REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
December 1991

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
Requirements Analysis Notebook for the Flight Data Systems Definition in the Real-time Systems Engineering Laboratory (RSEL)

5. FUNDING NUMBERS

6. AUTHOR(S)
Richard B. Wray

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Lockheed Engineering and Sciences Company
2400 Nasa Road 1
Houston, Texas 77573-3799

8. PERFORMING ORGANIZATION REPORT NUMBER
LESC-29702

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Flight Data Systems Division
Houston, Texas 77058

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CR 185698

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 66

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A hybrid requirements analysis methodology has been developed, based on the practices actually used in developing a Space Generic Open Avionics Architecture. During the development of this avionics architecture, a method of analysis able to effectively define the requirements for this space avionics architecture was developed. In this methodology, external interfaces and relationships are defined, a static analysis resulting in a static avionics model was developed, operating concepts for simulating the requirements were put together, and a dynamic analysis of the execution needs for the dynamic model operation was planned. The systems engineering approach was used to perform a top down modified structured analysis of a generic space avionics system and to convert actual program results into generic requirements. CASE tools were used to model the analyzed system and automatically generate specifications describing the model's requirements. Lessons learned in the use of CASE tools, the architecture and the design of the Space Generic Avionics model have been established, and a methodology notebook has been prepared for NASA. The weaknesses of standard real-time methodologies for practicing systems engineering, such as Structured Analysis and Object Oriented Analysis, have been identified.

14. SUBJECT TERMS
requirements analysis, space avionics, avionics architecture, methodology, systems analysis

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
Unlimited

DOCUMENT CHANGE RECORD

The following table summarizes the change activity associated with this document.

ISSUE AND DATE	CHANGE SUMMARY	SECTION
January 6, 1992	Pages 2-2 through 2-10 - The basic methodology was expanded to include operating concepts development, dynamic analysis and fidelity checking.	2
	Pages 3-6 and 3-15 - Figures were upgraded to show concurrent processes.	3

ACKNOWLEDGEMENTS

This document has been produced by Mr. Richard B. Wray of the Lockheed Engineering & Sciences Company (LESC), with significant inputs from several other people. Special acknowledgement is given to Mr. Ben Doeckel of LESC for producing the appendices on naming conventions, in codeveloping the avionics architecture on which this document is based, and in contributing many ideas for methodology alternatives. Acknowledgement is also given to Mr. Jack Sassard, Mr. John Stovall and Mr. Graham O'Neil of LESC for assisting in the development of the avionics architecture and constructive criticisms of the methodology.

CONTENTS

Section Page

Section Page

Section Page

APPENDICES

TABLES

Table Page

FIGURES

Figure Page

ACRONYMS

AP	Application Platform
API	
AS	Application Software
BIT	Built-in-Test
C&T	Communications and Tracking
CASE	Computer Aided Systems Engineering
CRC	Control and Reporting Center
DB	Docking/Berthing
DMS	Data Management Systems
EE	External Environment
EEI	External Environment Interface
EP	Effector Embedded Processor
F2D	Functional Flow Diagrams
FDIR	Fault Detection, Isolation and Recovery
LESC	Lockheed Engineering & Sciences Company
GN&C	Guidance, Navigation and Control
HOOSA	Hybrid, Object Oriented, Structured Analysis
JSC	Johnson Space Center
MDP	Multiplex Data Processor
NASA	National Aeronautics and Space Administration
OMS	Operations Management System
OO	Oribt Transfer-to-New Orbit
OOA	Object Oriented Analysis
OS	Orbit-to-Surface
OSE	Open Software Environment
OSI	Open Systems Interconnect
OSK	Orbit Station Keeping

PDL	Program Design Language
RSEL	Real-time Systems Engineering Laboratory
SATWG	Strategic Avionics Technology Working Group
SDP	Standard Data Processing
SDSS	Space Data System Services
SGA	Space Generic Avionics
SP	Sensor Processor
SO	Surface-to-Orbit
SOCS	
TBD	To Be Determined
VHDL	Very High Description Language
VHSIC	Very High Speed Integrated Circuits

1. INTRODUCTION

The systems engineering methodology is a set of methods being developed by Lockheed Engineering and Sciences Company which are applicable to systems development over the entire life cycle of the system. The system life cycle includes:

- Phase A: Conceptual definition of a system within a larger picture of needs (i.e., a strategic view of the need for a system)
- Phase B: Requirements definition of a system which must produce complete and consistent requirements.
- Phase C: Design development for a system.
- Phase D: Operations and support for a system.

This methodology notebook covers Phase A, Conceptual Definition and Phase B, Requirements Definition, and will be developed over several years in an on-going effort. The other phases of the system life cycle are identified here to establish an improved contextual understanding of how all analyses tie together to perform systems engineering.

1.1 PURPOSE

Previous space programs have usually relied upon one space vehicle in development or flight at a time. In the future, it is likely that multiple space vehicles will be in development or in flight at the same time. In order to be able to afford the development of these space vehicles, new and evolutionary approaches to the design of these vehicles must be developed. Avionics systems are a prime candidate for the development of evolutionary approaches as there is much commonality between the functions that must be provided for all space vehicles. This study was begun to develop a generic methodology for defining avionics architectures that could be tailored to match the varying requirements of all space vehicles without requiring the system engineering team for each new vehicle to reinvent the requirements analysis and design process.

A methodology is defined as a set of methods, techniques and tools for performing a defined task. Achievement of such a methodology would be both cost and time (schedule) efficient for all future space programs using the methodology. This document is the first step in providing such a methodology for requirements analysis and design. It identifies a set of alternative static approaches, techniques and types of automated tools which can facilitate the difficult task of requirements analysis and design within the overall systems engineering process. Later revisions of this document will add consideration of dynamic approaches, techniques and tools; additional requirements analysis issues; and conversion from the requirements analysis and specification process to the design process.

This study is part of a multi-year effort by Lockheed Engineering and Sciences Company to define an overall methodology, using a generic architecture for avionics, which can be applied to all future manned space mission avionics. The approach of this study is develop the needed methodology by performing an actual requirements analysis and defining selected elements of the advanced avionics architecture for future space missions. The architecture being developed is a superset architecture with all the avionics requirements needed to establish the design of compatible space platform avionics elements.

This section covers the organization of the notebook, the background of the development of this methodology, and key concepts derived from the analysis.

1.2 REPORT ORGANIZATION

This document provides the methodology recommended to perform requirements analysis for space avionics subsystems. Section 1 provides this introduction, covering the purpose of the document; the background leading to its development; the key concepts developed in the course of the preliminary analysis; and the definitions, assumptions, and conventions used in the analysis.

Section 2 provides an overview of the methodology. First, the systems engineering process (based on the Electronic Industries Association Bulletin SYSB-1) is summarized. The overall process of requirements analysis recommended by this document is then detailed, including the basic methodology, the use of tools, the conventional approaches often used, and features of the hybrid approach being recommended. Differences between the requirements and the design features are explained. The remainder of this section is left to be developed (TBD) for later stages of the methodology development in FY92.

Section 3 then applies the methodology to the development of the open generic architecture to provide an example of a specific application of this methodology. The architecture is summarized. Specific usage of the methodology in performing the called for static analysis is shown, and specific techniques needed are described. Additional examples in the remainder of the section are left TBD in later stages of the methodology development in FY92.

Appendices are provided to identify the definitions, assumptions for the architecture and the conventions used in developing this methodology. The appendices also summarize specific findings and guidelines, developed in the course of this task, for the use of CASE tools, requirements documentation accessed, the generic space avionics architecture (published under separate cover as volume 2), and the naming and diagramming conventions which need to be followed.

1.3 BACKGROUND

Johnson Space Center (JSC) initiated the development of this methodology by Lockheed Engineering and Sciences Company to create a design/analysis methodology based on practical experience in developing the requirements for an advanced space systems avionics architecture. The process followed was to define a Space Generic Avionics (SGA) system architecture, record the process by which this architecture was developed, document the informal discussions and the resulting logic/thought processes held on an ad-hoc basis during technical work periods. This architecture was to be a real requirements architecture with utility in any future space program or mission. Over a one year period in FY 1991, data was gathered on existing space programs (i.e., the shuttle and space station) to use as a basis of real developments so the SGA architecture would not be a "blue sky" approach, but would be based on reality. This data was used to build the architecture. Almost every informal meeting, every ad-hoc discussion and every critical or questioning thought was recorded by keeping a log notebook next to the computer. These notes were then compiled into documented notes which were the basis for writing this methodology notebook.

This architecture development monitored efforts by the Strategic Avionics Technology Working Group (SATWG) and its contractors to build generic, standard or open architectures. The results of these efforts were considered in the development of the SGA architecture. The focus of the SGA architecture was a structure which could hold all space avionics requirements and insure avionics subsystem compatibility through compatible requirements. It resulted in a top-down architecture with hooks for every potential avionics subsystem.

This architecture is not a completed product, rather it is a living architecture which can continue to grow as more people support it and more ideas are added to its structure. By this reasoning, the SGA architecture is a space-function oriented architecture which stresses the operational needs, the applications required to satisfy those needs, the applications' requirements for services to enable them to operate, and the allocation of applications and services to hardware and software. It treats hardware and software as secondary to the operational and services aspect of a space avionics system.

While the appearance of the SGA architecture is one of software, it is intended to represent higher level features comprising both hardware and software. Since all space data systems are so heavily dependent on software, this consideration was a primary driver to insure effective software requirements, and especially effective software-to-software and software-to-hardware interface definition.

The leading source of underlying requirements for the SGA architecture (outside of shuttle and station documentation) was the Space Avionics Requirements Study, NASM-37588-TD006, 21 October 1990 by General Dynamics Space Systems Division, as briefed to the SATWG. In particular, the driving requirement for an open architecture allowing multi-vendor sources for components, interchangeable and interoperable elements with reduced complexity and cost and common elements, where feasible, led to the SGA architecture approach of using common space applications relying on common operating services. The requirement for a robust, modular system led to the need to avoid preconceptions on partitioning functions between subsystems, and to build a structure which can facilitate alternative allocations to actual flight missions and facilities. The requirement for handling technology upgrades led to a structure which isolated specific technologies from requirements implementation as much as possible.

1.4 KEY CONCEPTS

Some of the key concepts either developed for this methodology or used for this analysis methodology are described in this section. A number of software and systems development principles were followed in laying out the analysis practice recommended in this methodology, including:

- abstraction
- information hiding
- inheritance
- modularity
- robustness
- extensibility

Abstraction is the principle of using only those aspects of an entity, object, operation, function, process, or other subject which are relevant to the current purpose and ignoring those aspects not needed to improve analytical focus on the current subject. This principle simplifies a complex subject to render it more susceptible to analysis. In object oriented analysis, data abstraction is the principle of defining a data type in term of the operations that apply to the entities of the type.

Information hiding (also called encapsulation) is the principle used in developing system structures where components should encapsulate or hide a single requirements or design decision, with an interface that reveals little of the inner workings of the system. Software information hiding refers to the technique of making the external interface to an entity public, but keeping the internal design details hidden from view. Hiding of the internal design information allows the implementation of the entity to be changed without requiring the external interfaces of the entity to be changed. By hiding the internal implementation, changes are easier to make with minimal rework when system changes are needed.

Inheritance is the principle of receiving properties or characteristics from an ancestor. In systems definition, it allows the specification of common attributes and services only once because they can then be passed to all descendent or referenced subsystems.

A modular requirements architecture is an architecture in which the elements are autonomous, coherent and organized in a robust structure. Requirements must be decomposable, understandable, protected and have continuity. Decomposability means requirements can be broken into smaller pieces with potentially simpler solutions or at least better understanding and a capability for further decomposition as needed. Understandability means all requirements related to a subject can be found and viewed together, and individually and jointly understood by the analysts and designers. Protection means the architecture limits the effect of abnormal conditions in design elements at run-time to just the affected modules or at least will limit the propagation of abnormal conditions. Continuity means that small changes in the requirement will only cause small changes in the design. Requirements changes should not cause disproportional changes in the design, and design changes should be limited to one or a few design modules. Requirements changes should not affect the architecture of the system, unless the change is one for the architecture directly.

Robustness is the ability of systems to continue functioning in abnormal conditions. This principle clearly relates to reliability, fault tolerance and fail-safe/operate concepts. Robustness is more, however, because it addresses the system's ability to operate in conditions not originally foreseen by the specification without catastrophic failures, without exhibiting behavior that disturbs the rest of the system, by failing (if necessary) in a "graceful" manner by terminating cleanly and safely.

Extensibility is the ability of a system to be extended or adapted to new conditions, changes in specifications or new technology. Extensibility is facilitated by simple requirements and designs where feasible, minimal complexity (for its own sake) and decentralization.

As a result of the analysis leading to this methodology, several key concepts and findings for the methodology and for the architecture used as a case example were established. These are described below.

1.4.1 METHODOLOGY CONCEPTS AND FINDINGS

The systems engineering process applies to all phases of the life cycle for a space program, from concept development, to requirements definition, design and operation. The major element needed for an effective system to operate is well established requirements, which are needed in each of the phases of the life cycle. A methodology has been established in this project, through actual development of an architecture, to develop good requirements. This methodology is called the Hybrid, Object Oriented, Structured Analysis methodology. It blends several existing methodologies into an approach which has been shown to work in practice in defining the requirements for a major system architecture. The weakness of existing methodologies (such as Yourdan, Ward-Mellor, etc.) is that they all seem to take an academic, simplistic view of systems (e.g., an Air Traffic Control system consisting of a radar, planes, missions, and airports) with no technical backup, or a technical view of a very small system (such as a screen window) with indepth technical backup but no relevance to a larger system. Putting them into practice with a realistic large scale system is ***very*** difficult.

Requirements analysis needs automated tool support. The tools need to provide basic structured analysis capabilities, centered on an integrated data repository. A goal in picking automated tools should be the quality of their support in enabling this methodology to be implemented. A selection factor should be the accessibility of their data repository to other tools so that data from one tool can be extracted and passed to another tool; preferably by using open standards for the data repository structure. No weight should be given to a tool claiming to be capable of performing all phases of automated development, since such a claim is far beyond the state-of-the-art in present tools, and may not be desirable anyway. It seems likely that the tools are secondary to the quality of the analysts in performing requirements analysis; if so, then the tools should be selected to enhance the abilities of individual analysts.

The state-of-the-art, analytical techniques to be used require training to gain understanding of static structured analysis, interface analysis, information modeling, object oriented analysis, control state analysis, timeline analysis, performance analysis, dynamic system modeling, and others. Analysts with experience in all these techniques are not common, and efforts should be established to train experienced systems engineers in the additional techniques called for in this methodology. One of the requirements for such training will be for the Training Department to budget and bring in experienced training contractors to supplement their in-house capability. Another key finding is that requirements definition is part of a concurrent engineering approach, and will require the integrated efforts of engineers with experience in several different disciplines, from requirements to design, from hardware to software, and from operation to supportability. Development of an effective requirements model needs iteration between the concerns of each specialty to insure that the resulting model is responsive to all discipline concerns. Effective techniques to enable multiple disciplines to efficiently interact need to be developed.

Development interface capabilities are needed. Much of the development of requirements for complex space systems is taking place at geographically disbursed contractor sites; these distributed requirements need to be capable of being coordinated on a continuous basis as they are developed. A capability is needed to acquire working level requirements (in process) from these sites, test them against each other and a larger model of the system, and to feed back weaknesses and strengths to the developers of individual requirements sets. This is necessary to avoid waiting too long before erroneous, deficient, weak or conflicting requirements are uncovered; the later the correction of requirements, the greater the cost. This implies some data repository structure or interface standards are needed to insure that requirements data can be exchanged. It also implies that more frequent technical interchanges would benefit the development of subsystem requirements, especially if the interchanges could be working level with minimal preparation time needed to "pretty up" briefings for "dog and pony" shows.

The development of this methodology is still underway. Additional parts of the methodology are needed to define complex system requirements completely and consistently. The parts to be developed next include control requirements definition techniques, dynamic analysis/modeling and performance techniques.

1.4.2 ARCHITECTURE CONCEPTS AND FINDINGS

An architecture was used as a target to develop this methodology for requirements analysis. The Space Generic Avionics (SGA) requirements architecture was created to address generic requirements for an architecture of avionics functions, objects, processes, data, and capabilities for any future space vehicle. This architecture had to provide for a generic space avionics based on open standards. It started with logical interface requirements at the top level and expanded these requirements at successively lower levels of definition as the methodology and architecture were developed. If a function is needed for any space vehicle, then it should be adapted from a generic form and available in the architecture. Thus, the architecture is a "shopping list" of all possible avionics elements. This architecture integrates multiple lower level (or sub-architectures) together to develop a consistent structure which incorporates functional architectures from the systems development domain, processing and object oriented architectures from the software domain and hardware architectures from the hardware domain. These paragraphs summarize key points with respect to the SGA architecture. For more detail, see Appendix D to be published in FY92 which will provide more descriptive material and diagrams.

A starting requirement for the architecture was that it be adaptable to all future manned space missions, including:

- Surface-to-orbit missions to reach either low earth orbit, lunar orbit or mars orbit from the local planetary surface.
- Docking and berthing operations between adjacent space platforms such as shuttles arriving at the space station, heavy lift launch vehicles linking with orbiting vehicles, lunar ascent vehicles mating with their orbiting transfer vehicles, etc.
- Orbit station keeping operations by orbiting platforms maintaining stable orbits around a planetary body, such as the space station in earth orbit, the lunar transfer vehicle in moon orbit, or the mars space station in mars orbit.
- Orbit-to-orbit transfers, which may be from low to high earth orbit, from earth to another planet, from earth to the moon, or from earth (or another planet) into deep space.
- Orbit-to-surface missions to land on the earth, moon or mars from an orbiting or arriving space vehicle.
- Fixed surface operations in a fixed base such as on the moon base or mars base, where such a base may be performing permanently manned complex operations or just temporarily manned exploration operations.
- Mobile surface operations in rover or similar vehicles moving on the planet surface.

The SGA Architecture consists of the avionics as a black box surrounded by external elements with which it interacts. The avionics system includes all hardware, software and other electronics needed to control and operate the space vehicle, and provides the coordinated functionality for end-to-end processing in handling the information needed to know the platform's elements, to control its interaction with its environment, and to respond to human commands. This avionics structure is shown in figure 1-1. The SGA Black Box provides the capability to meet the top level user requirements, i.e., those requirements actually serving to represent user needs.

Within the SGA Black Box are the primary functional entities which enable the avionics to support and sustain the crew. These primary functional entities are the traditional applications control subsystems, an operations control application subsystem which integrates all activities from the traditional applications control subsystems to serve the crew, a standard data services subsystem to provide support to all the traditional and operations control subsystems and the crew's display and control subsystem which enable the crew to interface and direct the avionics. The traditional applications control entities consist of Electric Power, Environment and Life Support, Payload Operations, Guidance, Navigation and Control Communications and Tracking Control Subsystems.

The Space Data System Services (SDSS) entity consists of all services and operating system entities supporting the crew's avionics subsystems. Figure 1-2 shows the top level structure of the SDSS architecture.

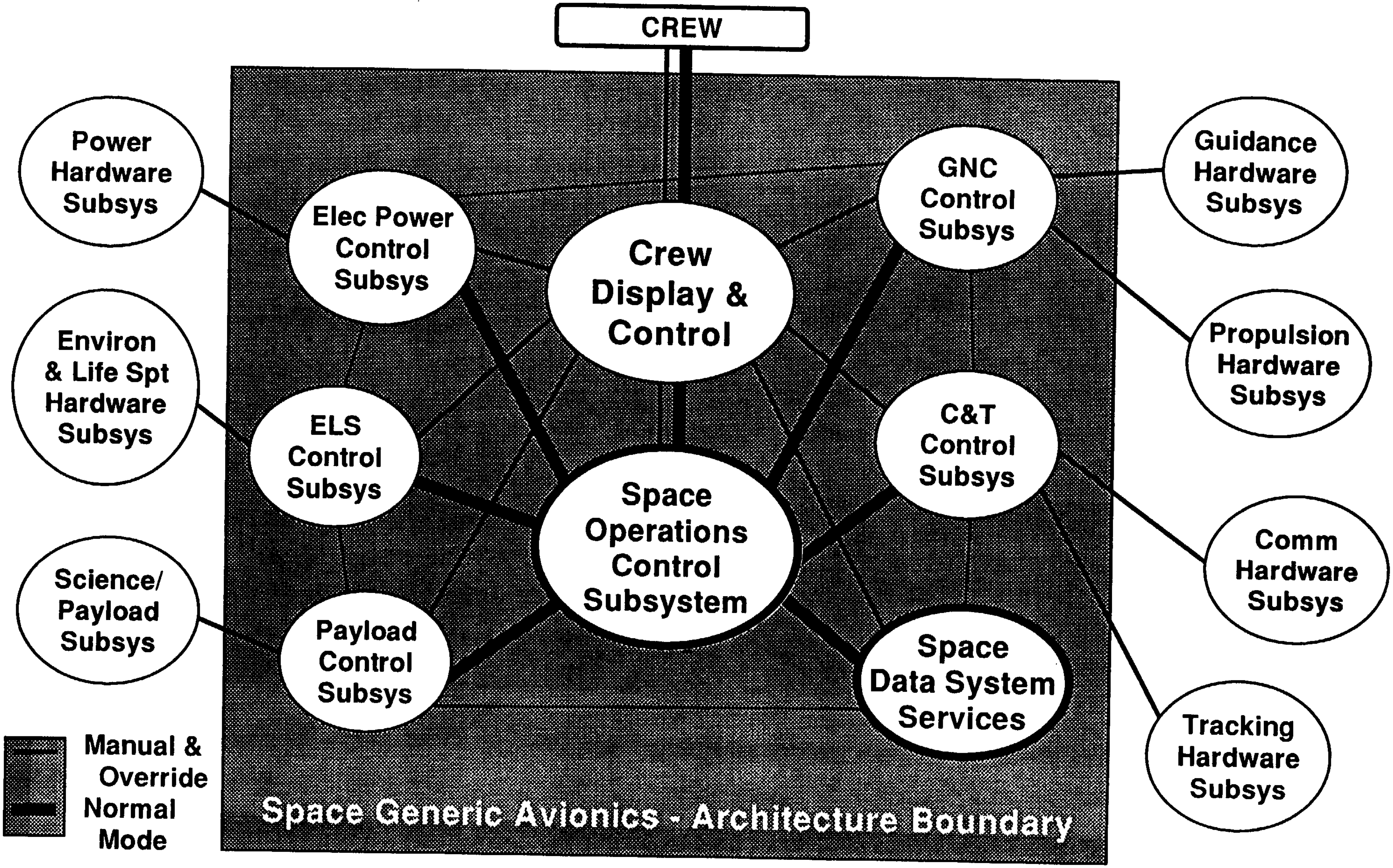


Figure 1-1.- Space Generic Avionics Open Architecture.

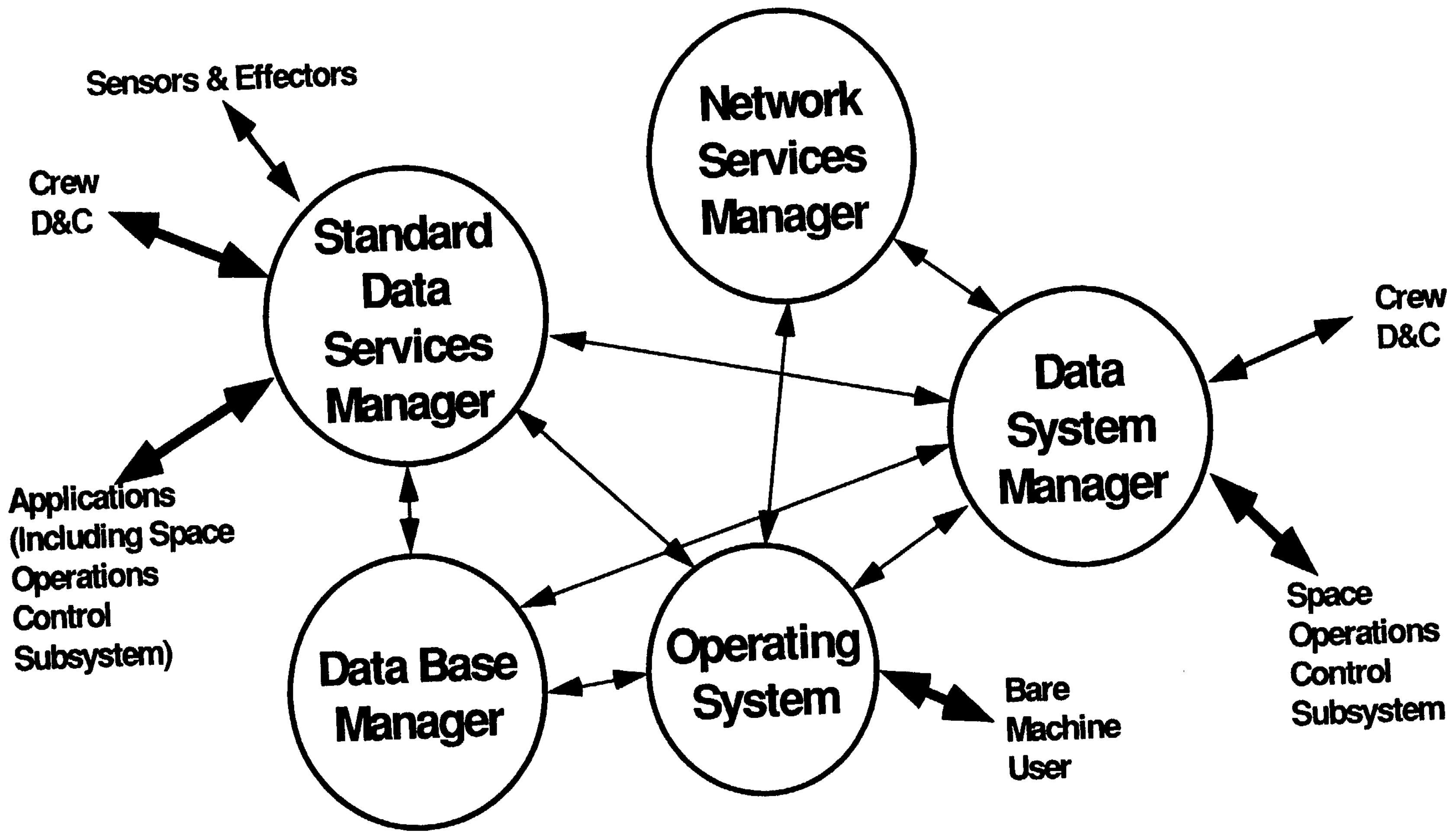


Figure 1-2.- Space Data System Services Architecture.

