998.

[139]. Zimmerman, Marc, Kristina Lundqvist, and Nancy Leveson. "Investigating the Readability of State-Based Formal Requirements Specification Languages." Proceedings of the International Conference on Software Engineering, Orlando, FL, May 2002.