2. SYSTEMS ENGINEERING METHODOLOGY IMPLEMENTATION

The systems engineering methodology is a set of methods applicable to systems development over the entire life cycle of the system. The system life cycle is identified below. The flow of description from Phase A to D is not intended to imply that one phase is or must be completed before the next starts. The process should be an iterative process using simulations, prototypes, preliminary designs and test cases as needed to provide an ever better solution for a system.

- Phase A: Conceptual definition of a system within a larger picture of needs (i.e., a strategic view of the need for a system)
- Phase B: Requirements definition of a system which must produce complete and consistent requirements.
- Phase C: Design development for a system.
- Phase D: Operations and support for a system.

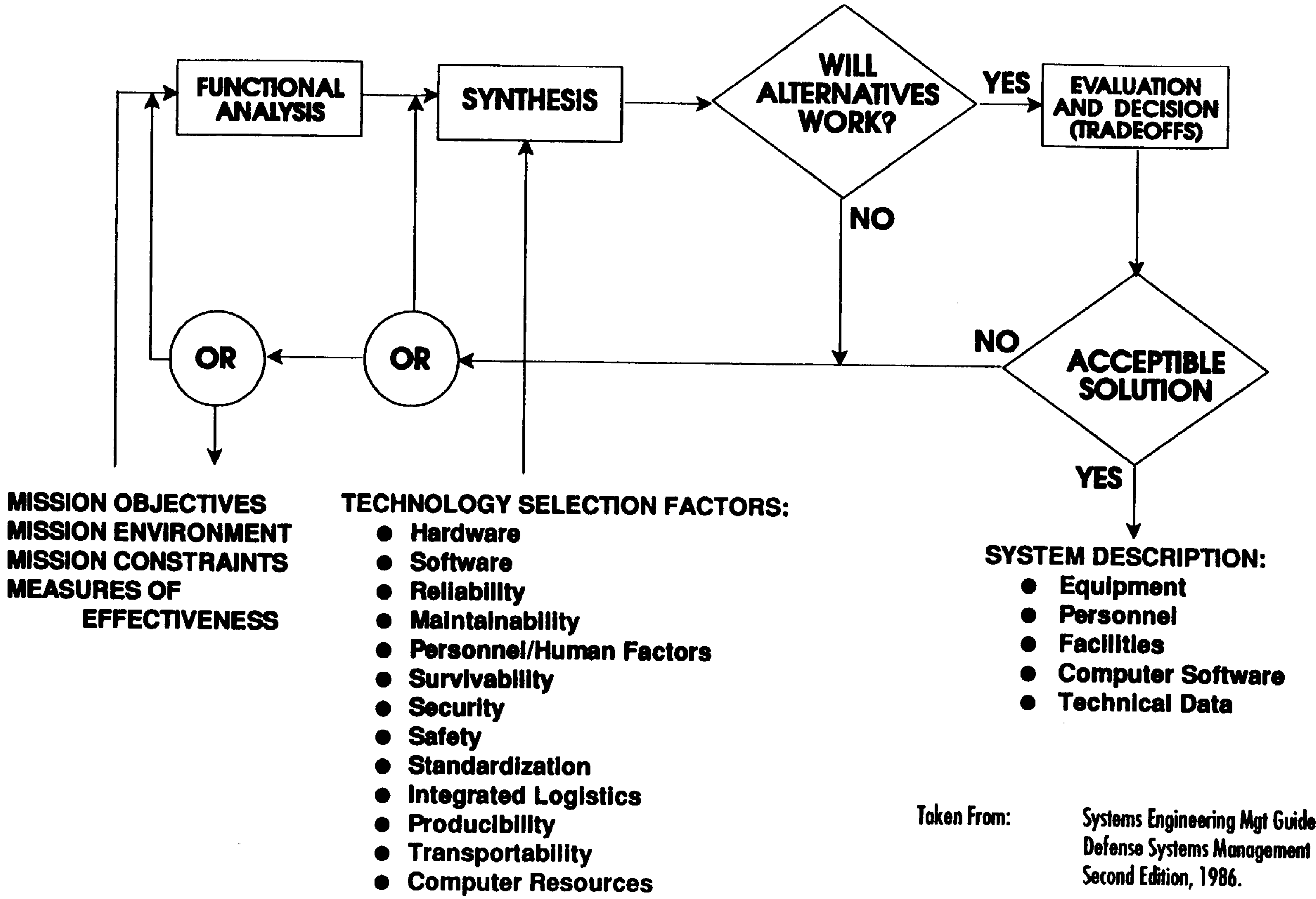
The focus of this methodology notebook is on requirements analysis for Phase A and Phase B, although it also can be applied to definition of detailed design requirements in Phase C or for the definition of operational and support facilities in Phase D. This section summarizes the systems engineering process and its analytical methods, with the emphasis on the methods used in requirements analysis. The other phases are summarized to establish an improved contextual understanding of how all analyses tie together to perform systems engineering.

The analysis process needed to perform a successful requirements analysis task does not consist of just one analysis. There are many types of analysis which must be performed with the results being integrated into an set of requirements which describe the technical needs of the users. An analysis of the system environment needed for the user to enable the requested system to operate effectively must be developed. A static analysis of the organization of system requirements provides an effective structure for holding the requirements for designers, but will not be sufficient for them to be able to assemble the technology into an effectively operable system. An analysis of quantitative performance needed from the system and its elements must provide effective goals for designers to target which remain stable during design because they are based on the needs of the mission, not the capabilities of technology. In an electronic system, especially in a real time system such as an avionics system, analysis and understanding of time and timing requirements are critical to insure that not only is the system designed to operate as fast as needed, but that it can be guaranteed to operate as needed.

The methodology must support concept definition, requirements definition at the systems and subsystems level, definition of the requirements for a design implementation, requirements for developing prototypes and simulations, and requirements for analyzing the performance of prototypes and simulations of the actual intended system or subsystems.

2.1 SYSTEMS ENGINEERING PROCESS SUMMARY

The system engineering process requires the performance of a process to develop a complete set of requirements for an entity that is realizable, within available technology, consistent with cost and budget, schedule and other constraints. Since the system engineering process is divided into requirements analysis and systems design, it is usually necessary to cycle back and forth between them to generate ideas, develop robustness and test detailed requirements and designs. The process is iterative and is summarized in figure 2-1.



Taken From: Systems Engineering Mgt Guide
Defense Systems Management College,
Second Edition, 1986.

Figure 2-1.- Conventional Systems Engineering Process.

The activities typical of the Systems Engineering Process are: functional analysis of inputs, synthesis of requirements, evaluation and decisions, and description of the system elements, usually in the form of system or subsystem/segment specifications. The typical functional analysis relies upon Functional Flow Diagrams (F2D) to show logical sequences and relationships of operational and support functions at the system level. F2Ds will show progressively more detail as the project being analyzed moves from the Concept Development phase to Requirements Definition to Design. More detail will be added in the way of operating and support functions in the Operations and Support phase of development. Synthesis is usually described as a process of developing design approaches or alternative approaches to meeting the requirements. Block Diagrams are used to develop and portray conceptual schematic arrangements of system elements meeting system or subsystem requirements. Evaluation and decision is the tradeoff part of the process during which stated operating requirements and engineering designs are evaluated continually while correlating characteristics of alternative solutions. Constraints establishing the selection criteria are assessed. Risk assessments are developed. This process continually iterates until an acceptable solution is found.

Table 2-1 shows the key activities implementing the System Engineering Process. This section describes the elements of the methodology as they fit into the system engineering process. It addresses definition of requirements, requirements prototyping and simulation, requirements performance analysis, design requirements definition and the use of the open software environment standard in the process.

TABLE 2-1.- STEPS IN THE SYSTEM ENGINEERING PROCESS

1. Determine user needs, measures of effectiveness analysis, and requirements identification.
2. Identify system options.
3. Perform functional analysis of user needs and requirements.
4. Identify the system elements within each system option.
5. Allocate functional requirements to the elements within the system options.
6. Assess each system options' satisfaction of quantitative user performance requirements as well as a sensitivity analysis to identify "point solutions".
7. Conduct trade studies within each system option to determine the optimum allocation of performance requirements to the system elements.
8. Conduct trade studies across the system options to determine the system option that best satisfies the performance requirements, sensitivity to constraints, producibility, logistics support, manpower considerations, etc. and compare risk assessment of each system option.
9. Identify the preferred system option and document with Type A specifications and operational description papers.
10. Finalize the allocation of performance requirements to the elements within the preferred system option (Type B specifications.)
11. Define interfaces between all system internal elements and between system and outside world (i.e., the surrounding operational and support environments.).
12. Enable start of detailed design of system elements and interfaces.
13. Begin technical performance monitoring and assess system impacts of any parameters of system elements that are projected to fall outside specified ranges.
14. Monitor test and evaluation of the system elements against their specifications.
15. Integrate the complete system and assure all interfaces are working properly.
16. Test and evaluate system performance and sensitivity to constraints against the established user needs (Type A specification).
17. Finalize the definition and documentation of the preferred system option (Type C specification).
18. Assure that all support equipment, documentation and manuals are completed and in compliance with requirements.
19. Provide support and monitor results for system transition to user.
20. Subject all change activity to an appropriately scaled down process and review to assure compatibility with basic user requirements.

(Reference Electronic Industries Association Bulletin SYSB-1)

2.2 REQUIREMENTS DEFINITION

The purpose of this methodology, as previously stated, is to provide an environment for analyzing and specifying requirements. The parts of this analytical technique are to use a basic methodology and to tailor its use to fit within the constraints of selected computer aided systems engineering tools. This results in an approach which accomplishes the purpose of this document, by resulting in complete and consistent requirements specifications. This section describes the assumptions and methods for accomplishing these steps. It also describes some key features of requirements associated with defining interfaces, standards.

2.2.1 METHODOLOGY ASSUMPTIONS

The methodology was developed by developing an architecture and recording all the thinking, false starts, good ideas and "hallway" discussions that took place. These notes then were compiled into this notebook. As part of the development of the methodology, assumptions had to be made to enable continued expansion of the methology. These assumptions are listed below so any constraints they may have place on the methodology will be visible.

- To develop this methodology, an avionics analysis was conducted. Although avionics does include the sensors and effector hardware for controlling the space vehicle, in this analysis a simplification will be used which assumes that the avionics we are examining excludes the sensor and effector hardware, in order to present a somewhat smaller problem for development and illustration purposes. Since the sensor and effector hardware development actually must take place in close cooperation with the controlling processing applications subsystem, from the viewpoint of overall high level control over avionics, this assumption merely isolates these hardware items from the central controller, which is desirable as an implementation of the principle of information hiding.

- Requirements common to multiple subsystems, processes or components should be stated only once.

- Requirements for generic entities are based in part on analysis of the requirements for functions, processes and data of an actual program. These requirements are then "converted" into generic requirements which are more appropriate to the architecture being defined. If the original program requirements are clear and do not conflict with the requirements of the rest of the architecture, then they are adopted as much as possible without change. Entity requirements are only changed if and when their actual non-generic nature becomes clear.

- A decentralized functional requirements model can be implemented in either a distributed, centralized or hybrid processing approach. A centralized functional requirements model can only be implemented in a centralized processing approach. The centralized model is a special case of a decentralized model (with the number of distribution nodes equal to 1). Thus, a decentralized model is used for generality.

2.2.2 BASIC METHODOLOGY

The basic methodology recommended is to perform an analysis of the static elements of the target system. This analysis should not be concerned with the tenets of any specific method such as Yourdan Data Flow Analysis, Ward-Mellor Structured Analysis or Object Oriented Analysis. The methodology presented below is a hybrid approach, drawing on elements of each of many formal methods. This methodology has been developed by documenting the practices of an actual analysis and expanding it where necessary to ensure effective definition of all the needed requirements for a space generic avionics system. This basic methodology is presented assuming an analysis of requirements is being conducted from initial concept definition with no available space generic avionics architecture. Development of requirements for an avionics system which can adopt an existing or this generic architecture would be more effective if based on tailoring the existing requirements of the generic architecture.

The basic approach is to perform an external environment and interface analysis, then a requirements static analysis, followed by determination of the operating concept for a simulation of the system, then a dynamic analysis of requirements performance, and finally by checking fidelity against the real environment, as shown in figure 2-2. This emphasizes utilization of a top-down, a bottom-up and another top-down analysis technique with iteration between the results of both techniques.

The first step in this methodology should be to define the environment within which the target system must operate, both externally in space and internally with other systems.

From this, establish the external interfaces which connect to entities outside the system being developed. This will facilitate definition of the boundaries of the target system which in turn will improve definition of the scope and content of the target system. This establishes the external purposes (requirements) which the system must meet. It also provides the starting point for the following steps.

The second step is to perform a static analysis of the requirements. This involves gathering together all functions, processes, services, inputs, outputs, data attributes and quantity performance data identified by users as needed, related to or otherwise associated with the target system without attempting to strongly or firmly categorize it. A search through documentation on similar programs, or programs in other operating mediums is needed to identify related capabilities which might suggest areas of requirements needed on this program or developmental system. For instance, in defining space operations control requirements investigate the operations control requirements associated with U.S. Air Force Control and Reporting Centers (CRCs) for possible requirements relevant to space. At this stage, do not attempt to filter or limit the areas of search. if any area is suggested for consideration, accept it without qualification and do not address whether it is relevant. This activity is similar to brainstroming in trying to come up with approaches, requirements areas and other objects which might trigger a thought in one of the requirements analysts.

Analyze this material to identify the categories into which the material falls. Establish higher level categories of understanding for the requirements using the lists from the last stage of the data gathering as checklists to see if all requirements categories have been established. Partition between such categories based on explicit interface criteria (e.g., all processes must operate as independently as possible or no single points of failure are permitted). Continue to boil these categories up into higher level entities until only one entity is left which represents the target system. This system entity then can be analyzed with respect to the results previously found in analyzing the target system environment and external interfaces.

Then it is necessary to re-analyze the current multi-level structure to determine if a better structural scheme can be established. If so, the elements of the structure need to be reorganized. If no better scheme can be identified at this time, this re-analysis may suggest that some of the elements are not located in the most effective place or that some of the internal interface could be better defined.

Having performed (so far) a middle-to-top analysis, a top-down analysis should be performed to explode entities, processes, data attributes or entities at each stage of the analysis into better defined entities, processes, data attributes or entities. This downward explosion analysis should focus, at each level, on how the system at that level would operate and develop associated diagrams describing system, subsystem or sub-...-subsystem level operation. This explosion process develops ever more detailed sub-processes. It should determine at each explosion level if the breakdown is complete for the entities being exploded, regardless of where the original process requirements derived from. This explosion structure forms the basis for cross checking the analyst's logic, and for later dynamic analysis.

The third step is then to develop an operating concept for the simulation of the requirements, which must be based on the operating concept for the actual system. Users should be closely involved to insure that the operating concept is both effective and supports investigation of alternative operating concepts if they have not been rigorously defined.

Then a dynamic analysis (i.e., simulation) of the requirements must be performed to determine the object values of performance requirements, and to test out potential requirements interactions. Finally, the dynamic and static requirements must be verified to represent the real world environment with sufficient fidelity to convey effective system development requirements to the system designers.

This basic methodology is referred to as a Hybrid, Object Oriented, Structured Analysis (HOOSA), as shown in figure 2-2.

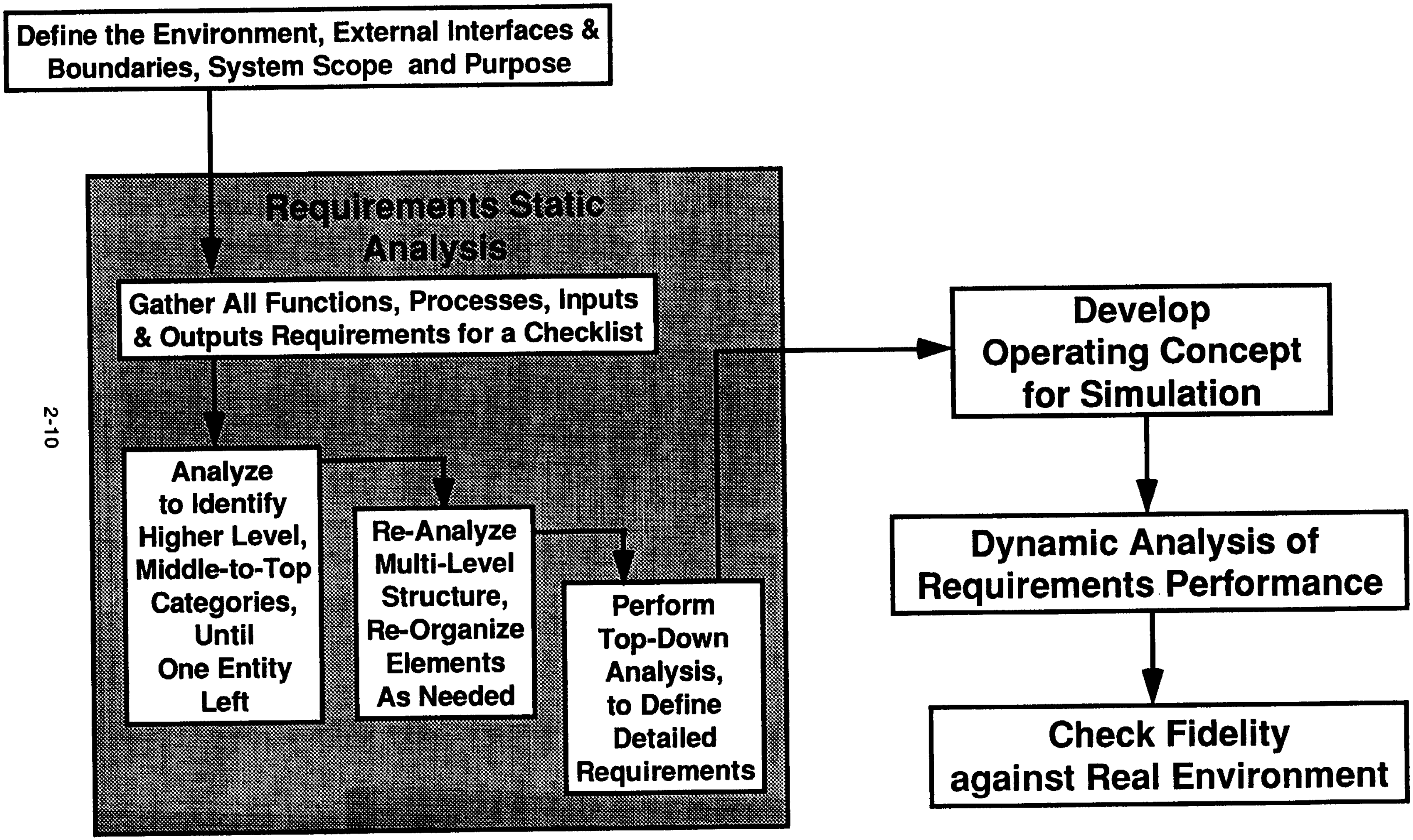

Define the Environment, External Interfaces & Boundaries, System Scope and Purpose
Requirements Static Analysis
Gather All Functions, Processes, Inputs & Outputs Requirements for a Checklist
Analyze to Identify Higher Level, Middle-to-Top Categories, Until One Entity Left
Re-Analyze Multi-Level Structure, Re-Organize Elements As Needed
Perform Top-Down Analysis, to Define Detailed Requirements
Develop Operating Concept for Simulation
Dynamic Analysis of Requirements Performance
Check Fidelity against Real Environment

Figure 2-2.- The Hybrid, Object Oriented, Structured Analysis Methodology.

2.2.3 USING TOOLS FOR SYSTEMS ENGINEERING

The performance of the Hybrid, Object Oriented, Structured Analysis requires use of advanced automated tools because the extensive interfaces across one level, the multiple levels of breakdowns, and the need for consistency across multiple levels of processing and interface definition can only be effectively maintained by automated tools which operate in a graphics mode and offer automated level balancing and consistency checking. Only automated tools can maintain the needed links between graphics and a data base, and offer an integrated data repository for all elements of the analysis. Such tools, referred to as Computer Aided Systems Engineering (CASE) tools, offer a very powerful capability to the systems engineer. This section identifies the key practices and issues for CASE tool-based analysis, describes the role of static hybrid analysis approach and the approach that should be used for documenting the analysis and resulting specifications.

The use of CASE tools offers an entire range of options not previously available (using manual techniques) to an analyst. The analysis must address static techniques, dynamic techniques and documentation to show the results. Static techniques capture the basic information on the problem or system environment using multiple approaches. Dynamic techniques extend the static model by capturing timing and movement information to test the models effectiveness in operation. Documentation is always needed to be able to present the results or to provide specifications to designers for construction of the resulting system. Figure 2-3 presents an overview of the scope of capabilities included in CASE tools used for this methodology development. Examples of each type of capability are shown and described below.

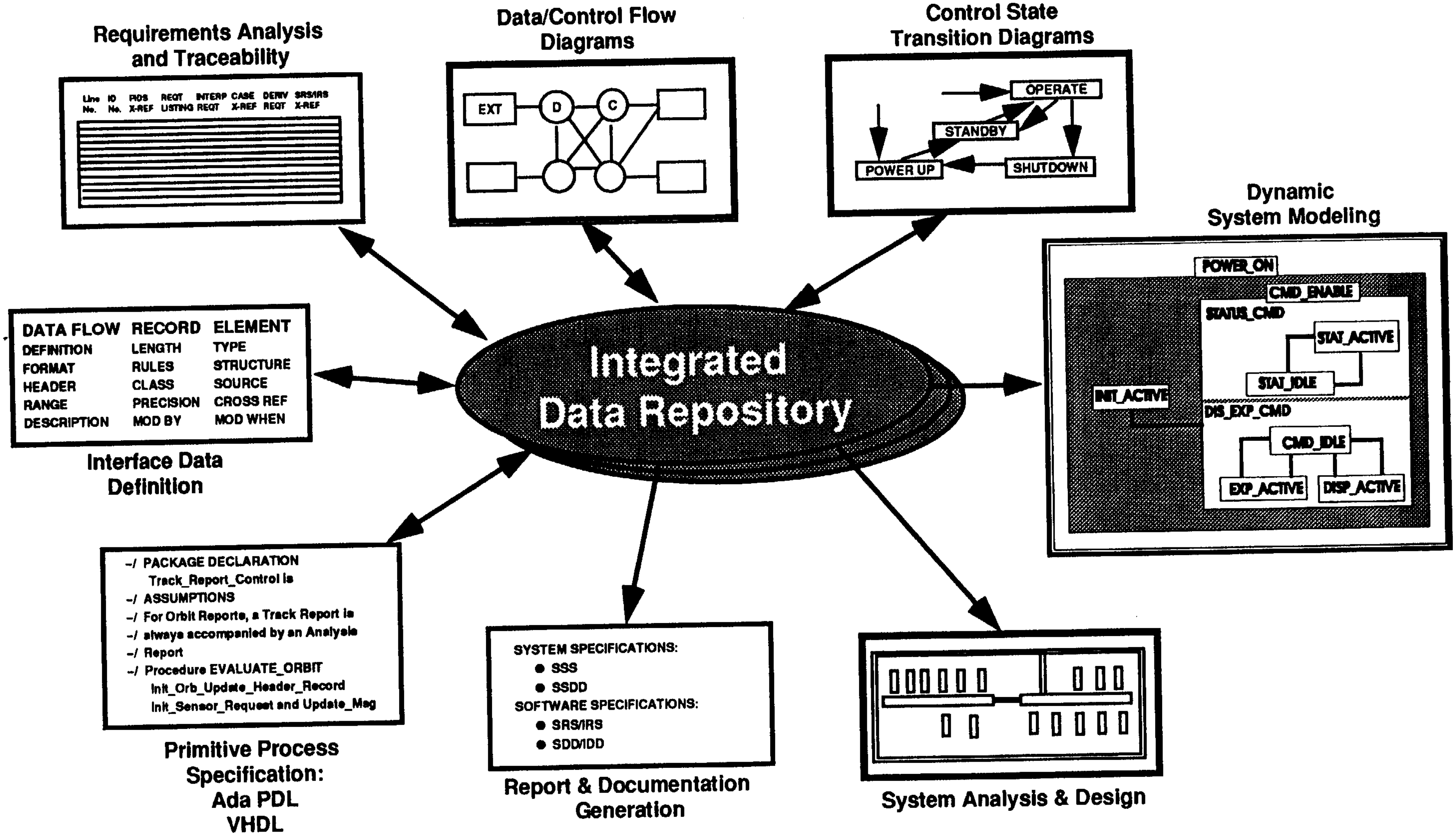


Figure 2-3.- Computer Aided Systems Engineering (CASE) Features.

The key to the value in CASE tools lies in large measure in their integrated data repositories which capture all data in the system in one integrated location, with access from anywhere in the analysis process. The use of CASE tools enables the building of data (D) and control (C) flow diagrams which describe the processing in the system. Control state diagrams can be used to describe the flow of control in the system to capture the method of operation. Tables of requirements can be captured in the data repository and directly manipulated in the data repository or from the graphic representation of the system. Interface definitions can be more easily created by attaching interface requirements to the graphical representation of the interfaces and using the data repository to cross check them. The building of graphically oriented diagrams can continue by structured techniques to lower and lower levels, until the "bottom" has been reached with the definition of "primitive" processes and data. At the bottom of a structured analysis, the tool enables the definition of primitive process specifications, which can use program design language (PDL) for software entities or Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) for hardware entities. From the data repository, results can be captured for use in dynamic modeling of the system, for performing system analyses and design, and for producing reports, specifications and other documentation. These techniques are described in more detail below.

Tools treat inputs to a process and outputs from a process independently, linked only by name similarity. Thus the user of data and the generator of data are independent unless the analyst wants to force dependency by using the same name for the data being flowed. See Appendix E for naming conventions.

2.2.3.1 Multi-tool Use

Perhaps some day there will be a perfect tool which enables requirements analysis and design to be performed in a single environment, but that day is far off. Current environments and tools for requirements development are far from being sufficient or acceptable ***as integrated environments and tools.*** A developer should not mandate or rely on use of a single tool or single methodology. The current state of the art in methodology and in tool implementations of methodology are still evolving rapidly. It is recommended that tools be chosen which are mature and robust over tools which are the latest on the market, but have not proven themselves in use to be reliable and relatively bug-free. Data reliability in architecting and designing a system is an absolute essential. It would be better to pick any one of many tools in several categories which are ***acceptable*** and then to press on with the target system requirements development. While the target system developers are working in their areas of product system expertise, tool developers could be looking to insure that environment developers are not going astray by focusing on bigger and better tools while basic tool needs are forgotten. For instance, in the current state of the tool art, the biggest needs are for open integrated data repositories, and effective interfaces between different tools to enable analysis and design development to move from one tool to another without loss of analysis or design data.

2.2.3.2 Types and Quantities of Tools Needed

The tools which are currently needed include:

- Computer Aided Systems Engineering (CASE) static analysis tools which support an integrated data repository which includes interfaces that are not treated as so proprietary that the vendor will not enable the developer to adapt the repository structure to the developer's individual needs. Tools such as Excelerator by Intersolv and Teamwork by CADRE Technologies allow the developer to structure the repository and to write their own software to translate data for other tools.

- Quantitative performance analysis tools are needed to analyze quantitative performance requirements such as input/output rates, throughputs, instructions per second and operations per second. Tradeoffs between numeric requirements in different parts of the requirements model must be supported. Only spreadsheet tools have been identified to date which partially meet this need. No dynamic analysis, partitioning and allocation tools appear to exist yet.

- Timing analysis tools are needed to allow a user to analyze the mission and mission dependent timing factors, to allocate time requirements down from mission dependencies to system elements, to partition and allocate time requirements into ever finer (smaller) quantitative time requirements for lower level subsystem elements, to tradeoff time requirements between different subsystem elements, and to investigate the effects of system timing changes on the mission. Currently, only spreadsheets offer partial capability in this area.

- Dynamic CASE tools are needed to determine that the interfaces between parts of the model have no unexpected requirements conflicts when actually executed, that no timing difficulties will appear in the sequencing of required activities, and that process execution can be tested before commitment to design.

Automated tools must check consistency across all diagrams and interfaces at each level, and between parent and childs within any one explosion path. The use of multiple tools or multiple instantiations of one tool (if one tool is found which does in fact perform all analyses needed for requirements analysis and design) must be applied judiciously, without apriori blinders on tools needed.

The minimum number of tools required include:

- One static analysis tool is needed for each individual engineer/analyst's work area to develop ideas and flesh them out. This systems engineering tool needs to be used to prove out ideas for completeness and for consistent logic, based on a complete system model (if it exists). Ideas must be able to be tried out with minimal risk to the overall system by isolating this model in an individual's workstation where no easy interaction with a networked system risks propagating premature changes prior to management approval.

- At least one tool is needed for static development of a group revision of the team approach. This is the main tool for formal specification (if needed) of a complete and consistent model of the system as the developer plans to specify it to the builder (contractor). Configuration control capabilities in this tool are critical to preclude likely loss of control over the design in a team approach.

- At least one set of tools is needed for quantitative performance analysis of numeric performance requirements and alternatives to be tested for tradeoffs. Spreadsheets must suffice as a minimum until tool technology catches up to this need.

- At least one tool is needed for dynamic development of executable models of interacting processes using simulated data. Dynamic models prove out the completeness, timing, data flow path usage, process linking logic, consistency of data, and coherency of the concept in simulated use. Executable models provide a more rigorous form to developing systems. This tool is separate from the quantitative performance analysis tools discussed above.

2.2.3.3 Tool Output and the Data Repository

The output of CASE tool analysis should be documentation built automatically around the data repository serving the tool. Reviews should address the tools' view of the system.

The documentation should be produced by the tool in its basic format and should be acceptable for review as long as the data needed is contained in that hardcopy. Hardcopy requirements need to be defined to address the key data to be reported, not the structure of the data report. Specification requirements need to be considered to determine whether the hardcopy output in tool format, system dynamic models or the data repository can be accepted as the contractually binding specification to be provided to system builders.

The data repository should be capable of storing all the data generated by the system engineering process described in table 2-1 and the detailed tables that follow table 2-1. The data repository must be capable of modification to support the specific structure of data needed in the project being analyzed. The data repository should be capable of conversion of its data base to the data base structure of other data repositories to facilitate transfer of the data underlying an analysis from one tool to another tool.

2.2.4 CONVENTIONAL ANALYSIS APPROACHES

The method used in this document merges three conventional analysis techniques: structured analysis, object oriented, and interactive development to produce the recommended methodology: the Hybrid, Object Oriented, Structured Analysis (HOOSA) methodology. This recommended methodology is thus a hybrid of each of these three and has been tested on the large space system definition. Each of these standard approaches is described below, along with the features of each pulled into the recommended hybrid approach.

2.2.4.1 Structured Analysis Approaches

Structured Analysis techniques are based on modeling the data flows in a system. The focus is on defining the data and control flows throughout a system, the transformations on the data and control flows, the data stores used in moving data and control, primitive process specifications and data dictionaries of the data and controls. By addressing the movement and transformation of data and control, there is a close correspondence between the real world and the requirements model constructed to represent the users' needs. Numerous tools are available to implement these techniques, the most widespread of which are the Ward-Mellor, DeMarco and Gane-Sarson methodologies. However, a common problem in structured analysis techniques is knowing when to stop since data can always be more finely subdivided. Another difficulty is selection of bubbles, representing events in the real world which use or provide data or control, and placement of stores as intermediaries between bubbles. These techniques commonly focus on modeling the physical flow of data and control, which is once removed from the actual requirements as noted in Section 2.2.5.2 in the discussion of logical versus physical requirements. Figure 2-4 represents an example of the partitioning process often used in structured analysis.

The structured analysis techniques are exemplified by figure 2-4. The highest level context diagram shows the system as a monolithic bubble with rectangular external entities outside the system. These external entities in the external environment drive the need for interfaces from the system to outside systems, and establish the basic purpose for which the system is being created. The system can then be exploded or broken out into a system diagram which shows the major subsystems (here shown as A1, A2, A3 and A4). The system diagram's external interfaces are breakouts of the external interfaces from the higher level context diagram. The system diagram's internal interfaces are new descriptions of data attributes needed as input to a subsystem or produced as output for some other subsystem. Each subsystem can then be further broken out (indefinitely) into more detailed lower level subsystems or processes. Thus, for example, subsystem A3 is broken into processes A31, A32, A33, A34 and A35. The inputs to A3 are broken into the inputs to A31 and A34 in the diagram. The output from A3 has just one process output from A35.

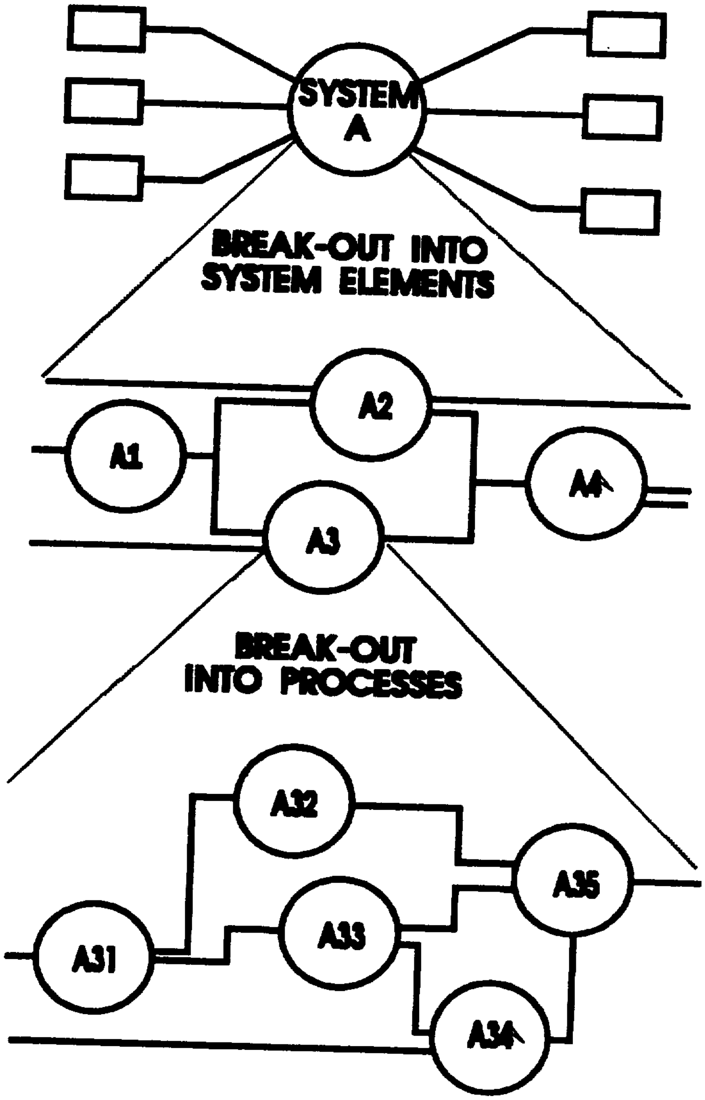


Figure 2-4.- Structured Analysis Example.

Closely related to structured analysis is another technique called functional decomposition. This technique also uses the structured breakdown approach shown in figure 2-4, but addresses functions, subfunctions and functional interfaces. The basic approach of functional decomposition is to select functions based on the processes which need to be performed for a system. This technique focuses an analyst's attention on the defining of functions and processes which need to be performed by the entities in the real environment. It can be difficult determining whether the requirements accurately reflect the needs of the real environment entities. Testing can be difficult to verify that the users' needs have been met. Functional decompositions are difficult because the functions are highly volatile (subject to requirements changes), and can be relatively subjective in how they are constructed because there is not a clear and direct linkage to the real environment.

Another related approach is a technique called information modeling. This technique models the entities in the real world and the information attributes, relationships and classification types associated with them. As such, it more closely follows the needs of the users and more faithfully represents their requirements. This technique is also similar to the object oriented approach discussed next.

2.2.4.2 Object Oriented Approach

A new methodology known as object oriented analysis (OOA) is currently evolving and shows promise. It offers the use of "objects" as abstractions of real world entities involved in solving problems; it can improve the definition and capture of requirements for solving problems within a real environment. Its definition is still changing, but includes features from information modeling approaches by representing real world objects and their information needs. It is also includes features from the programming language design and development world by use of classes of objects with inheritance among class members.

Object oriented modeling is not yet sufficiently mature and stable as a methodology to be relied on for analyzing major systems or sets of systems such as found in larger space systems, but does offer some features which can improve our methodology. These features are shown in the figure 2-5. OOA suggests the value of addressing object services and attributes as an intrinsic whole instead of functions or data individually and separately. The problem with analyzing functions or data per se is

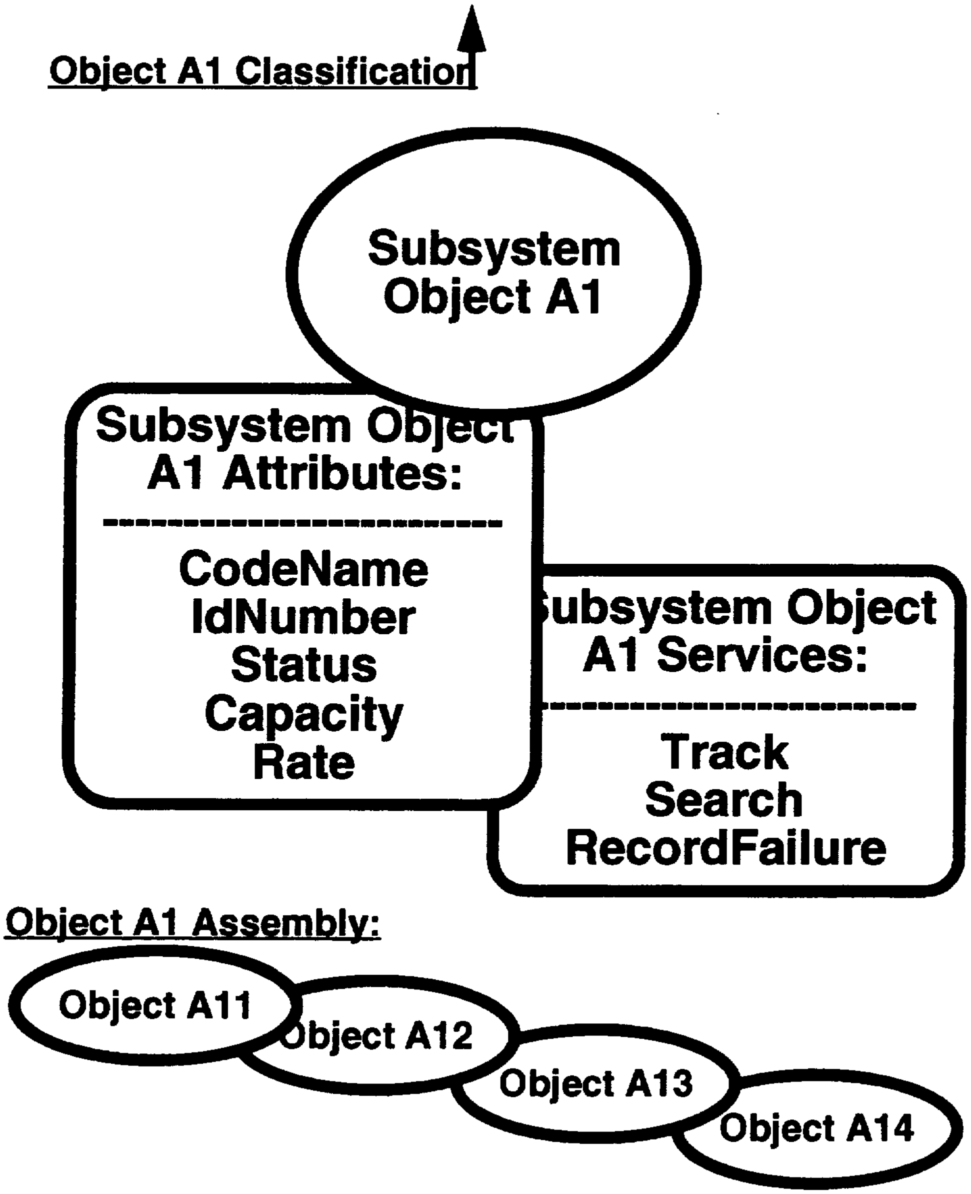


Figure 2-5.- Object Oriented Approach Features.

that they tend to change as the environment understanding changes or needs change. However the objects involved in the problem space do not change (as much). By abstracting the functions and process into services, and attaching the data with an object, a more stable definition can result. But the process is only as good as the analysis.

Since the OOA approach is not yet well defined, the Hybrid, Object Oriented, Structured Analysis approach described in section 2.2.5 was selected to avoid accusations that the recommended approach does not precisely follow some individual variation of the OOA approach; the interest here is to define requirements in a practice that works rather than follow some pro-forma methodology which may not be clearly understandable.

2.2.4.3 Interactive Development

Another technique needed in the hybrid approach was interactive development. This is an informal technique to use computer tools to develop a model of the system. First a static model is built to capture all the requirements, to demonstrate the requirements and their interpretations as understood by the analysts, and to obtain operational user validation that the requirements appear to be correct and acceptable at the point in time they are being demonstrated. No final validation can be expected in a complex system because the complexity interferes with the human ability to determine whether the requirements model actually captures the intended requirements of the operational user, as well as the technology capability which will implement the requirements. The second stage of interactive development is to implement a dynamic model of the system which determines whether the requirements interactive in unanticipated ways or are subtly in conflict.

Dynamic requirements simulations should not be built before the static model for new systems because the dynamic execution of a model can introduce its own difficulties which interfere with the analyst and users ability to "see" if the requirements have been effectively captured in the model. Separating the users statements of the requirements (i.e., the static model) from the operating dependent view of the requirements (i.e., the dynamic model) simplifies verification and validation of the requirements. (Note these models are addressing requirements not designs, as will be discussed below.)

2.2.5 HYBRID METHOD FEATURES

The merger of the structured analysis, object oriented approaches, and interactive development yields a more robust methodology with improved description of requirements for space systems development. Features facilitating object oriented improved requirements descriptions, supporting better definition of interfaces, stronger utility of standards, inheritance of requirements to lower level elements in a top-down structure, clearer partitioning between applications and services and allocation of performance requirements, and improved definition of concurrent engineering requirements must be part of an effective methodology. This section describes some of the key hybrid approach features needed and how they should be used in the methodology.

2.2.5.1 OOA Features Needed

The features of OOA used in our methodology include definition of objects by abstracting the processes and data, and establishing the services which operate on the data based on inputs to the object. In the Hybrid Object Oriented Structured Analysis, abstracted processes and data are referred to as entities to distinguish them from OOA's objects. Data attributes are defined for each object and similarly for each hybrid approach entity. Services are the processes performed as a result of messages received by the object. In the hybrid approach, services are more system process oriented.

OOA suggests the use of assemblies which are component parts of an object broken down into lower level objects; this is similar to the hybrid approach's structured breakdown of entities into entities at lower levels or sub-entities. Classification is a feature of OOA by which each object is identified as a member of a class for inheritance purposes; multiple class memberships are possible for each object. The hybrid approach allows only one class membership, namely that higher level entity which spawns the entity. Inheritance is partially available in the hybrid approach as noted below.

OOA also offers features such as information hiding (encapsulation) whereby the external view of an object is represented at each level as a monolith, or single entity, with only inputs and outputs needed by other entities at the same level. The internal workings of each entity are hidden and not accessible from outside that entity to protect it from external interference in its operation. Information hiding is achieved in the Hybrid Object Oriented Structured Analysis by defining external interfaces and services for each entity which are the only access points for the entity.

Inheritance is a feature of OOA by which common attributes and services of an object as specified once and then extended to each specific case of the object. In the Hybrid Object Oriented Structured Analysis, inheritance is achieved by defining that the requirements for an entity automatically apply to all lower level entities. If a requirement does not automatically extend to all lower level entities, then it would not show up at the higher level, but would be attached to the lower level entities to which it applied (if it applied also to all lower level entities in turn). This suggests an iterative process by which an entity is created and requirements attached to it. As lower level entities are later created, some requirements may not apply universally and thus would be moved down to lower level entities.

2.2.5.2 Interface Requirements Definition

The requirements defined for the system must include quantitative and performance requirements because these are the points which determine whether the as-built product will actually meet the users' needs; building the product so that it works is often easier than making it work fast enough or often enough, etc. Verifying that interfaces function acceptably can be difficult. Identifying and specifying hardware interfaces can be easier than defining the architectural and software interfaces since hardware can be touched and viewed. Architectural structures (above the hardware/software partitioning level) and software interfaces can be hard to define because they are hard to visualize. Figure 2-6 represents the difficulties involved in defining architectural and software interfaces.

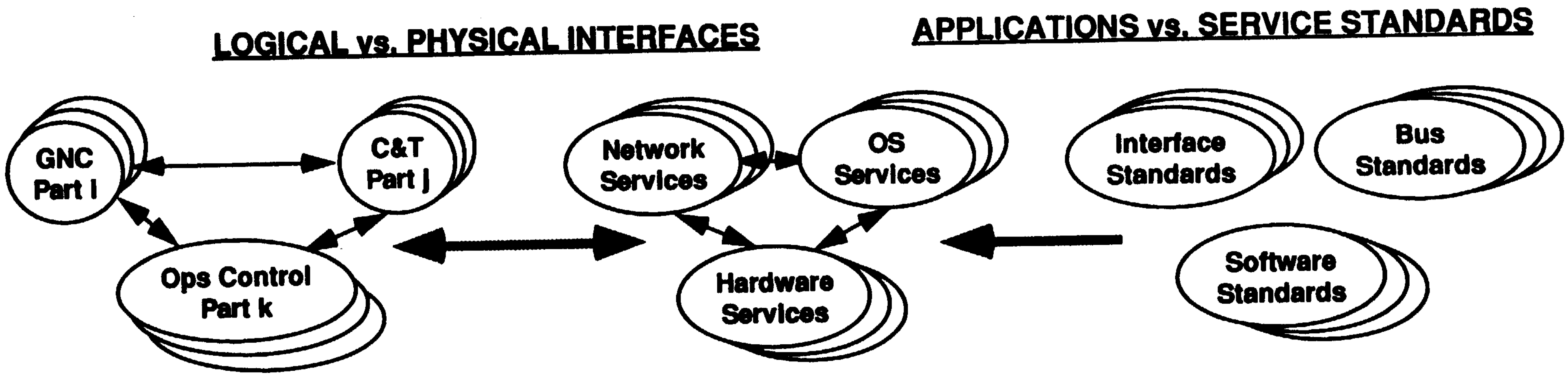


Figure 2-6.- Architecture and Software Interfaces.

Within an individual subsystem, as represented by the stacks in Figure 2-6, interface definition is a conventional process known to specialists in these areas of expertise. As systems get bigger and more integrated, however, more interfaces will exist between different subsystems; such interfaces may derive from use of common hardware elements, from sharing of resources including services, or from reuse of standardized software packages. Such interfaces pose different problems in requirements definition because they are a new problem. Definition of the interfaces between such different applications subsystems as Guidance, Navigation and Control (GNC) and Communications and Tracking (C&T), as shown by the stacks on the left in this figure, require a more explicit use of an effective methodology.

Intersystem applications interface requirements should address ***logical*** requirements of the interface. That is, the end user of the data should be identified with the reason the data is needed, and the source originating the data should be identified. Detailed performance characteristics for the interface to meet the end user application's requirements should be identified. Routing of the data should not be a concern to the source and user because the routing (i.e., physical requirements) should be transparent to these entities.

Similarly, within the services area, services interfaces should also define the logical service interface requirements and not the routing unless the routing is relevant to the logical flow of data. ***Physical*** interface requirements are normally a design issue unless the physical implementation has implications for the logical use or need for data, only then should the physical implementation be specified as a requirement. For instance, a service such as a Reports Generator getting data from a Data Base Manager might not need to know the inter-network addressing of the Data Base Manager, but the Network Manager providing the data would need to know the routing requirements of the hardware services.

2.2.5.3 Standards Requirements

Figure 2-6 points to the interaction between applications and service standards. Standards must be applied to the definition of aspects of the developing system such as interfaces, buses, software development, etc. Typically, these standards will apply to the services, especially as they are planned to be physically implemented. Other standards may apply to the applications or may be developed specifically to guide the applications development. If so, these sets of standards must not be in conflict, or design problems will occur. For instance, if an application standard called for distribution of timing synchronization through software for maximum flexibility, and a service standard called for updating timing synchronization marks at 5 nanosecond intervals, this would present a standards conflict to the designers since software cannot achieve 5 nanosecond speeds while hardware can.

The standards to be applied must also be tailored to the purpose of each applications subsystem to which they apply by the use of standards' profiles which depict how the standards will be specifically applied to each targeted subsystem application. Standards must be tailored in profiles to specify how the required interfaces should operate. Application of the standards must identify specific sections and subsections of the standards and attach those tailorings to the data base as a requirement for each of the application or service entities to which the standard profile is being applied. This will be clarified in the next few paragraphs.

2.2.5.4 Requirements Inheritance

Hierarchical and cascaded requirements are needed in requirements listings to achieve inheritance as shown in figure 2-7. The bubbles on the left of the figure represent two levels of entities on a structured breakdown chart, where entities A31 to A35 are breakouts of entity A3. Requirements would be stated on each entity. To make this work in this analysis, the following guidelines were implemented:

- Requirements on any entity i on a level j must apply to each and equally to every entity i1 to ik on level j+1 for every i = 1 to n and j = 1 to m. Thus in figure 2-7, requirements stated on entity A3 are defined to apply to every entity A31 to A35.

- Requirements not applicable to all of A31 to A35 must be applied to each of them individually. For instance, if only A31 and A32 have service requirements to generate reports, and they generate the same report for different data cases, then the requirement to call the Reports Generator service and access the same report table and generate the same type of report data from the same data base fields (but containing different data field entries), would be repeated in both A31 and A32.

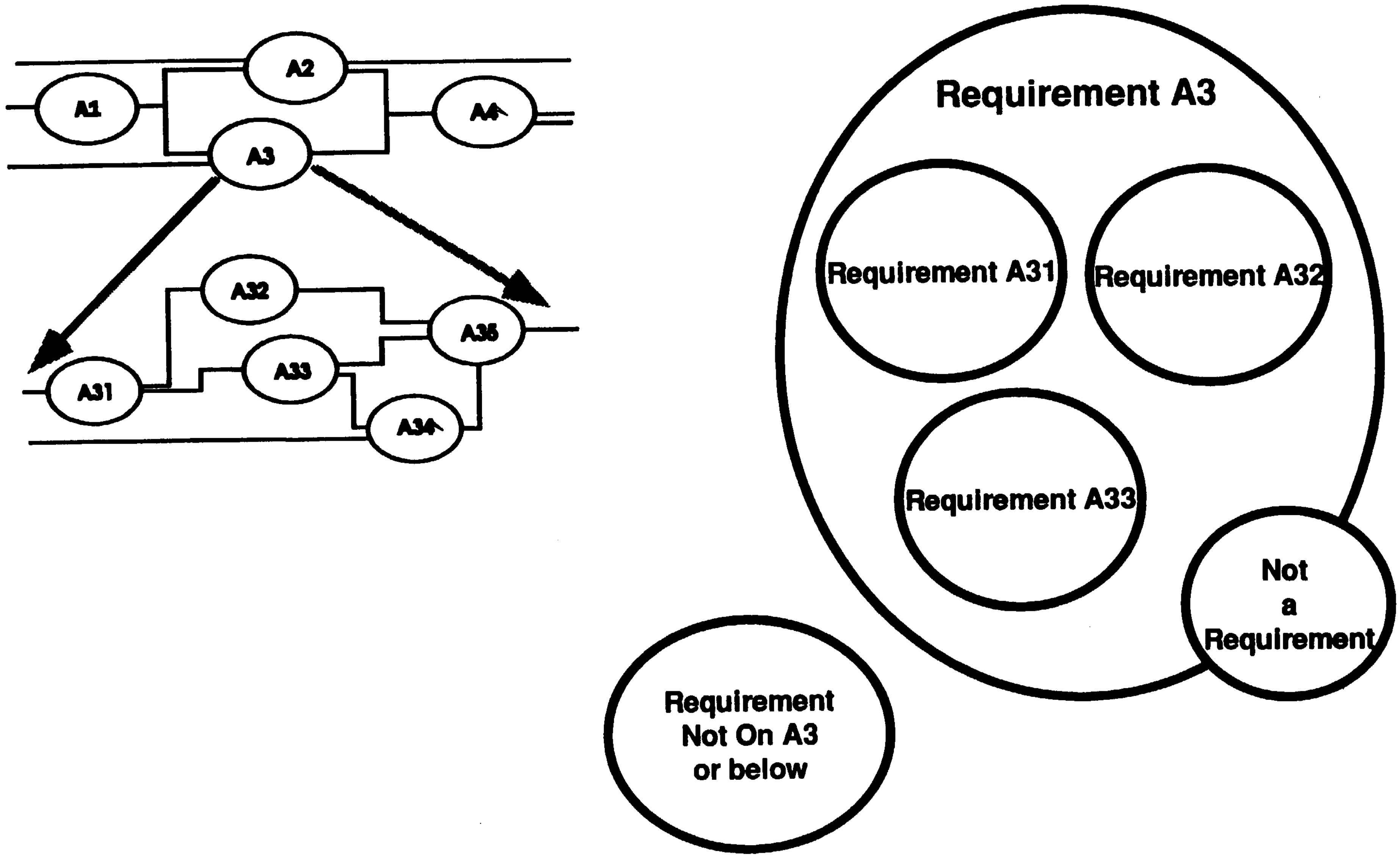


Figure 2-7.- Requirements Inheritance Approach.

- Where requirements A31 to A35 are additive and sum into A3, then on A3 they are applied as a Note rather than as a Requirement. The requirement is the lower level statement which is testable. For instance, if timing requirements were applied to A31 of 100 seconds, to A32 of 200 seconds, to A33 of 300 seconds, to A34 of 400 seconds, and to A35 of 500 seconds, the sum (1500 seconds) would be applied to A3 as a Note for informational purposes but not as a requirement because the 1500 second figure is not testable; only the individual figures of 100, 200, etc seconds are testable.

- Requirements cannot cross the boundary as shown by the "not a requirement" bubble in the figure. Requirements which apply to more than one but less than all entities A31 to A35 must be stated and repeated for each entity to which the requirements do apply.

- Requirements cannot overlap. The same requirement can be stated more than once for different entities as noted above, but the different entities cannot use pointers or references to requirements stated elsewhere. For instance, A31 and A32 must be disjoint requirements statements (no overlap); although the basic requirements statement may be the same, each may be more tailored to be specific to the entity containing it. A requirement in A31 may be implicitly derived from A3 by inheritance with additional explicitly stated modifications, or may be explicitly stated new requirements. Requirements listings should include explicit statements, as Notes, of the analysts understanding and interpretation of what the requirement means for future reference.

2.2.5.5 Performance Requirements

Between the applications subsystems such as GNC and the service subsystems such as Operating Systems (OS), interfaces are needed to identify the applications interactions with services, the performance requirements to be levied on the services by the applications need (if any), and any unique operating requirements on the service. This set of requirements can be linked together as illustrated in figure 2-8. While the applications subsystem requirements are defined in the structure on the left in this figure, the detailed requirements are included in the data repository (i.e., the rounded rectangles under each bubble) accessible from each entity (i.e., bubble) and underlying the definition of each entity. For instance, state/mode control for vehicle

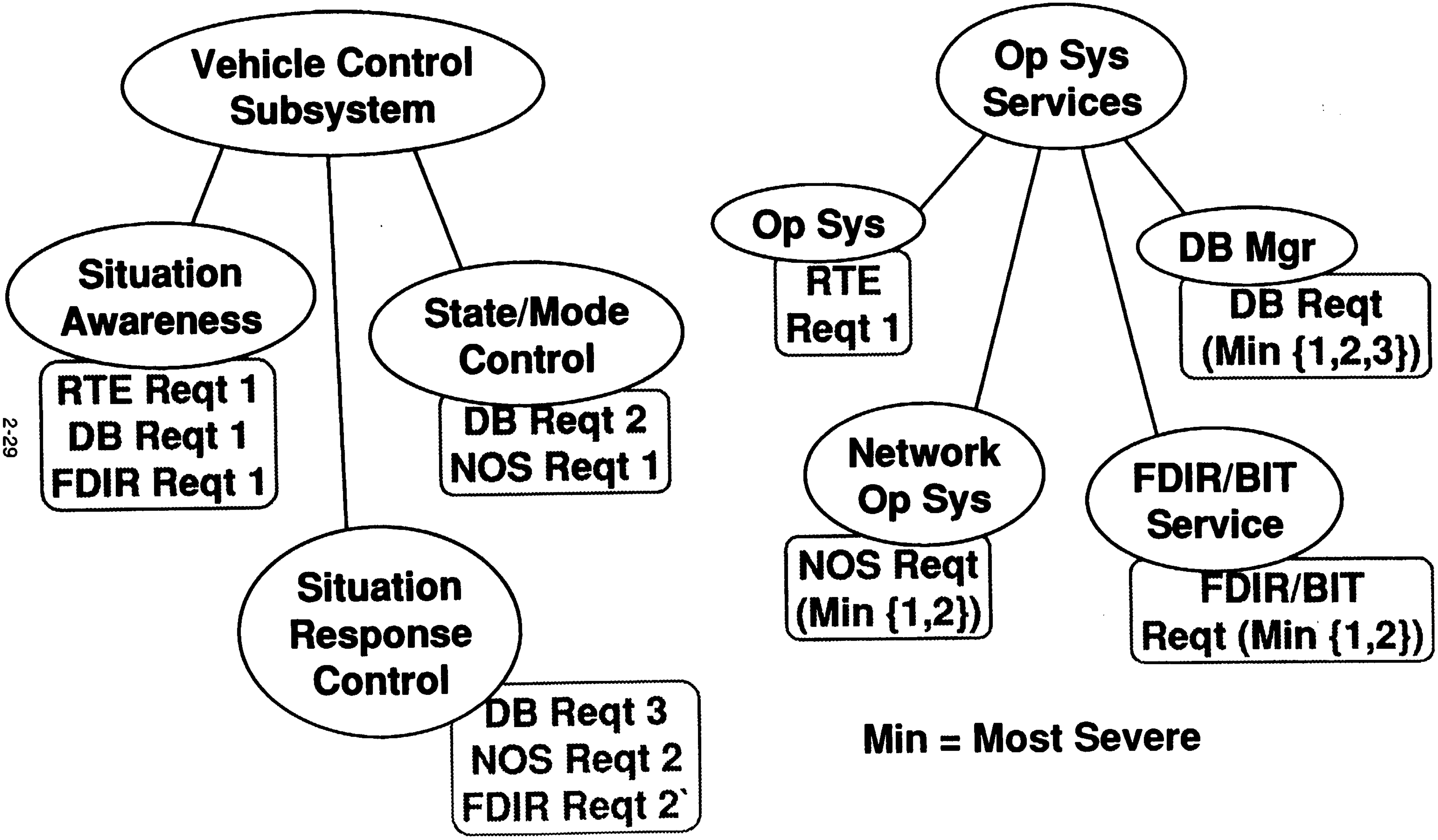


Figure 2-8.- Example of Application Requirements for Services.

control might establish requirements on the Data Base Manager (DB Reqt 2) and on the Network Manager (NOS Reqt 1). Each of these is a call on a standard service provided by the Operating System Services. The analyst would compare all calls for the DB Manager to determine the nominal as well as the most stringent individual requirements in order to specify the overall requirements for the DB Manager.

Just as the performance requirements are placed in the data repository so they can be attached to the services which must meet them, so also can the standards profiles be placed in the data repository. The placement of standards profiles (i.e., tailorings of the standards to the entity) in the data repository indicates that the standard as tailored in the profile is a requirement just as important as the performance requirements in the data repository. The data repository must then maintain traceability from the structural as well as the performance and standards requirements back to the generating source of the requirement to support trading off requirements and other factors such as costs. Cost considerations can only be realistically handled if the generating source and all links to the resulting requirements are explicitly known.

2.2.5.6 Concurrent Engineering Requirements

Concurrent engineering is concerned with the development and specification of requirements. The requirements for the major functions (such as GNC control, communications control, etc.) must be coordinated with the requirements for supporting and operating these functions. The support and operations requirements are usually labeled the Quality or "Ility" requirements. Table 2-2 summarizes the types of data that may be needed for identifying and specifying requirements. Concurrent engineering requirements are defined here as all the inter-related requirements.

Table 2-2. Requirements Summary for Documentation

o System Entities
o Architecture
o Environment
o Qualities
o Timelines and Timing Requirements
o Performance Requirements
o Attributes (Data)

The development of requirements for system entities, the architecture, environment and data attributes will be discussed extensively in this methodology. In summary, the system entities requirements must result in the definition of the specifics shown in table 2-3.

Table 2-3.- System Entity Requirements Summary

- o Purpose
- o Functions/Processes/Services Contained
- o Special Class Membership (if applicable)

The architecture requirements must result in the definition of the elements shown in table 2-4.

Table 2-4. System Architecture Requirements Summary

- o General
- o Platform Unique Extensions
  - - Orbiter (Earth, Moon, Mars)
  - - Transfer Vehicle (Earth Orbit-to-Lunar Orbit, Earth Orbit-to-Mars Orbit, etc.)
  - - Excursion Vehicle (Lunar Orbit-to-Lunar Surface, Mars Orbit-to-Mars Surface)
  - - Rover Vehicle (Lunar Site-to-Near Region, Lunar Site-to-Far Region, Mars Base-to-Construction Site, etc.)
- o Mission Unique Requirements
  - - Surface-to-Orbit (SO)
  - - Docking/Berthing (DB)
  - - Orbit Station Keeping (OSK - Low Planetary Orbit up Geosynchronous Orbit)
  - - Orbit Transfer-to-New Orbit (OO)
  - - Orbit-to-Surface (OS)
  - - Base Internal Operations (Base)
  - - Base Excursion Operations (BE)

The environmental requirements must result in definition of the elements shown in table 2-5.

Table 2-5. System Environment Requirements Summary

- External
- Internal

The system data attribute requirements must result in the definition of the elements shown in table 2-6.

Table 2-6. System Data Requirements Summary

- Purpose
- Type (Block Data, Message, etc.)
- Trigger/Event Driven
- Synchronous/Asynchronous
- Continuous/Intermittent
- Criticality
- Filtering
- Storage
- Flows, Records and Fields
- Hierarchy and Structure
- Generic or Specific
- Platform Unique Extensions
- Mission Unique Needs
- Foreign Source
- Data Security

Requirements must be documented in specifications. The specification documents must cover the types of data addressed in Table 2-2 to specify systems and subsystems; this data should be established during requirements analysis and should be available in the data repository for the system.

In the remainder of this section (2.2.5.6), the concern is with the quality, time, performance and cost requirements.

2.2.5.6.1 Quality Requirements Definition.

A part of definition of the performance and standards requirements is specifying the requirements for the Qualities or "Ilities", that is, for Connectivity, Flexibility, Reliability, Maintainability, Recoverability, Simplicity, Commonality, Expandibility and Maturity as shown in Table 2-7. Related are tailoring and unique extension requirements needed to adapt the system being specified to the platform in which it operates. The quality requirements can affect every aspect of the developing system, and should in fact be designed into the system from the start rather than afterwards as an "add-on", which has been shown to not be effective in many programs. For instance, requirements for reliability must not only specify the actual requirements numbers, but also must consider architectural issues in reliability such as, are additional qualities needed in the requirements data base such as triple voting concurrent processing for life critical processes, or are special entities representing unique functionality or capability to achieve higher reliability needed such as a voting assessor. To be designed early, quality requirements must be specified as part of the requirements analysis.

Table 2-7. System Quality Requirements Summary

- Connectivity (Closely Coupled, Loosely Coupled, etc.)
- Flexibility
- Reliability
- Maintainability
- Recoverability
- Simplicity
- Commonality
  - Definition
  - Classes (if any) of Common Elements
  - Class Requirements
- Expandibility
  - Spare Capability
  - Growth Capability
  - Computation of Spare vs. Growth (Source Requirement x Spare x Growth, [Source Requirement x Spare] + [Source Requirement x Growth], etc.)
- Maturity
  - Minimum Age of Capability prior to use
  - Experience with Capability
  - State-of-the-Art Importance relative to Maturity
- Security
  - Communications
  - Operations
  - Development

2.2.5.6.2 Timelines and Timing Requirements Definition

Timelines must be developed in a flow-down from the timelines inherent in the mission activities which must be performed. Timelines establish the major scale of time which must be refined and allocated to specific entities and their services to perform. The timelines determine the timing and time requirements for specific events. The time requirements of individual applications and services should be established by trading off the timings that can be accomplished with time requirements that cannot be met, while insuring that the time requirements related to the mission activities continue to be met. Some of the factors of importance in timeline analysis are shown in table 2-8.

Table 2-8. System Timelines and Timing Requirements Summary

- Operations/Mission Timelines
- Flowdown Procedures from Timelines to Timing
- Applications Processing Timing
- Operating System and Services Timing

2.2.5.6.3 Performance Requirements Definition

Performance requirements must be defined in a flow-down manner. The requirements need to be capable of being traded off directly against one another and indirectly against elements in the other requirements areas. Table 2-9 summarizes the requirements that need to be developed.

Table 2-9. System Performance Requirements Summary

- o Qualitative
- o Quantitative
  - Power
  - Weight
  - Volume
  - Instructions or Operations per Second
  - Memory
  - Mean Time Between Failure (MTBF)
  - Mean Time to Repair (MTTR)
  - Heat Dissipation
  - Electromagnetic Interference/Compatibility (EMI/EMC)
  - Input and Output Rates
  - Throughput
  - etc.

The key parameter driving requirements performance for the entire processing system is throughput. Careful definition of throughput performance for the system, and allocation of throughput performance down to each processing element in the system is essential.

2.2.5.6.4 Cost Requirements Definition

Cost considerations are a primary driver of the design of the system, since the affordable costs limit the design of the system. Thus, they constitute legitimate requirements for the system. Cost goals are needed up front to focus development efforts and to establish when enough (affordable) development has been accomplished. Trades of total system performance on an end-to-end basis need to be performed using alternative architectural implementations (i.e., instantiations of this generic architecture), and a methodology on architectural tradeoffs is needed. The architectural factors to be considered will affect allocation of processing, memory and other resources, and must be made on an objective, justifiable basis. Table 2-10 summarizes the overall costs of interest that should be flowed down to lower level elements in the requirements analysis process.

Table 2-10. System Cost Requirements Summary

o Total Life Cycle Cost
o Developmental Cost Elements
o Unit Cost Elements
o Operation and Support Cost Elements

Examples of the results of this architectural tradeoff process might result in an allocation requirement (for example) that looks like the following:

> Sensor and control processing to meet real time demands will be performed on the vehicle, with enough assets allocated to perform the mission with sufficient redundancy. Ground based processing will provide the necessary planning, scheduling, interpretation and analysis of the data which demands heavy or expensive processing resources. The intermediate stages (off-vehicle or off-ground) will provide minimum computation redundancy and maximum required communications support.

2.2.5.7 Hybrid Methodology Requirements Summary

The Hybrid, Object Oriented, Structured Analysis methodology recommended and described herein is derived from an amalgamation of standard structured analysis techniques, merged with some of the advanced techniques developed in the object oriented and interactive development approaches. From OOA, our methodology gets the concepts of abstraction of entities with explicitly defined inputs and outputs, parts breakdown of lower level entities, informations hiding of internal processing and data usage, and some requirements inheritance. The interface requirements are based on the logical interfaces which address the ultimate system user of data and the provider of the needed data. Standards that need to be applied are identified by their profiles in the data repository underlying each data flow and entity. Performance requirements are determined for each user application and allocated to the lower level entities and the respective services. The development goal should be to establish objectively supportable minimum performance objectives (requirements) to be achieved with acceptable quality and minimal costs.

Development of requirements must address the concurrent use of all requirements, not just the primary performance requirements. Operation, support, quality and site adaptation requirements are equally important because they enable the system to actually function. Cost requirements are important because they enable the system to be obtained. It is important to understand the distinctions between the requirements and the designs, as will be discussed in the next section.

2.2.6 DIFFERENCES BETWEEN REQUIREMENTS AND DESIGN APPROACHES

There is a difference between the requirements and design views of a system. The requirements analysis process starts off from some user needs which can be (presumably) met by a new or modified system. These user system needs then get turned into requirements for the developer community. They may be turned into requirements by the user issuing a requirements specification in some form or by telling the contractor to prepare specifications which the developer and the users will validate. Validated requirements specifications will then be used by the contractor/developer to design and build the system. The overall process of working between requirements and design activities is shown in figure 2-9.

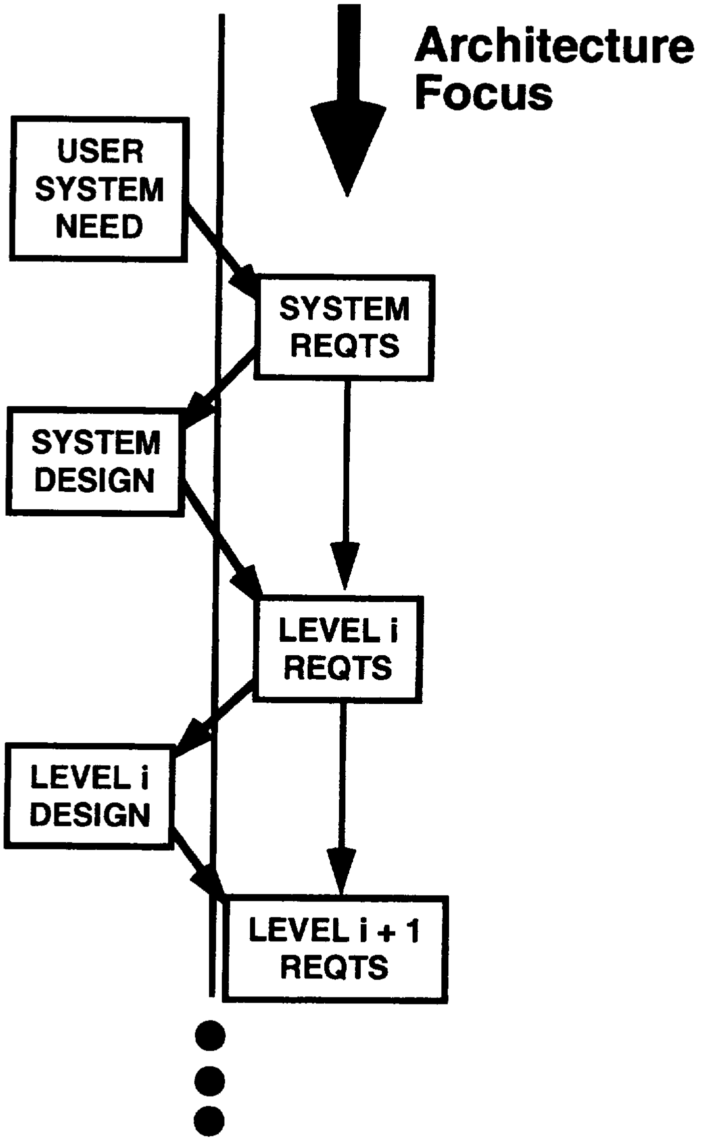


Figure 2-9.- Relationship of Requirements to Design Activities.

The process of requirements analysis is usually portrayed as one of determining the high level requirements and refining them to lower and lower levels of detailed requirements in subsystems, modules, components, and units. But actually the process looks more like the flow in this figure. After preliminary system requirements are developed, some system design is accomplished. Requirements at level 1 result in a level 1 design, both of which then drive lower level requirements at level 2. The requirements at level 2 must not only accommodate the level 1 requirements but also the design assumptions at level 1. This process (and its assumptions) in turn enable the next level of detail to be developed initially as requirements, which in turn lead to a design (or assumptions about a design). The process then repeats for lower and lower levels. Not shown in the figure, are the iterative loops that take place back and forth between requirements definition, design assumptions (which are attempts to see if the requirements at the same level actually are feasible and effective), and lower level requirements definition.

Although the focus in requirements analysis is on the right side of the figure, you cannot forget the left side, or the requirements will not work because too much valid material will be left out of the requirements. While considering design issues, however, it is important they be as minimal as possible and appropriate to the level of requirements being addressed. The design assumptions at each higher level drive some key requirements at lower levels; otherwise known as "design requirements".

2.2.6.1 Requirements vs. Design Determination

Since all requirements are derived based on the need of the human users, only the top level requirements directly traceable to these needs are actually "pure" requirements. All other lower level requirements have been derived based on some explicit or assumed higher level design knowledge, hence they are in fact design requirements. The key in defining requirements for systems is to not be concerned with artificial distinctions between "pure" requirements and design requirements, but to be concerned with establishing what is needed to eventually build the system. This will necessitate both types of requirements. The important thing is to not build in design features or assumptions too soon, and which are not appropriate to the level of requirements definition.

A key difference between requirements and design derives from the way requirements are identified. A proof of a requirement is that it answers the question, "What do you need." A proof of a design feature is that is answers the question, "How can it be done." After requirements (either "pure" or design) are defined, they can be allocated to the design at the same level, which provides the testability and traceability because the design (at the same level) can be tested to determine if the allocated requirements at that level have been met. Note that the design for a level need NOT be organized, developed or presented in a form that is parallel to that of the requirements. Requirements are grouped for the convenience of the requirements analysts and the designers who must both understand all the requirements for an entity. The requirements structure should not necessarily constrain the design as long as it meets the requirements content. The requirements structure aids understanding by all involved in building the system, it does not necessarily represent the design of the system. In fact, the design may and often will use different design entities to facilitate design improvements and features that are not being done to meet requirements.

Partitioning of avionics higher level entities into lower level entities needs to be based on explicit criteria. If an avionics entity is to be broken into entities A, B, C, etc, these entities can be selected based on what makes sense for understanding of the requirements in a requirements analysis, not what will necessarily ease the design. Requirements must be clear and understandable to the requirements analysts so that they can present clear requirements to enable the designers to know what is expected of their design product.

The partitioning between conventional control subsystems (such as GNC or C&T) and the operations control subsystem (Space Operations Control Subsystem) in the architecture has been clarified with the definition of criteria for determining whether requirements are part of one or the other. The primary criteria is the determination whether a specific process or data requirements serves a closed loop control functional need or an open loop functional need, with subsystems requiring astronaut input being treated as a open loop subsystems since the input is not from within the system. Closed loop (over time) control requirements have been defined to be part of the control subsystems such as GNC attitude control, while open loop control requirements (such as for Traffic Management) have been defined to be part of the operations control subsystem. This makes the allocation (in design) of open loop functions such as traffic management more effective since the interfaces between the open loop functions and the closed loop functions can be better defined by this approach. This does not imply that traffic management would not or should not be implemented (for example) in a GNC design. A function such as taking navigation data off laser gyros would appear to be open loop, but in fact is a closed loop functional requirement over time, since the gyros have to be resynchronized periodically to compensate for drift.

2.2.6.2 Lessons Learned

Some specific lessons learned in requirements analysis are that:

- Mental confusion between the difference between requirements and design features (i.e., "what" and "how") very easily leads to confusion over what is needed in the requirements, which leads to specifying of design features resulting in a poor design with subsequent and frequent re-work needed.
- Requirements statements should be listed with one requirement to a line with an individual unique identifier for each requirement.
- The specific CASE tools used are not nearly as important as the need to use CASE tools which have an integrated data repository, thus allowing the developer to establish requirements that include a linked data base without the distraction of having to develop a methodology for building that data base.

2.3 REQUIREMENTS PROTOTYPING AND SIMULATION

This section identifies the methodology practices for CASE tool-based analysis, describes the role of static hybrid analysis in prototyping, the changes that occur when transitioning to a dynamic modeling approach, and the approach that should be used for documenting the analysis and resulting specifications.

TBD in FY 92.

2.4 REQUIREMENTS PERFORMANCE ANALYSIS

This section identifies the practices for analyzing architecture performance, as a method of determining the performance of alternative architectures, and the performance merits of alternative requirements specifications.

TBD in FY 92.

2.5 DESIGN REQUIREMENTS DEFINITION

This section identifies the practices to be used in defining design requirements, i.e., the requirements which are dependent on previously established design assumptions at a higher system/subsystem level.

TBD in FY 92.

2.6 OPEN SOFTWARE ENVIRONMENT USE

The methodology for requirements analysis is based on use of open and generic standards and environments. One of the key complementary environments is the Open Software Environment (OSE) being developed by the Strategic Avionics Technology Working Group (SATWG). This environment will establish a set of specifications, standards and procedures common to all missions which must operate concurrently, with inherent upgradeability. Definition of entities and interfaces based on the OSE model can facilitate requirements definition for designs which have the open and generic characteristics needed. Figure 2-10 depicts the OSE model.

There are three types of entities used in the OSE Model: Application Software, Application Platform and External Environment. Application Software (AS) is the set of processes, data and associated performance parameters and documentation in electronic form related to a data processing system. Application Platform (AP) is the set of services and resources needed to run the applications. External Environment (EE) is the set of entities outside the boundaries of the entity of interest which need to exchange information with the entity of interest. The external environment includes permanent data stores, electronic communications entities and human entities.

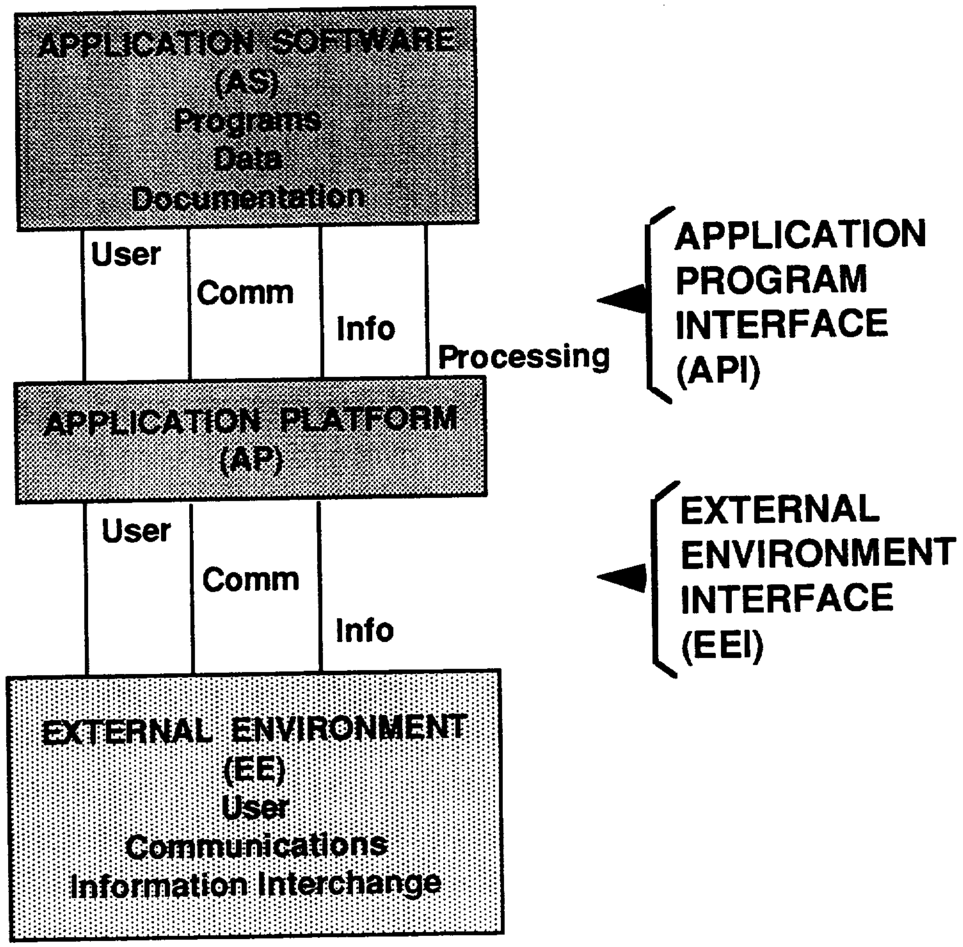


Figure 2-10.- Open Software Environment Model of Applications and Interfaces.

Applications Software interfaces through the API, Application Platforms interface through the External Environment Interface (EEI). The API interfaces are: User (the interface intended to provide access with the user), information interchange (non-communications language bindings to be provided through the Applications Platforms), communications (language bindings for services available to exchange state and information between Applications Platforms and Applications Software), and processing (language bindings for service communications available internally and not used for portability). The four types of interfaces used in the EEI are: User (physical access between the machine and human), information (language bindings for service using physical and logical file structure), and communications (language bindings for service for media definition, connectivity, and protocols for state and data).

The OSE model, shown in figure 2-10, using ASs, APIs, APs, EEIs and EEs can involve multiple subsystems. In our Space Generic Avionics architecture, as described in section 3.1, each subsystem application (e.g., GN&C Control, C&T Control and SOCS) is the Application Software. The central architecture consisting of processing hardware and system services are the Application Platforms. The subsystem application to system services interface is the API, which is implemented for communications through the Space Data System Services (SDSS) communications network services at the Open Systems Interconnect (OSI) layer 7. The Application Platform (i.e., avionics) to External Environment (i.e., the users, hardware sensors, effectors and communications devices) interface is the EEI, which is implemented for communications through the SDSS communications network services at the OSI layer 1.

The standard hardware architecture can be overlaid with the OSE interfaces, as shown in figure 2-11. This is another way of looking at the use of the OSE model. Standard Data Processors (SDP), the Multiplex Data Processors (MDP) and the Sensor and Effector Embedded Processors (SP and EP) are the host computers for the Application Platform and its services, as well as the Application Software. Communications from the SDPs over the core network, local buses and direct communications links are communications to other standard processing elements, hence are external interfaces. Communications within each processor (whether the SDP, MDP, SP or the EPs) is an internal interface (the API).

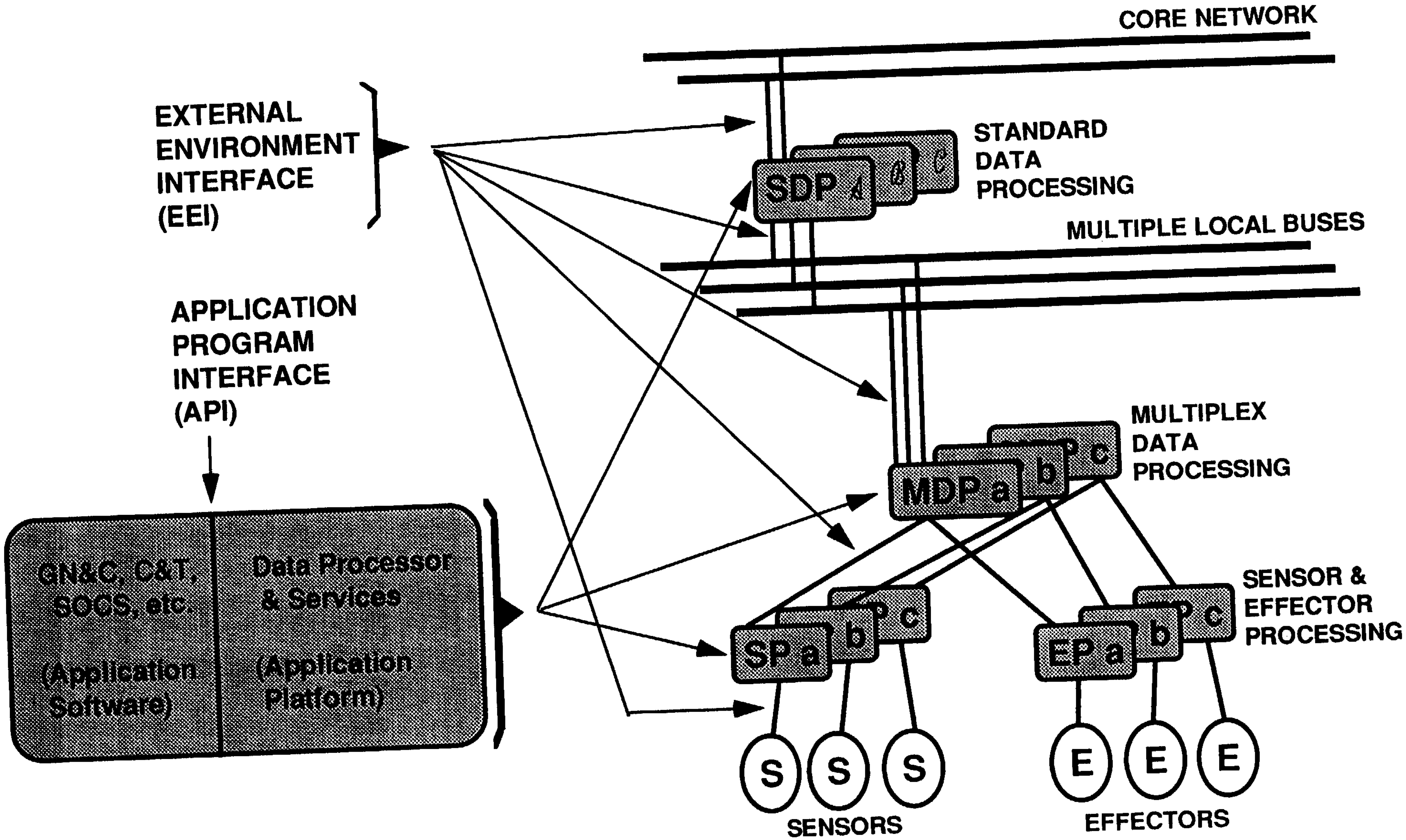


Figure 2-11.- Open Software Environment Interfaces
Applied to a Standard Hardware Architecture.

This is used in this methodology to create requirements at the appropriate level. System requirements are created with each entity for the data attributes needed by that entity or needed to be provided for some other entity; these data attributes are logical data flow requirements. They identify the source of the data and the end-user needing the data, as well as the characteristic attributes required of the data. They are not concerned with the mechanism for implementing the data requirements. The implementation related requirements for the interfaces are a physical issue relating to the mechanisms provided flowing the data from the source application to the end-user application. The OSE model addresses standards for the physical interface through the APIs and the EEIs. The source of the design requirements for the APIs, APs and EEIs (as previously noted) is the logical data attribute requirements and entity Applications Software requirements. This is illustrated in figure 2-12

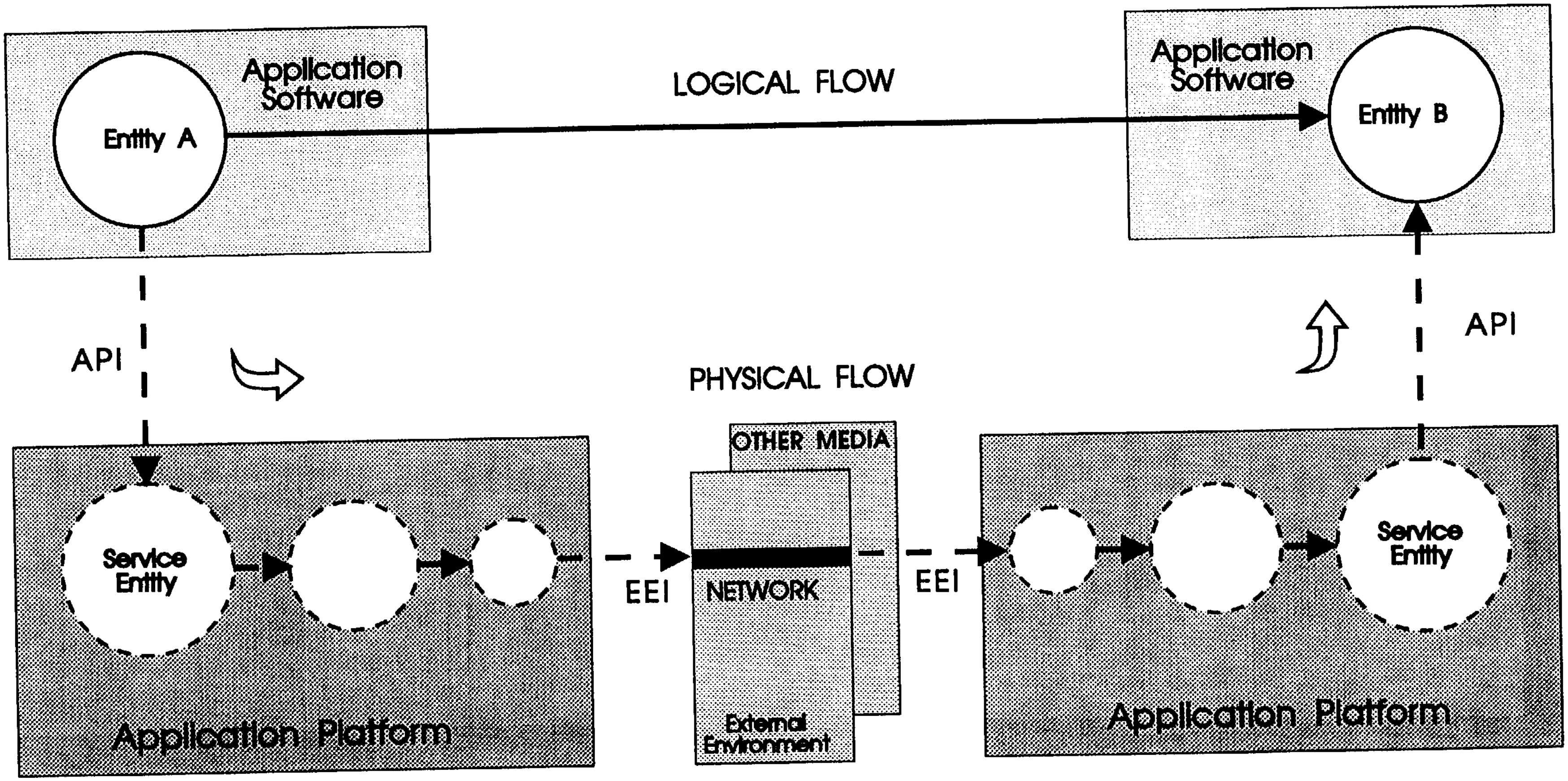


Figure 2-12.- Logical System Requirements Flowdown to Physical Design Requirements.

3. METHODOLOGY APPLICATION

The approach to performing a requirements analysis and design development is to consistently apply some basic techniques which both capture the user's requirements, and help drive out the unstated requirements needed by the user to operate an effective system. Requirements development must be based on application of the systems engineering process appropriately tailored to the phase of the target system's life cycle. Although requirements definition is recognized to be difficult, traceability between related requirements and subsequent design <u>must</u> be accomplished from the beginning of conceptual requirements analysis; if not, then it will be extremely difficult to backfill the requirements traceability later. Yet requirements traceability is the heart of design quality assurance and the key to proving to the user that the delivered system does meet the requirements.

The preferred method of assembling requirements analysis techniques and results into a traceable structure in this hybrid approach is to use automated systems engineering tools which include an integrated data repository. The requirements must be organized into a structure which enables both the developers and the users to understand the breadth and depth of proposed requirements and also the impact of the requirements on design and subsequent costs. This requirements structure for a target system is the requirements architecture of the system. This section identifies the target architecture used in developing the basic methodology, the specific practices and techniques followed in performing requirements analysis of this architecture, prototyping and simulation use in proving the effectiveness of the architecture and implementation of performance analysis in developing this methodology.

3.1 GENERIC ARCHITECTURE DEFINITION

A Space Generic Avionics (SGA) architecture was used as the target for development to establish the methodology by use in a real analysis development activity. The SGA architecture is shown Figure 3-1. Although the definition of avionics varies depending on its source, for this activity to develop a methodology, we used an avionics definition which assumed that the control subsystems for each of the more traditional subsystems (such as GNC or C&T) were within the avionics boundary while the hardware sensors and effectors were outside the avionics boundary. This was to

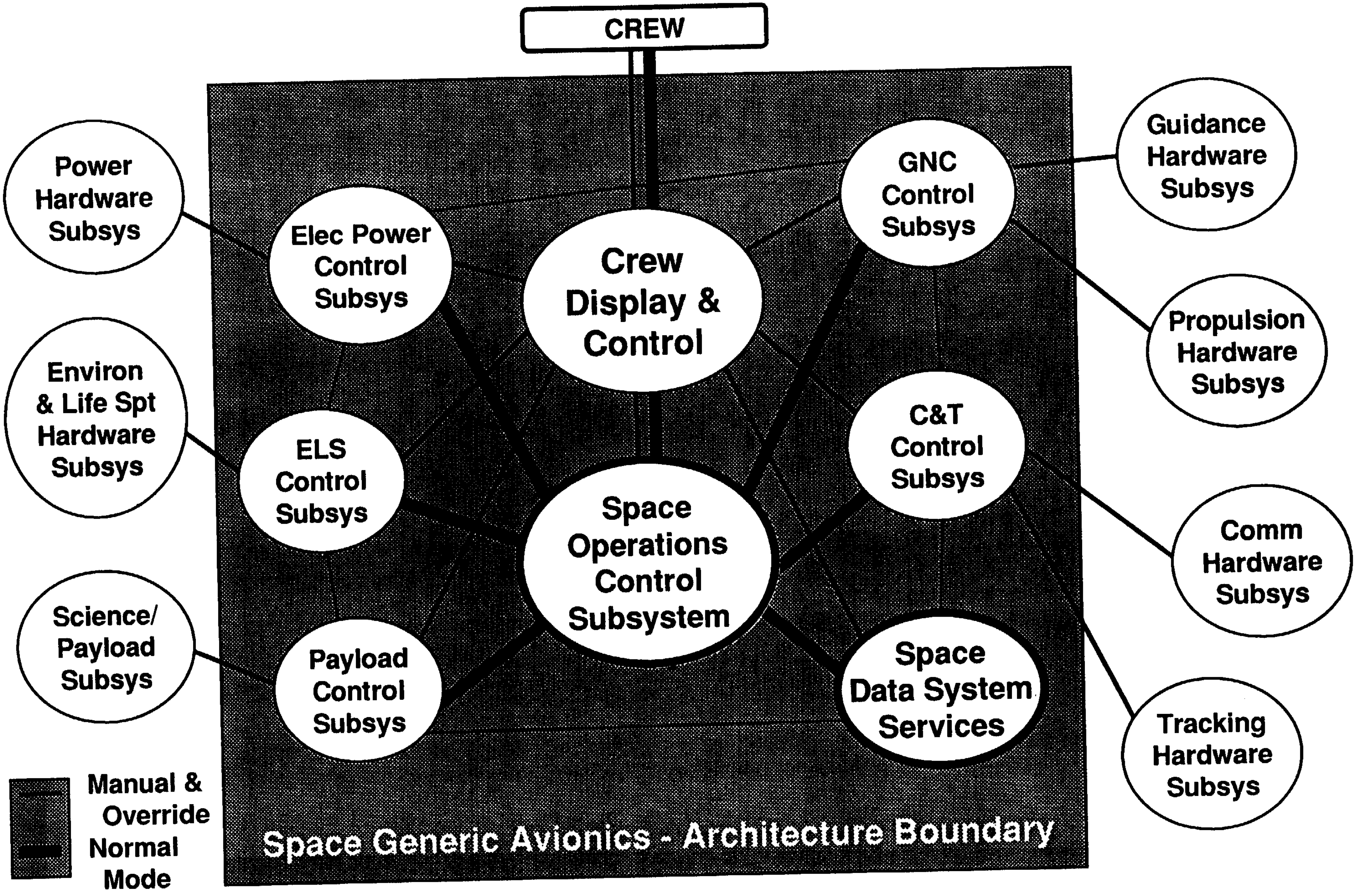


Figure 3-1.- One View of the Space Generic Avionics with the Assumed processing boundary.

facilitate boundary definition with its attendant conditions, enable a stronger focus on architecture development.

Since the NASA Johnson Space Center (JSC) divisions associated with each of these subsystems knows their subsystems well, it was assumed that a major focus of the architectural development should be the interstices of the JSC divisions (i.e., the interfaces between subsystems such as between GNC and C&T). It was also assumed that closed loop control over hardware subsystems would be less likely to change over the course of evolution from one mission to the next. The open loop control over operations activities would be more likely to change from one mission to the next, so the emphasis was on partitioning and clarifying the boundaries between closed loop control subsystems and open loop operations subsystems. These subsystems are both applications.

Another focus was to determine how to define the performance requirements in the architecture for the services needed to enable these applications subsystems to function effectively. Thus the darkened lines on the operations control application and the data system services bubbles and interfaces in the architecture in Figure 3-1 are depicted as another primary focus of this analysis. This focus provided not only a methodology, but also some value added avionics structure for operations control and data systems services. This diagram is not intended to suggest that these are the only interfaces of concern in a space avionics system, nor that the subsystems revolve around the operations control subsystem as a central point of control.

3.2 REQUIREMENTS ANALYSIS

The methods used in requirements analysis must provide for definition of the system concept, its requirements, the design requirements, definition of requirements prototypes and simulations, and determination and verification of performance requirements. This section summarizes the specific methods used in performing the requirements analysis (which is applicable to both conceptual and system requirements definition phases).

3.2.1 CASE Static Hybrid Object Oriented Structured Analysis

The first step in applying the static Hybrid, Object Oriented, Structured Analysis methodology lies in building modified data and control flow entity diagrams. Then control state transition diagrams must be developed to describe the control changes which govern system or subsystem operations. Finally, the data repository must be populated with real requirements data. This section describes these development processes.

To illustrate the building of such an analysis structure, the example of an avionics system will be addressed, with the simplification that avionics is treated as the electronics subsystems and human interfaces of a space vehicle and excluding the avionics hardware subsystems such as Guidance, Navigation and Control or Thermal sensors and effectors.

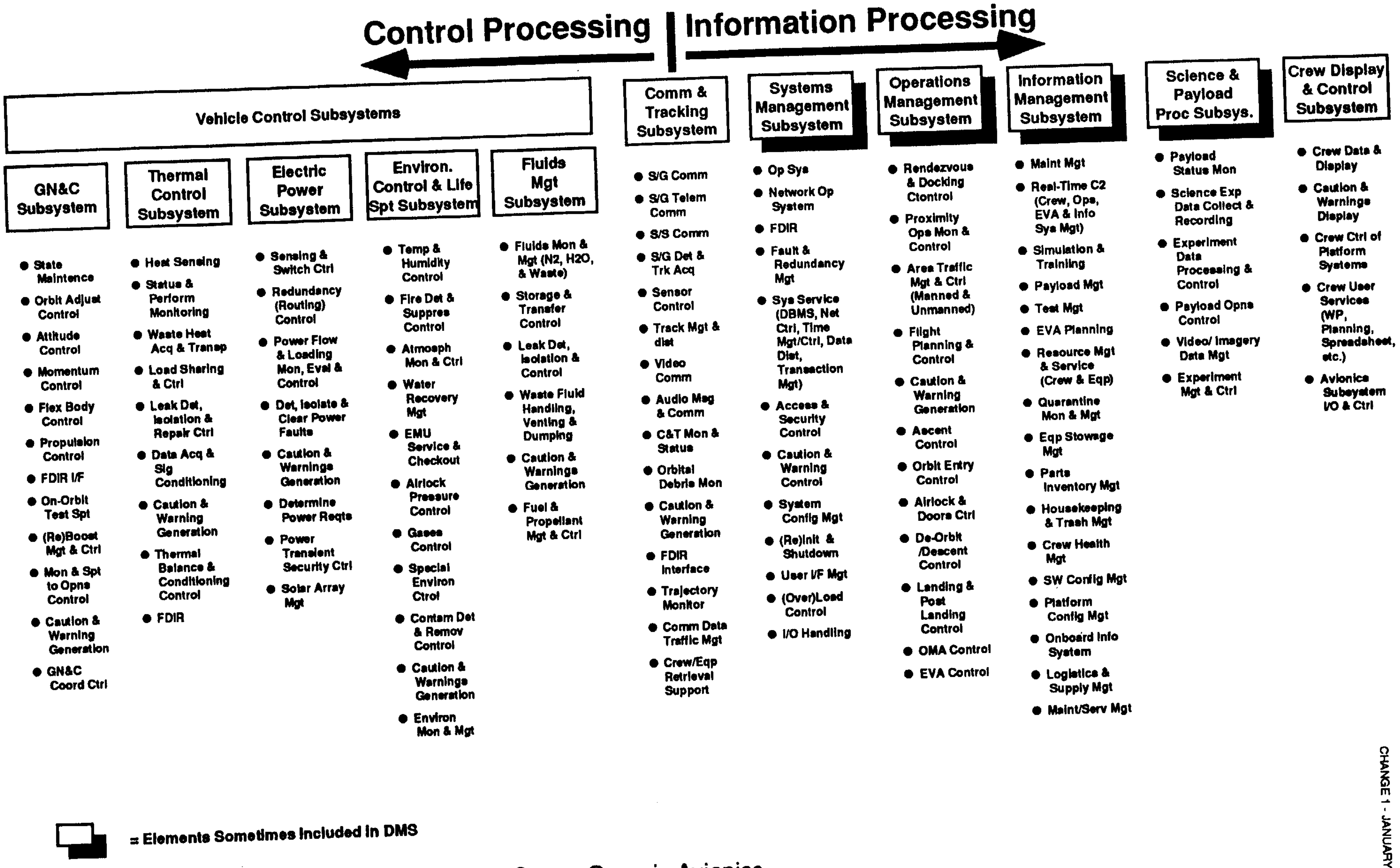
3.2.1.1 DATA AND CONTROL FLOW DIAGRAMS.

Merged data and control flow entity diagrams are often referred to as process flow diagrams, which are somewhat different in fact. This analysis will use bubbles to illustrate and describe the process entities being analyzed. The process entity bubbles may represent either data processes or control processes. Although referred to as data/control flow entity diagrams, the bubbles are thought of as entities (with noun names) to clarify they encompass more than just data or control processes, and include other requirements as previously described more related to object oriented development.

In some tool conventions (e.g., Excelerator), data processes are represented as solid line bubbles, while control processes are represented as dashed line bubbles. However the term "control" in many (if not all) automated implementations of CASE analysis uses a definition for a control process as one in which hardware or software components are directly controlled by the control process sending or receiving signals to activate, deactivate or monitor companion processes at the same level on a diagram. Thus control processes by this definition cannot be exploded into lower level controls, but only into control state or logic matrices representing the detailed truth tables for their connectivity. This restricts the analyst's ability to determine the requirements for a system since he cannot define a control structure and then determine the control entities which comprise it. For example, the use of a high level system management control entity is common, and is often exploded into entities such as operating system controls; network operating system controls; and fault detection, isolation and recovery (FDIR) controls. The use of control processes is represented in these approaches with a data process bubble (solid line shape) which is exploded into lower level controls also represented by data process bubbles, and only shows up as a dashed bubble when the lowest level is reached for controls with specific signals being sent and received.

The first step in creating data/control flow entity diagrams is to gather together all the processes, events, major data items, and any other information which can then be used to identify logical groupings (i.e., entities) of user information handling. These groups of information handling should define user needs for processes and the related data being processed. An example of a table of space system user information processes grouped together is shown in figure 3-2. These processes are not intended to be definitive space processes but simply a checklist of processes against which the subsequent requirements analysis can be measured to determine if all "traditional" space functions are accounted for or accommodated. The data and processes remain linked together in entities in this analysis. The logically groups of user information entities are then accumulated into higher level categories as suggested by the functional checklist shown in the figure. It is desirable for the major categories to relate back to the NASA organization groupings or work divisions to simplify interface development and control.

This functional check list also suggests some of the partitioning into higher level entities. A vehicle control entity appears needed to coordinate the subsystem applications operating on the vehicle and to deconflict their activities. This vehicle control subsystem would also provide a means of human operator coordinated control over all vehicle applications or operations. It also suggests that another higher level function needed is one of operations control to coordinate all activities and processing inside the space vehicle with each other and with outside activities and processes. This operations control would also provide the place for requirements for logic "glue" as needed to enable the activities and processes to respond to humans and which



Control Processing | Information Processing

Vehicle Control Subsystems

GN&C Subsystem

- State Maintence
- Orbit Adjust Control
- Attitude Control
- Momentum Control
- Flex Body Control
- Propulsion Control
- FDIR I/F
- On-Orbit Test Spt
- (Re)Boost Mgt & Ctrl
- Mon & Spt to Opns Control
- Caution & Warning Generation
- GN&C Coord Ctrl

Thermal Control Subsystem

- Heat Sensing
- Status & Perform Monitoring
- Waste Heat Acq & Transp
- Load Sharing & Ctrl
- Leak Det, Isolation & Repair Ctrl
- Data Acq & Sig Conditioning
- Caution & Warning Generation
- Thermal Balance & Conditioning Control
- FDIR

Electric Power Subsystem

- Sensing & Switch Ctrl
- Redundancy (Routing) Control
- Power Flow & Loading Mon, Eval & Control
- Det, Isolate & Clear Power Faults
- Caution & Warnings Generation
- Determine Power Reqts
- Power Transient Security Ctrl
- Solar Array Mgt

Environ. Control & Life Spt Subsystem

- Temp & Humidity Control
- Fire Det & Suppres Control
- Atmosph Mon & Ctrl
- Water Recovery Mgt
- EMU Service & Checkout
- Airlock Pressure Control
- Gases Control
- Special Environ Ctrol
- Contam Det & Remov Control
- Caution & Warnings Generation
- Environ Mon & Mgt

Fluids Mgt Subsystem

- Fluids Mon & Mgt (N2, H2O, & Waste)
- Storage & Transfer Control
- Leak Det, Isolation & Control
- Waste Fluid Handling, Venting & Dumping
- Caution & Warnings Generation
- Fuel & Propellant Mgt & Ctrl

Comm & Tracking Subsystem

- S/G Comm
- S/G Telem Comm
- S/S Comm
- S/G Det & Trk Acq
- Sensor Control
- Track Mgt & dist
- Video Comm
- Audio Msg & Comm
- C&T Mon & Status
- Orbital Debris Mon
- Caution & Warning Generation
- FDIR Interface
- Trajectory Monitor
- Comm Data Traffic Mgt
- Crew/Eqp Retrieval Support

Systems Management Subsystem

- Op Sys
- Network Op System
- FDIR
- Fault & Redundancy Mgt
- Sys Service (DBMS, Net Ctrl, Time Mgt/Ctrl, Data Dist, Transaction Mgt)
- Access & Security Control
- Caution & Warning Control
- System Config Mgt
- (Re)Init & Shutdown
- User I/F Mgt
- (Over)Load Control
- I/O Handling

Operations Management Subsystem

- Rendezvous & Docking Ctontrol
- Proximity Ops Mon & Control
- Area Traffic Mgt & Ctrl (Manned & Unmanned)
- Flight Planning & Control
- Caution & Warning Generation
- Ascent Control
- Orbit Entry Control
- Airlock & Doors Ctrl
- De-Orbit /Descent Control
- Landing & Post Landing Control
- OMA Control
- EVA Control

Information Management Subsystem

- Maint Mgt
- Real-Time C2 (Crew, Ops, EVA & Info Sys Mgt)
- Simulation & Training
- Payload Mgt
- Test Mgt
- EVA Planning
- Resource Mgt & Service (Crew & Eqp)
- Quarantine Mon & Mgt
- Eqp Stowage Mgt
- Parts Inventory Mgt
- Housekeeping & Trash Mgt
- Crew Health Mgt
- SW Config Mgt
- Platform Config Mgt
- Onboard Info System
- Logistics & Supply Mgt
- Maint/Serv Mgt

Science & Payload Proc Subsys.

- Payload Status Mon
- Science Exp Data Collect & Recording
- Experiment Data Processing & Control
- Payload Opns Control
- Video/ Imagery Data Mgt
- Experiment Mgt & Ctrl

Crew Display & Control Subsystem

- Crew Data & Display
- Caution & Warnings Display
- Crew Ctrl of Platform Systems
- Crew User Services (WP, Planning, Spreadsheet, etc.)
- Avionics Subsystem I/O & Ctrl

= Elements Sometimes Included in DMS

Figure 3-2.- Space Generic Avionics Potential Functions Checklist.

may not be obviously a part of an individual subsystem's applications. (To repeat an earlier point, this checklist is being used to develop a structure for gathering requirements, which can then be used for an integrated view of the avionics requirements.)

The shadowed elements in Figure 3-2 represent functions sometimes required of present vehicle (i.e., Space Station Freedom or Shuttle) data management systems (DMS). Note that this structure does not correspond exactly to the station software partitioning, and is not precisely the same as the implied partitioning of the station requirements specification in JSC 31000, Vol 3, Rev E. Its purpose is to focus analyst attention on data service requirements and related or ancillary processing requirements.

Other systems on a platform, not treated as part of the avionics but which are closely coupled to or controlled by the avionics, include the systems such as the platform structure, power/propulsion, environmental control, the crew, fluid mgt system, etc.

The next step is to develop the Level 1 Context Diagram. This diagram depicts the highest level view of the system, where the system is a black box in its environment, with the emphasis here to define the external interfaces and the information needed by the system or by the environment from the system. The entities in the external environment should reflect real world entity abstractions (such as shuttle) or platform real entity abstractions (such as thermal hardware subsystem). Partitioning between the system as black box, and the external entities should be based on explicit criteria (e.g., external entities to avionics are the hardware and embedded firmware for the subsystem sensors and effectors). The interfaces should be defined simply in the diagram using the naming conventions in the appendix, and should carry the names of both the system as black box and the external entity to enable data base searches keyed on names. The CASE tool data repositories are usually name sensitive, so this enables use of the repository to store and retrieve data needed by external elements and by the system as black boxes.

Then the subsequent step is to develop the next level of diagrams (level 2) which define the system entities, their processes, and their attributes in diagrams and their performance and standards requirements in the data repository. The higher level interfaces on the level 1 diagram should be carried down to this diagram, with the level 1 data flows being broken open (exploded) into more detailed data flows relevent to each of the system entities used in the system diagram. Key system elements should be identified, organized by clearly distinguishable categories based on explicit criteria. The data flows between system entities define the needs for data by each system, and the needs to generate data by each system.

All subsequent steps involve developing lower level diagrams (level 3+) with entities representing subsystems/subprocesses, their attributes, their performance requirements and applicable standards. At each level, entities and data flows must be defined based on objective criteria. The bottom level will have been reached when relatively "primitive" entities represent simple processes and data flows can be identified. Primitive entities can be easily described in "primitive process specifications" as part of the explosion process in the CASE tools. Definition of primitive processes depends on a specific subsystem being defined and the depth of knowledge existing on its requirements.

Some specific notes about the explosion process used in defining lower level diagrams in a Hybrid, Object Oriented, Structured Analysis are addressed next. The same level (e.g., level 5) in different trees (e.g., one in vehicle control and one in system control) are not necessarily equal, because the decomposition process in different trees may be attempting to achieve different local objectives, describing different requirements. Each tree structure is intended only to aid in the understanding of the requirements for the set of elements in that tree.

It is important to distinguish between logical and physical data flows. While logical flows usually only show originator and end user of data, they may need to show physical entities such as data stores if the data stores are an intermediate "user" by holding data for long periods of time (which acts to partially de-couple the source from the ultimate end-user). While physical flows usually show routing of data between two entities, they may also only act as pass-through entities. Usually logical flows are associated with requirements and physical flows are associated with design.

If only requirements are wanted, then analysis should only address logical flows. The OS services normally address physical flows, however, their purpose is to support the applications. An OS must be designed, which means the OS requirements need to be defined. Since the OS doesn't care what services are provided or data is passed, analysis can develop logical OS services and flows based on generic or categorized data being passed. In this case, the OS physical flows are also logical flows. While the applications requirements derive from the users' needs, service requirements derive from the applications needs. Thus, in a complete architecture, the requirements for services as well as for applications need to be defined.

Logical flows may have constraints (i.e., requirements) imposed on them by higher level processes in their own tree or by processes in any other tree at any level which interacts with the flow.

The use of data stores in a logical structure distinguishes between two cases: (1) where data goes from originator to end-user because such passage benefits both entities, and case (2) where data goes from originator to data store and independently goes from data store to end-user because the source does not care who uses the data and the user does not care where the data comes from as long as it is timely and accurate. A user of data must either (1) select the data needed based on real time considerations for similar data and based on predefined criteria, or (2) select the data needed based on specific predefined items to be used at specific predefined points in processing. If real time criteria are used, they must be subject to adjustment to enable the crew to make changes based on their specific mission needs during the flight profile being flown.

After data and control flow entity diagrams are built, they must be informally validated or tested by the developer to determine if the logic is sufficiently robust to stand against changing requirements as a baseline for a time. If every change in requirements causes a architectural change (i.e., a change in the data and control flow entity diagrams) then the system is insufficiently defined for use. One technique to validate the logic of the analysis and breakdown diagrams is to create operating thread diagrams as describe in a later section.

3.2.1.2 Control State Transition Diagrams

State transition diagrams are used to develop the requirements for the control logic involving changes from one steady state of operation to another in response to commands from the crew. They involve developing stimuli-response tables for each command stimuli the crew may initiate, and linking the crew commands to operating and system responses.

TBD in FY 92.

3.2.2 TECHNIQUES FOR REQUIREMENTS ANALYSIS

The specific techniques needed to perform requirements analysis include those described below.

3.2.2.1 Entity Partitioning

The partitioning of the system and lower level entities into subsystem and process oriented entities must accommodate not only the standard structured analysis partitioning addressed previously, but also partitioning of groups of functional topics into related multiple layers with cascading functional capability.

This means, for a functional area such as safety, that there is a manager function, a lower level controller function, and yet lower level handler subfunctions. Consistent application of such partitioning is needed for requirements consistency, and for determination whether the requirements are complete. The analysis for this methodology has derived a structure of four layers to accomplish this:

1. Mission
2. Vehicle (including systems)
3. Subsystem
4. Component.

Thus the safety functional area could be divided into a mission safety manager, a vehicle safety controller and a subsystem safety controller. There is no component safety controller, which indicates that not all four layers are required. Another example is fault handling: there could be a vehicle fault manager, a subsystem fault controller, and a component fault tester (known as built-in-test - BIT). The functions in these examples could also be linked by noting that the vehicle fault manager supports the vehicle safety controller by providing fault alerts and warnings.

3.2.2.2 Facility to Vehicle Partitioning

Another partitioning approach that must be supported is a capability to partition and allocate entities or functions to any reasonable type of facility implementation of processing. The range of processing may stretch from standalone space platforms with no dependence on Earth-based facilities, to vehicles with high dependence on Earth-based facilities.

For example, a Lunar or Mars base might be established by the requirements analysis to be entirely self contained with no dependencies on Earth-based processing or mission control. A Lunar Transfer Vehicle might have extensive on-board GNC and Operations Management System (OMS) processing or might just contain implementation software for the processing determinations computed on Earth and relayed to the vehicle through an Integrated System Executive (sic - ISE) determined on Earth. Alternative allocations are possible based on different sets of criteria, depending on the level and types of decisions being made, whether as policy decisions out of NASA Level 1 or 2, or as technical decisions from timeline analysis.

The methodology established in this paper must support any type of allocation without redoing the analysis. Thus, for a Command Control function, figure 3-3 might represent the entities to be processed in Command Control. The mission plans manager could be implemented by allocating the requirements to the space vehicle or by allocation to the mission control center. Alternatively, some of the requirements could be allocated to the space vehicle (e.g., select keys), some to the mission control center (e.g., generate timeline) and some to the "back room" support personnel (e.g., coordinate the plans library entries). The requirements for this subsystem entity would be essentially the same in any case.

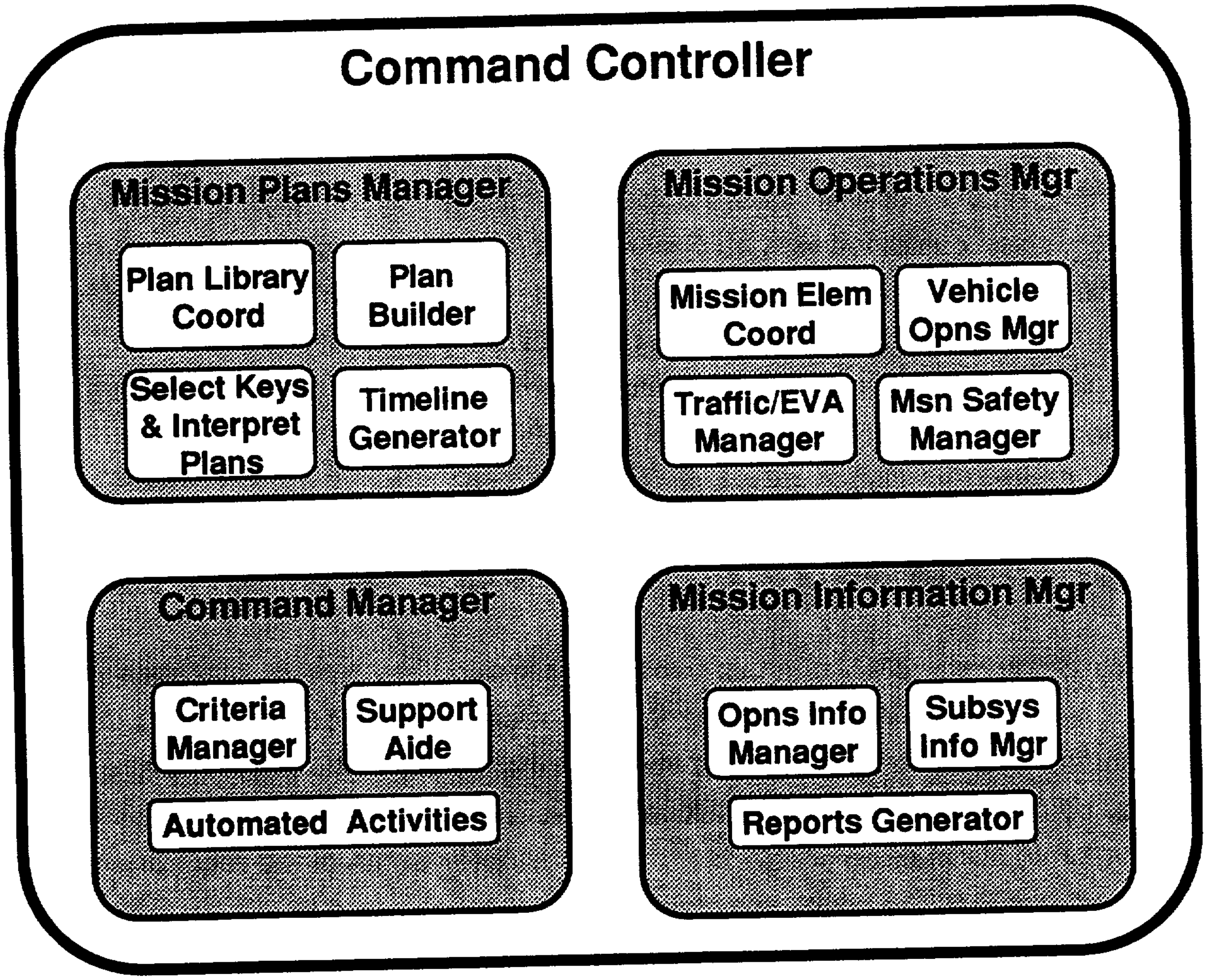


Figure 3-3.- Operations Control May Span Alternative Allocations.

3.2.2.3 Operational Thread Use

Operational threads are needed to validate the logic of the analysis and breakdown diagrams. The threads need to focus on how the crew or the system would perform specific operating tasks, not on simply how a entity or process executes. The intent is to focus on crew-task interactions, not just on entity or process interactions. Thus, for instance, if a task is "crew fires thruster n", then the thread to perform this task might look like figure 3-4. This provides an operationally oriented means of discussing the logic of the task with operational personnel to validate the entities and processes created in the analysis.

Another example involves developing an operating thread for a crew-system task, namely how the system responds to a resource failure with crew interaction. Thus, figure 3-5 identifies this thread. This thread seems to emphasize failure detection as a key element. Analysis of failure detection might then determine there are several alternative means of detecting failures as discussed in the next paragraph. In addition to being used to verify that all the needed entities and processing/attributes have been identified in their proper places, these alternatives give rise to a Fault Handling Architecture, which must also be reflected back into the Hybrid, Object Oriented, Structured Analysis diagrams.

Alternatives which must be considered for detecting failures (for example) might include managing the types of testing available to perform fault detection, controlling the schedules of testing to prevent interference with operational mission activities, running tests at low levels in hardware, determining trends indicative of failure even though individual tests do not show a failure. Figure 3-6 represents an architecture derived as a result of this analysis and reflected in the Hybrid, Object Oriented, Structured Analysis developed for this methodology.

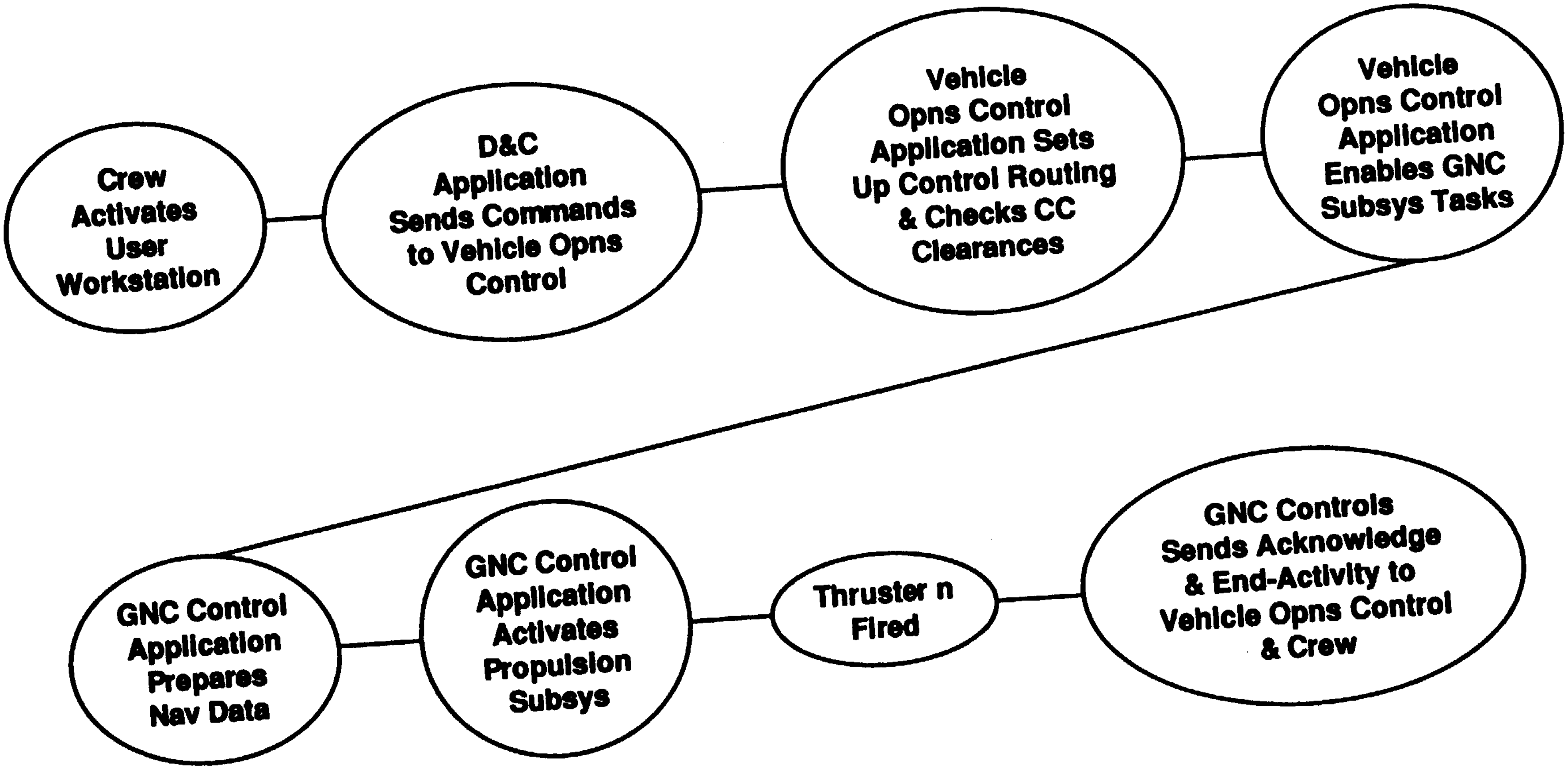

Crew Activates User Workstation
D&C Application Sends Commands to Vehicle Opns Control
Vehicle Opns Control Application Sets Up Control Routing & Checks CC Clearances
Vehicle Opns Control Application Enables GNC Subsys Tasks
GNC Control Application Prepares Nav Data
GNC Control Application Activates Propulsion Subsys
Thruster n Fired
GNC Controls Sends Acknowledge & End-Activity to Vehicle Opns Control & Crew

Figure 3-4.- Operational Processing Thread for "Crew Fires Thruster n" Task.

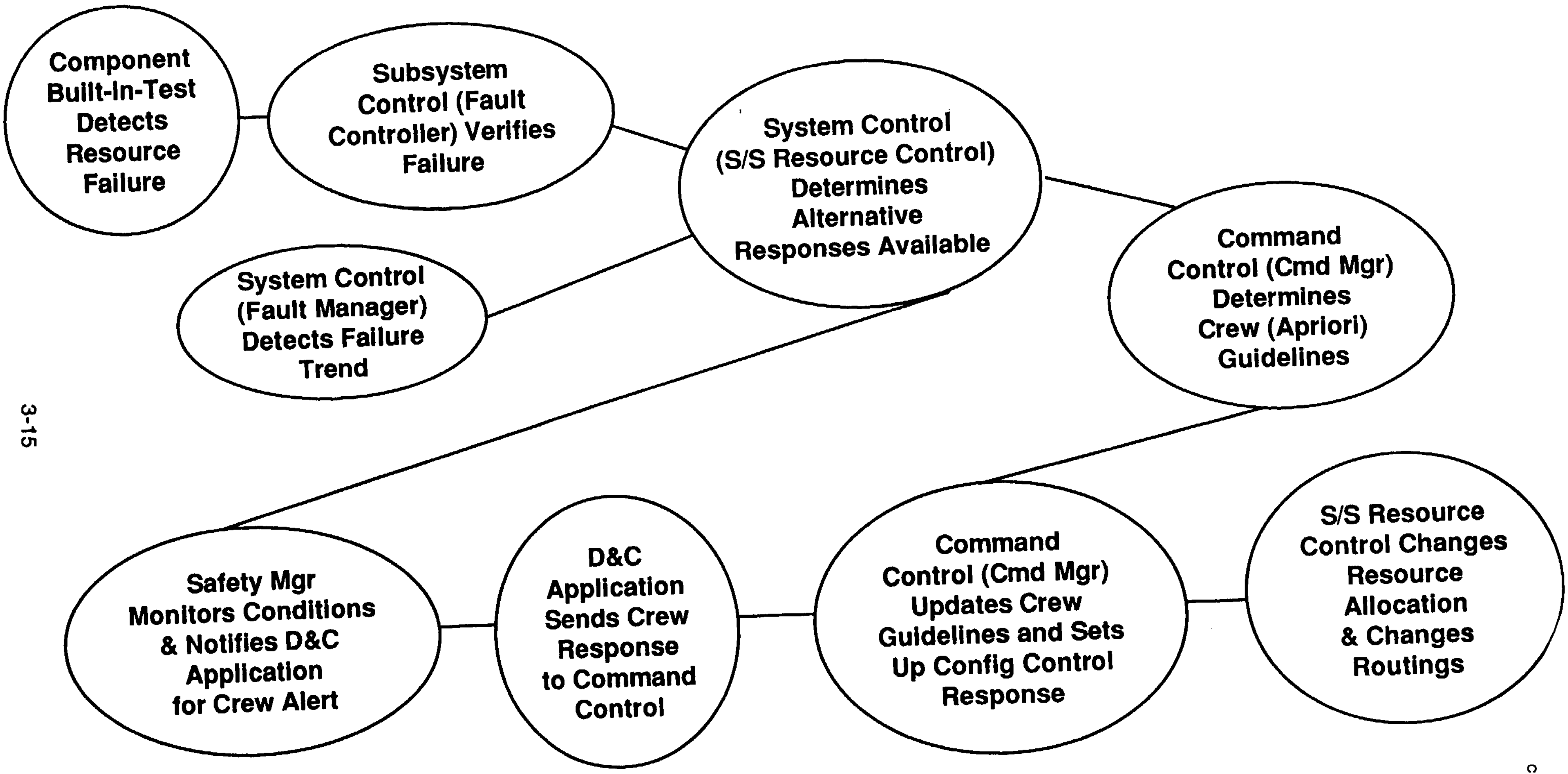

Component Built-In-Test Detects Resource Failure
Subsystem Control (Fault Controller) Verifies Failure
System Control (Fault Manager) Detects Failure Trend
System Control (S/S Resource Control) Determines Alternative Responses Available
Command Control (Cmd Mgr) Determines Crew (Apriori) Guidelines
Safety Mgr Monitors Conditions & Notifies D&C Application for Crew Alert
D&C Application Sends Crew Response to Command Control
Command Control (Cmd Mgr) Updates Crew Guidelines and Sets Up Config Control Response
S/S Resource Control Changes Resource Allocation & Changes Routings

Figure 3-5.- Operational Processing Thread for "System Responds to Resource Failure" Task.

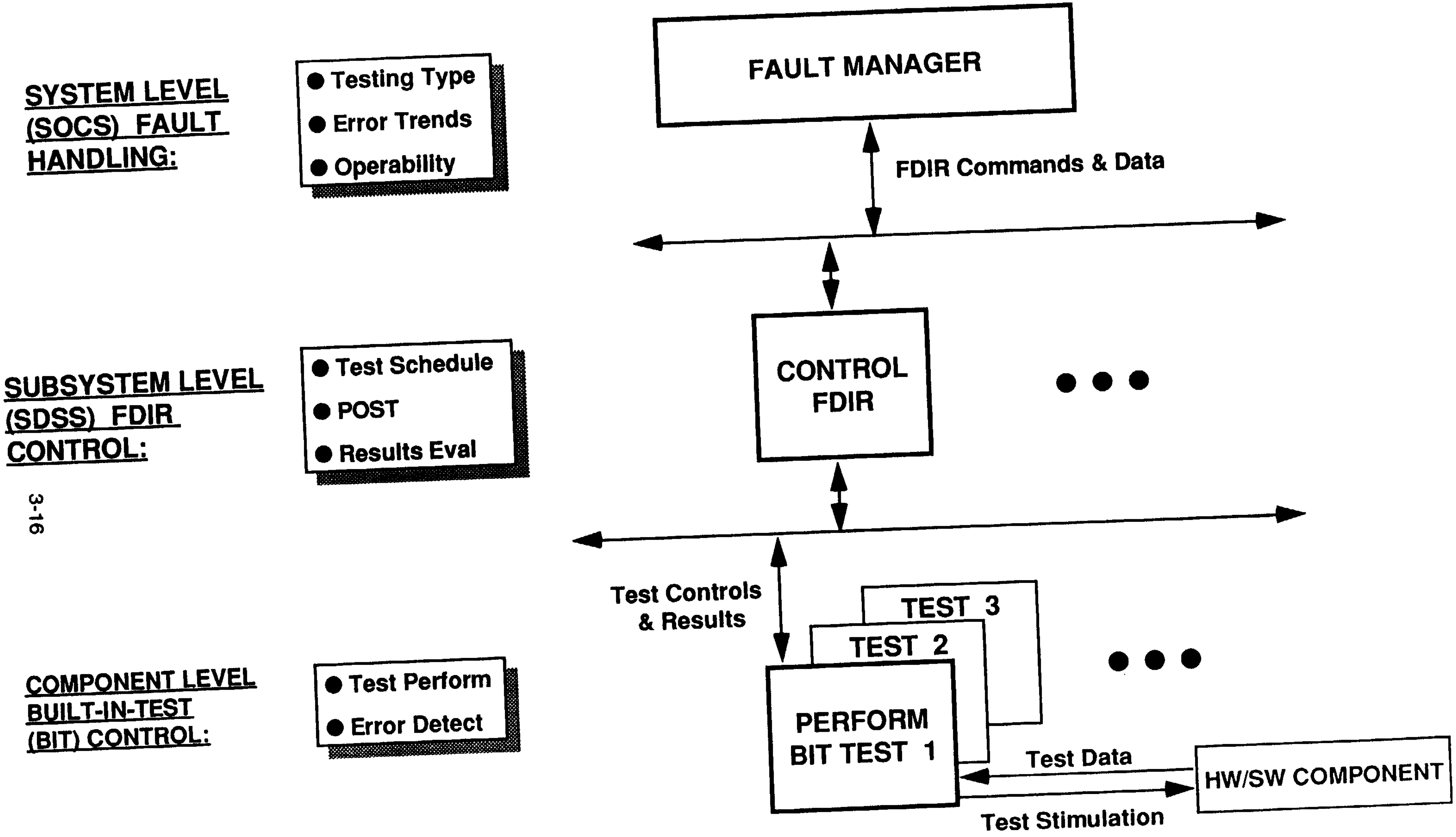


Figure 3-6.- Fault Handling Architecture Example.

3.2.2.4 Risk Management Requirements Definition

(TBD)

3.2.2.5 Fault Tolerance/Redundancy Management Requirements Definition

(TBD)

General requirements for fault tolerance, failure handling, redundancy management, and recovery will be addressed, and unique requirements to tailor the general requirements to a specific platform will be discussed.

3.3 PROTOTYPING AND SIMULATION IMPLEMENTATION

TBD in FY 92.

3.4 PERFORMANCE ANALYSIS IMPLEMENTATION

TBD in FY 92.

APPENDIX A
DEFINITIONS, ASSUMPTIONS AND CONVENTIONS

APPENDIX A
DEFINITIONS, ASSUMPTIONS AND CONVENTIONS

This appendix describes the definition and conventions adopted in this methodology.

A.1 DEFINITIONS

The terminology used in developing this methodology is defined herein, based on industry standards wherever feasible. Determination of the scope of architectures, avionics, systems, services and applications depends to some extent on the definitions accepted for these items since definitions can focus attention or exclude attention.

A system is the composite of equipment, material, computer software, personnel, facilities and information/procedural data that satisfies a user need. (Electronic Industries Association Bulletin SYSB-1)

The System Engineering Process is the methodology of sequential and iterative application of selected scientific and engineering efforts to convert user needs into a system solution that will best satisfy the requirements and constraints in accordance with "agreed to" effectiveness measures reflecting the users needs. The process needs to be divided into the macro steps of system requirements analysis and system design. (Electronic Industries Association Bulletin SYSB-1)

An Open Systems Architecture is defined as a structure of interconnected functional subsystems (i.e., black boxes) using non-proprietary communications, based on open interface standards, i.e., standards that are complete and consistent, published and accepted by a publicly accessible review body. Interfaces to the target's operational environment must be based on open interface standards. The architecture must be extensible through the addition of subsystems, services and resources following published rules. It must be precisely described and maintained by an openly accessible oversight board. Different vendor black box subsystems should be able to be added without knowledge of the subsystem internal structure or design. It consists of a system hardware architecture and a system software architecture.

A System Hardware Architecture is a set of hardware resources in a configuration of distributed computers, memories, buses and network elements. Some of the characteristics that determine the nature and requirements for a system hardware architecture are the number of processors, their type and topology, the speed and size of shared memory available, the local memory of each, the bandwidth and access to communications media, and the interfaces available for use by people, applications and platform software services in the hardware.

A System Software Architecture is the set of system functions performed by the applications software, and the structure of the platform software services that enable the applications software to perform their tasks. The functionality described by the system software architecture are the tasks which are required of the system to meet the needs of operational users.

A Laboratory Architecture is defined as a structure which is capable of being configured to represent a subject open system architecture. Thus, it must include (but not be limited to) non-proprietary standard communications, processing and interfaces. Interfaces to a simulation of the subject's operational environment is included. The lab architecture must include instrumentation, benchmarks, test/simulation controls, displays, and data analysis capabilities. It must be extensible through the addition of subsystems, services and resources following published rules. It must be precisely described and maintained.

An Avionics System is defined for the purpose of this methodology as the set of all electronic and processing based subsystems on a space vehicle, including all hardware, software and other electronics needed to control and operate the space vehicle. It is the collection of capabilities that provides the coordinated functionality for end-to-end processing in handling the information needed to know the platform's elements, to control its interaction with its environment, and to respond to human commands. Avionics provide for information acquisition, transmission, and storage of analog or digital signals and include the sensors, intra-platform communications, processing hardware, software and subsystems, data storage, human-machine interface subsystems, and response actuator controls used in the platform. (Adapted from JSC 31000, Vol 3, Rev E, Para 3.1.24.1.1)

The Space Generic Avionics (SGA) architecture is defined as the target Open Architecture Standard being developed to establish the preliminary methodology. It is a generic architecture, meaning that the elements of the architecture do not depend on any one mission or program for their definition. The elements of the architecture can be tailored to apply to many different space missions and programs. Tailoring may result in subsets of requirements applicable to a mission or program, but will retain architectural interface compatibility. The initial focus of the SGA architecture is for Space Vehicles; Other-Planet Bases are part of the SGA architecture but are not being addressed for now until the initial SGA architecture has been developed.

An entity is an abstract element that represents a real world entity, its data attributes and essential services with their respective performance and quality characteristics. It is used similarly to the term object from object oriented analysis, without intending to convey the implicit assumptions associated with object by practitioners of object oriented analysis and design .

A distributed system is a collection of computers, memories, buses and networks that are concurrently operating in a cooperative manner and communicating with each other. The system may be tightly coupled with shared memory capability or loosely coupled with messages used for coordination.

A data process subsystem is a subsystem with embedded data (determined by naming practices) and processing services, decomposable into lower level data process subsystems. Requirements attached to a data process subsystem are inherited onto lower level subsystems. It is setup and controlled by a runtime operating system.

A management subsystem is a data process subsystem which may interface to a human to determine options and select alternatives for implementation. A management subsystem which has no human interface may support one which does have a human interface, or it may be an artificial intelligence capability which replaces a human, perhaps in unmanned missions.

A control subsystem is a process which selects and implements alternatives based on a-priori criteria or real time guidance from a management subsystem. Control subsystems may be decomposed in lower level subsystems. A control function usually implements a unique avionics capability. Requirements attached to a control subsystem are inherited onto lower level control subsystems.

A handler subsystem is a data process which implements a predefined, directed procedure, either from a control subsystem or a management subsystem.

A service subsystem is a process which implements supporting alternatives transparently to the using control or data process subsystem. Service functions are usually widely replicated in support of many control or data process subsystems. This wide replication of functionality is a key determining characteristic in defining an individual process as a service in this methodology. Services are critical to system operation, not to mission or vehicle operation per se. An example of a service function is a Report Generator since many applications and control subsystems must generate reports; here, they call on the report generator service which knows how to look up the table defining the applications/control report, how to format the format for completion, how to find the data to fill the report fields with, and how to route the report for distribution based on a predefined distribution list.

A logical interface is defined as the characteristic requirements associated with an interaction between a source of data and the end user of the data. Data is used by an entity in a logical manner if it makes a significant transformation, conversion or operation on the data.

A physical interface is defined as the routing requirements associated with passing data from the source of the data to the end user of the data. Data is used by an entity in a physical manner if it passes the data on without changing the data; thus for example, network operating systems are physical interfaces to applications because they package or unpackage data and send it to another network node.

Concurrent engineering is defined as the application of multiple engineering disciplines to develop requirements in several different but related areas at the same time so the requirements are coordinated and mutually supportive.

A.2 ARCHITECTURE ASSUMPTIONS

The architecture development was used as the vehicle for determining what practices actually worked which should be included in this methodology. Assumptions about the architecture were necessary to permit continued development of concepts and entities, and were selected to place as little restriction on the underlying methodology as possible. However, in case they may have constrained the methodology, they are identified below. This section summarizes the architecture assumptions in three categories: those assumptions related to the operation of a space platform, those related to the processing to be performed, and those related to how the structure of the architecture was to be assembled.

A.2.1 OPERATIONS

- Human control requirements can vary. Direct links from the human entry systems to the sensor and effector firmware/hardware or any intermediate point on the processing chain may be needed for emergency and manual backup purposes. The range of control must accommodate any level of capability from manual to fully automated (e.g., through artificial intelligence aids similar to the Lockheed Pilot's Associate being developed for the U.S. Air Force).
- Operations control requirements must span the range from on-board controls to mission control center to the "back room" control support. Partitioning between these facility control requirements should be done when applying the requirements to a specific platform or mission, or should be delayed until a design implementation is being prepared to maximize developer flexibility.

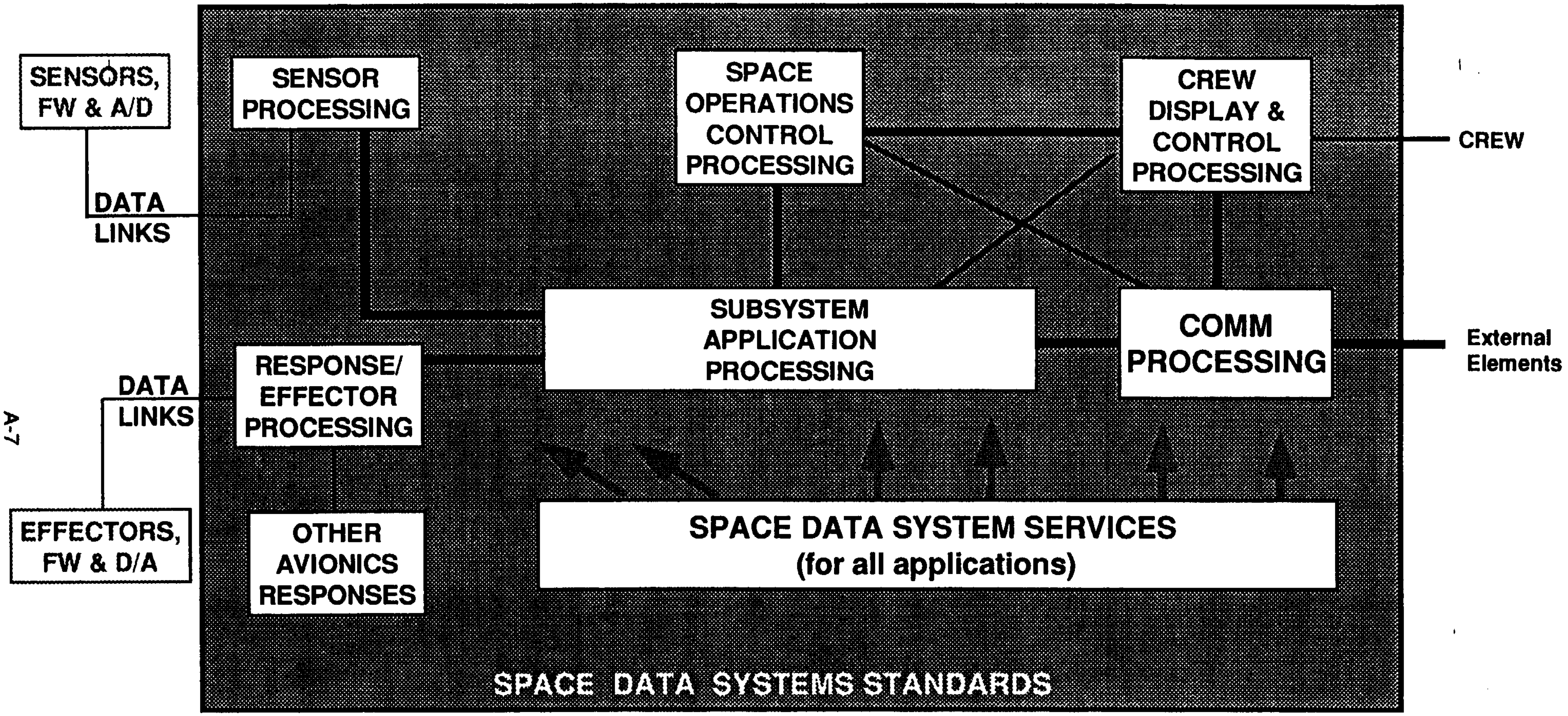
A.2.2 PROCESS

- The architecture must enable objective definition and interoperable processes for each entity selected for inclusion in a specific instantiation of the architecture for a specific platform.

- All entities have processes which can be applied to multiple vehicles with control parameters used to adjust between the same type process used in different classes of vehicles.

- Sensors and effectors are assumed to have firmware embedded in them for low level hardware control; this firmware processing is treated (for requirements and design purposes) as an integral part of the hardware. Sensors firmware processing may also enable or disable hardware, monitor power drain, monitor for abnormal conditions, implement built-in-test (BIT) of hardware and store results (these may alternatively be performed in the intermediate processors as described below).
- The architecture must handle alternative forms of processing and alternative allocations of these processes to different elements of the overall system for each mission-design. Low level processing is assumed to be embedded in sensor and

effector heads; such processing in firmware will be relatively "dumb" with sufficient capability to gather data, format it for transmission, and route it to appropriate controllers. Intermediate level processing includes processing such as sensor signal processing, effector response actuator processing, post sensor processing (e.g., track processing), multiplex data processing etc.; such processing is treated as a high level control structures (i.e., Control Application Programs) requiring some decision making capability to implement one of a number of alternative hardware control parameter sets in an intelligent system. High level processing includes two types of processing: one which provides a capability for the crew to control the vehicle or facility, and one for internal systems control of all activities. Processing such as needed for systems control, vehicle control, integrated logistics control, crew management, etc. are treated as high level command structures (i.e., Command Application Programs), which require interaction with humans and some capability to present alternatives to humans, and to interpret ambiguous responses from humans. Command application programs provide both types of high level processing. Figure A-1 depicts the processing architecture assumed which the Space Generic Architecture must handle, and which the methodology for development must be capable of analyzing.


SENSORS, FW & A/D
DATA LINKS
SENSOR PROCESSING
SPACE OPERATIONS CONTROL PROCESSING
CREW DISPLAY & CONTROL PROCESSING
CREW
SUBSYSTEM APPLICATION PROCESSING
COMM PROCESSING
External Elements
DATA LINKS
RESPONSE/ EFFECTOR PROCESSING
EFFECTORS, FW & D/A
OTHER AVIONICS RESPONSES
SPACE DATA SYSTEM SERVICES (for all applications)
SPACE DATA SYSTEMS STANDARDS

FW = Firmware Processing
A/D = Analog-to-Digital

Figure A-1.- Potential Process Partitioning included by the Space Generic Avionics.

A.2.3 STRUCTURE

- One of the purposes of this methodology is to enable the creation of an open, generic, standard architecture which can be tailored and reused for multiple missions. The methodology will then provide guidelines for doing the tailoring to create mission specific instantiations of the architecture. The reuse of the architecture and its components will become the standard way of developing new space data systems.

- The basic architectural guideline for differentiating processing levels is based on the philosophy of "Centralized Command and Decentralized Execution"

- The architecture must be a "shopping list" of all possible processes applicable to any space vehicle or other-planet base.

- Some entity processes only apply to a specific class of vehicle. Such special entity applications should be built into the naming conventions if feasible to more clearly convey the dependency of the entity application to the specific platform. The definition of entity names must use unique names for each entity for clarity and for tool searching of dependencies.

- The software principles of abstraction, information hiding and modularity are applicable to systems development and will provide the same benefits to requirements analysis and system design as they do to software analysis and design. Use of such principles will improve the maintainability and reusability of the architecture developed and used as the example for this methodology. Improvements in maintainability and reusability will not be allowed to reduce the requirements for performance which may be necessary; proof through architectural simulations must be provided that performance of an architectural instantiation is acceptable. Hard real-time constraints on system performance will exist and must be met.
- A hardware architecture was assumed consisting of a core network, multiple standard data processing (SDP) elements not necessarily of the same type, multiple buses, multiple multiplex data processing (MDP) elements, embedded sensor processing (SP) and embedded effector processing (EP). This is represented in Figure A-2. The interface plugs shown represent the unique

hardware interfaces which must be defined by standards and handled in processing. The triple dots represent continuation marks.

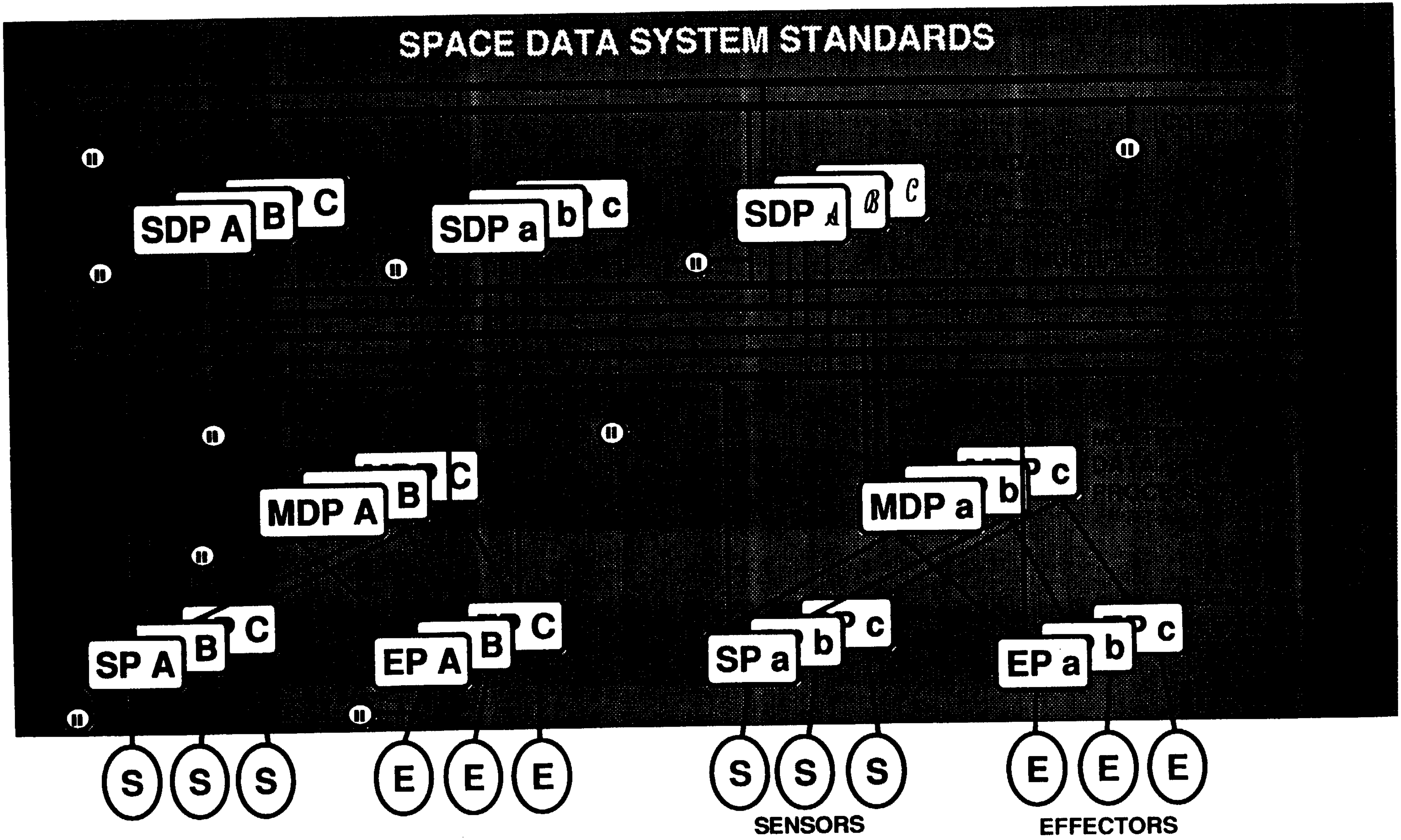


Figure A-2.- Hardware Architecture Assumed for the Space Generic Avionics.

A.3 CONVENTIONS

The initial definition and development of concept and architecture higher level requirements is done on a desktop machine with computer aided systems engineering (CASE) tools immediately at the analyst's fingertips to facilitate use of the tool at any time additional thought leads to new insights. For this analysis, the Excelerator CASE tool from Intersolv was used for this preliminary analysis. Later, revisions of the requirements were performed at a laboratory machine using the Teamwork CASE tool from CADRE Technologies. Each tool has its strengths and weaknesses (as discussed later) which were optimized by this approach.

The data which should be defined in a requirements analysis is stored in one of two ways. First, data can be stored in the CASE tool graphic screens using the entities and links between entities to represent required entities and required interfaces. Second, the specific requirement characteristics to be met can be stored in the tool's data repository attached to each specific entity and interface where the requirement is appropriate. The specific requirements which need to be stated generally address:

- Entity/Interface Purpose
- Associated Processes (for entities)
- External and Internal Environment (for entities)
- Qualities
- Timeliness
- Performance (Qualitative and Quantitative)
- Extension/Adaptation
- Attributes (Inputs and Outputs for entities)
- Source/Destination and Standards for Interfaces
- Requirements Source (Analysis, Document, Study, Trades, etc. with specific retrieval data)

Access to services should be through the appropriate application program interface (API), using standard service calls, not through direct access hardware interrupt or message activities unless provided for through standard service setup sequences (i.e., follow the rules). If performance requirements cannot be met through standard service calls, then direct access procedures may be required on an exception basis. The standard architecture must not preclude direct access alternative designs. Direct access alternatives should only be designed with explicit Project Management approval and enabled with explicit human control.

Software Reusability can include Algorithm Reuse, Top Level Design Reuse, Detailed Design Reuse, and Code Reuse. Where software reusability is an issue in architecture development and requirements definition, it must be defined explicitly.

Consistent and explicit definitions and partitioning of systems such as the Data Management System must be provided; they may be based on the generic definitions and results of the Space Generic Architecture. They should not be handled in a different manner from one mission or vehicle to the next. (Existing definition/scope of DMS has varied from one vehicle (i.e., Shuttle) to another vehicle (i.e., Station)).

APPENDIX B
CASE TOOL SUMMARY

APPENDIX B
CASE TOOL SUMMARY

B.1 TOOLS USED

The tools used in this analysis included Excelerator/RTS, Teamwork/SA, Excel spreadsheets and Coreldraw and Powerpoint.

Excelerator/RTS is used on the IBM 386 PC or compatible using color graphics for development clarity. Teamwork/SA is used on the Apollo 3500 in monochrome. Excel is used on the IBM 386 compatible for developing lists of processes, purposes, and data lists. Coreldraw and Powerpoint are used to develop graphic diagrams depicting miscellaneous logic charts, such as the operational processing threads.

B.2 OTHER TOOLS CONSIDERED

Other tools being considered for the next stage of development of this methodology include Statemate, Matrix-X and RDD-100. These tools will be used to implement dynamic modeling of the static architecture.

Statemate is TBD.

Matrix-X is TBD.

RDD-100 is TBD.

APPENDIX C
REQUIREMENTS DOCUMENTATION USED

APPENDIX C
REQUIREMENTS DOCUMENTATION USED

C.1 SHUTTLE

GNC Overview Handbook, GNC OV 2101, April 25, 1983.

Space Shuttle Avionics System, John F. Hanaway, Robert W. Moorehead, NASA SP-504, 1989.

C.2 STATION

Space Station Freedom Command and Control Data Flow Report, JSC-24673, April 17, 1991.

DMS Operating System/Ada Runtime Environment Software Requirements Specification, MDC H4189, 150A141A, August 22, 1990.

DMS Network Operating System Software Requirements Specification, MDC H4188, 150A191A, August 22, 1990.

DMS Standard Services Software Requirements Specification, MDC H4191, 150A241A, August 22, 1990.

DMS User Support Environment Software Requirements Specification, MDC H4192, 150A391A, August 22, 1990.

DMS Data Storage and Retrieval Software Requirements Specification, MDC H4187, 150A441A, August 22, 1990.

DMS System Management Software Requirements Specification, MDC H4190, 150A341A, August 22, 1990.

DMS Interface Requirements Specification, MDC H4193, 150A202A, August 22, 1990.

DMS MODB Manager Software Requirements Specification, MDC H4481, August 22, 1990.

Interface Requirements Document (DMS, SDP and MDM), MDC H4643, August 22, 1990.

Subsystem Level CEI Specification (Type B), Critical Item Development Specification for Network Interface Adapter (NIA), 152A404-PT1, October 27, 1989.

CEI Specification for Data Management System, Volume 1: Data Management System 212001A, WP-2, (DR-SY-06.1), SSFP SP-M_001.

C.3 OTHER

ACRV Technical Proposal, LMSC-F370228, November 16, 1989.

Space Avionics Requirements Study, NASA-37588-TD006, GD Space Systems Division, October 2, 1990.

OMV Design Review, No Date, No Document Number, TRV.

Advanced Launch Development Program Industry Conference Briefing, May 20, 1991.

APPENDIX D
NAMING AND DIAGRAMMING CONVENTIONS

APPENDIX D
NAMING AND DIAGRAMMING CONVENTIONS

When attempting to analyze a system of the size and complexity of the generic space avionics architecture it is easy to get lost in the numerous layers and processes represented by the data flow diagrams. For this reason, it becomes very important to establish naming conventions which will aid in the understanding of these diagrams.

D.1 PROCESS NAMING CONVENTIONS

The first rule when naming processes is that all processes must have unique names. When looking at the system as a whole, this would not necessarily be a requirement, because the process in question would be viewed in context with the rest of the system (e.g. process G is a part of process X and consists of processes A, B, and C). However, it turns out that making the names unique simplifies the procedures for coming up with unique data flow names. In addition, it also has the obvious benefit of removing any ambiguity in the process names when viewed out of context. It is also helpful when naming processes to use descriptors which have a common implied meaning whenever possible. Although the following descriptors are not required to be used in all process names, when they are used they imply the following meanings:

Manager - Interfaces with the crew to determine options and select an alternative to implement. (Direct interface with Crew D&C)

Controller - Process that selects and implements alternatives based on apriori or real-time criteria. (no *direct* interface with crew)

Handler - Process that implements selected procedures.

Some specific conventions are:

- In name concatenation, go from greater to the lesser class of names to facilitate searching by names.
- In naming processes, use acronyms if possible for id prefixes to relate the "buzzword" in common use to the process in which it falls. For example use "GNC" as the identification for the GN&C subsystem and "GNC 1" for the Guidance

function, "GNC 2" for the Navigation function, and "GNC 3" for the Control function (assuming these three functions comprise the GN&C subsystem).

- In laying out a diagram, show inputs from external entities (to the instant diagram) on the left and outputs on the right. Have processing move from left to right. Draw process entities first, arrange in the proper flow, then show the interfaces to external entities coming in and going out, check that all data in and out is accounted for and handled properly in a function, then connect up the processes with interior connections of entities.

- In Excelerator, develop the process entities, then the data flows, then describe the entities to ID them, then explode the entities to start a lower level process chart. The name of an entity in an explosion paths should be the same as the name of the parent entity from which it is being exploded, since the single entity being exploded and the lower level explosion diagram represent the same thing.

- Avoid using the same or almost the same name to represent different things unless intentional because the CASE tools use names to identify entities, and similar names will likely confuse the user, although the tool will not have any problems.

- Most tools are case-sensitive, so be careful with capitalization use.

- Show entity of interest surrounded by other entities with which it interacts.

- Treat entity of interest as a monolith to define its external environment.

- Naming should address entity of interest with a name (label) and an ID that can be appended to form lower level explosion entities; names should be brief to reduce typing burden.

- Use leading capital letters for entity names (labels) and all caps for IDs.

- First identify entities, then develop their 2 way data flows.

- Indicate data flow names by using the names of the entities at either end of the flow separated by a dash, or by one sides' entity.

- Describe data flows to consist of either (1) two or more one way data flows using underscores instead of dashes with the entities in the name in reverse order to indicate directionality subsequently partitioned by type of flow, or (2) two or more flow types subsequently partitioned by direction of flow and separated by underscores to indicate directionality.

- Use all capital letters for data naming

D.2 DATA FLOW NAMING CONVENTIONS

The naming of data flows has been the focus of much study and thought during the course of developing the structure of the generic data system. Although this may seem an insignificant, if not trivial task at the outset, it turns out to be quite challenging in practice. A logical and consistent naming convention for data flows, aids in the understanding of the diagram. The problem of naming data flows is twofold. First of all each data flow name must be unique. This is a requirement from a logical point of view as well as a hard requirement when using a CASE tool. Secondly, the name should convey as much information as possible about the processes it connects and its relative hierarchical position in the structure. It must do so, however, without being so long as to make the data flow diagram unreadable.

For the purpose of this discussion it is assumed that the reader is familiar with data flow diagrams. The following figure (figure D-1) presents three data flow diagrams in a hierarchical progression. In this case the data flows are labeled alphabetically to aide in the subsequent discussion.

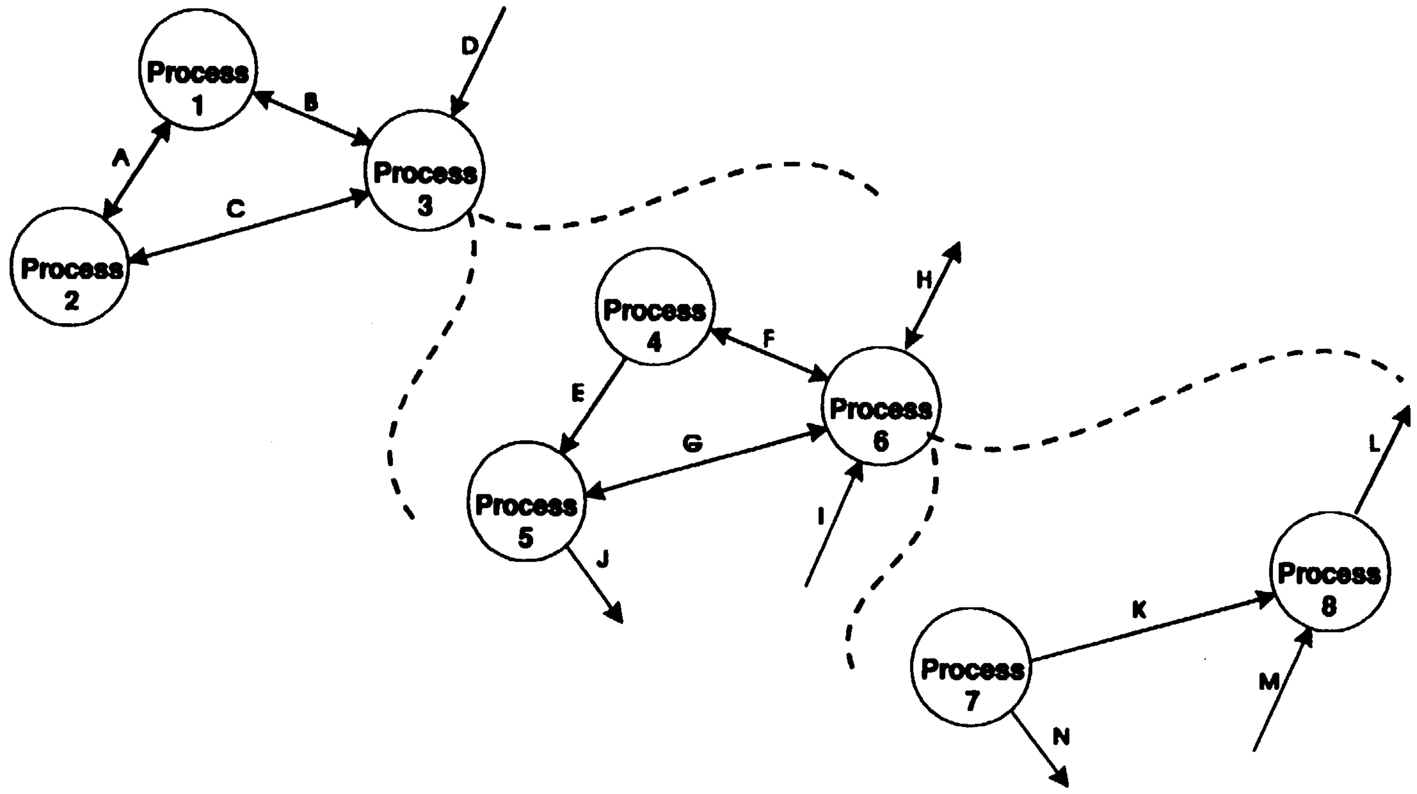


FIGURE D-1.- Data Flow Diagram Example.

In this example, process 3 "explodes" into the lower level diagram containing processes 4, 5, and 6. Data flows may also be "exploded" in a similar manner. for example, the data flow C might consist of data flows H and J in the lower level diagram.

A data flow connecting two processes on the same data flow diagram is an internal data flow. Examples of internal data flows would be C, E and K. A data flow for which only one end is connected on a given data flow diagram is an external data flow. Examples of external data flows would be D, J and M.

The following attributes for data flow names would be desirable:
Internal data flows should be named according to the subprocesses they connect.
A natural hierarchy of names would be helpful. i.e. when looking at the index of names they should fall alphabetically into the proper hierarchical structure.

External data flows should be traceable from level to level by the name alone. This permits tracing of the flows visually on the hardcopy without resorting to the data dictionary entries.

Ideally, the entire name of the connected processes could be used. This would eliminate any ambiguity caused by shortening the name. This is not, however, practical because of the length of the process names. Instead, a short version (3 to 4 characters) of the name is required. In general an acronym consisting of the first character of each word in the process name will suffice (because each process name is required to be unique). In some instances, however,. this will not guarantee a unique name. Take for example the data flows between processes named Vehicle Safety Manager and Vehicle System Manager. The acronym for each of these would be VSM which could lead to ambiguities in data flow names. For this reason, the shortened process names (acronyms) must themselves be unique. For the above example, one might use VSFM and VSYM for instance.

The following guidelines have been developed for naming data flows.

- A unique shortened name will be created for each process. Using figure D-1 as an example, the processes would all be given unique names of the order P1, P2, P3, etc.

- When these names are connected to form a data flow name, they are separated by a hyphen.

- Internal data flows will be named starting with the diagram process name followed by the connected process names. The internal data flow E would then be named P3-P4-P5 or P3-P5-P4. The order of the connected processes P4 or P5 is not important.

- External data flow names will start with the name they had in the previous level diagram, with the connected process name acronym as a suffix. For example, if data flow G was comprised of data flows N and M, and it was labeled P3-P5-P6, then data flow M would be labeled P3-P5-P6-P8, and data flow N would be labeled P3-P5-P6-P7.

Figure D-2 presents an example of this method applied to a set of data flow diagrams with representative process names.

These guidelines will produce unique data flow names between processes. In addition, when looking at different levels of the diagram it is possible to trace higher level data flows into constituent data flows. This method will also partially provide for a natural hierarchy of names alphabetically.

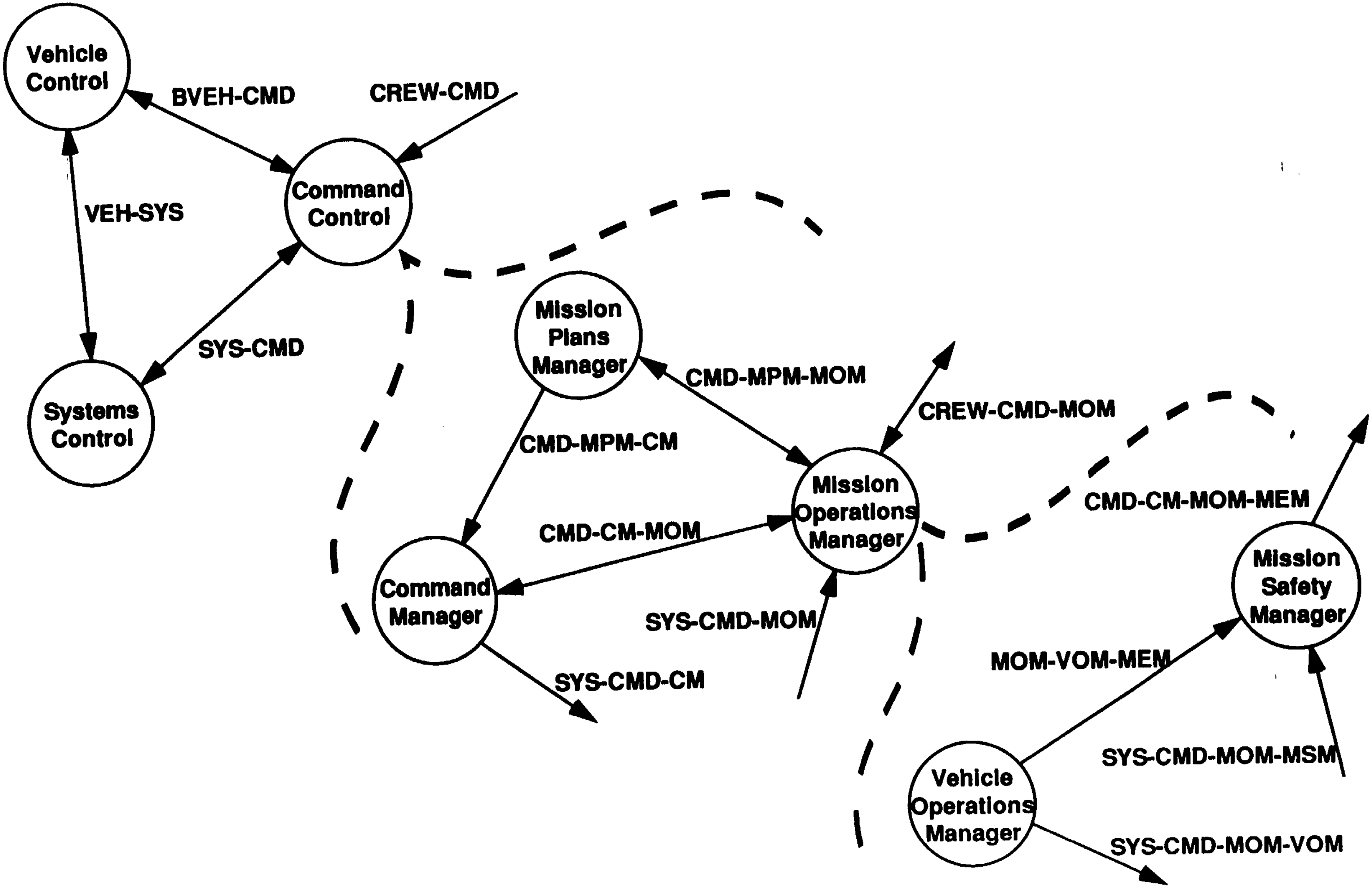


FIGURE D-1.- Data Flow Diagram Example.

Several alternative naming convention methods were tried, one of which is described below.

> One method that was tried was to begin the data flow name with the acronym of the process diagram on which they appeared. For internal process names, this is the same as was presented above. For external process names, however, this represents a significant change from how they are currently defined. The problem with this method is that external data flow names lose their traceability to the parent diagram. (of course the CASE tool can still trace the names, but visually the traceability is lost on the hardcopy). In addition, when looking at an index listing of the data flow names, the natural hierarchy is lost.

CHANGE 1 - JANUARY 1992

DISTRIBUTION LIST FOR LESC-29702 REQUIREMENTS ANALYSIS NOTEBOOK FOR THE FLIGHT DATA SYSTEMS DEFINITION IN THE REAL-TIME SYSTEMS ENGINEERING LABORATORY (RSEL)

NASA

EK111/D. M. PRUETT (5)
EK231/D. A. STEPHENSON
PT41/E. M. FRIDGE
EG1/D. P. BROWN

EK711/R. E. COBLENTZ
EK121/D. A. DYER
AMES/E. S. CHEVERS
EG111/K. J. COX

MITRE

S. BELL
MITRE CORP.
1120 NASA ROAD 1
HOUSTON, TEXAS 77058

MCC

COLIN POTTS
MICROELECTRONICS AND COMPUTER
TECHNOLOGY CORPORATION
3500 W. BALOONES CENTER DRIVE
AUSTIN, TEXAS 78759

UHCL

CHARLES HARDWICK
UNIVERSITY OF HOUSTON -
CLEAR LAKE
2700 AY AREA BLVD. - BOX 444
HOUSTON, TEXAS 77058

LESC

C18/J. R. THRASHER
C18/E. A. STREET
C18/R. E. SCHINDELER
C18/G. Y. ROSET
B26/J. P. SASSARD
C18/J. STOVALL
C106/P. G. O'NEIL

C18/G. L. CLOUETTE
C18/R. W. WRAY (10)
C18/M. W. WALRATH
C18/B. L. DOECKEL
B08/R. N. LUTOWSKI
B11/H. E. SMITH
C83/S. J.THOMAS

C18/JEAN FOWLER (MASTER + 2 COPIES)
B15/LESC LIBRARY (2)