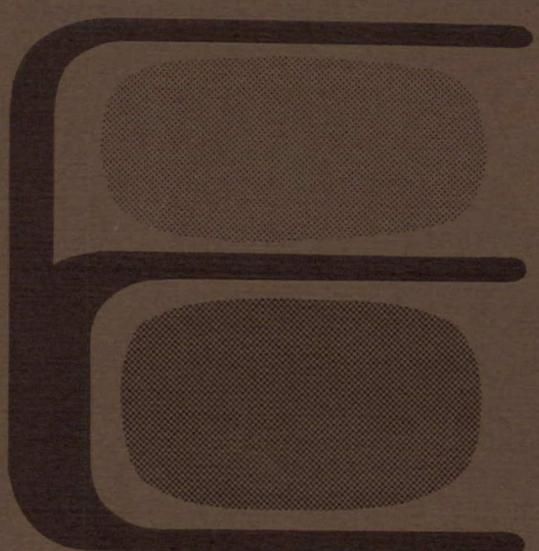
Eidetic Systems Corporation

Title Functional specification for the advanced autonomous spacecraft computer / I. Cunningham... [et. al.].



P
91
C655
G646
1983

check Queen
P
91
C655
G646
1983

(2)

FUNCTIONAL SPECIFICATION FOR THE ADVANCED AUTONOMOUS SPACECRAFT COMPUTER

Technical Report No.ESC-82-004

Industry Canada
Library Queen
JUIL 20 1998
Industrie Canada
Bibliothèque Queen

(1)

I.Cunningham
T. Gomi
M. Inwood
I. McMaster

Eidetic Systems Corporation

January 17, 1983

COMMUNICATIONS CANADA
JUIN JUNE 21 1984
LIBRARY – BIBLIOTHÈQUE

DD 4593194
DL 4593227

P
91
C655
G646
1983

Government of Canada | Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT DOC-CR-SP -83-032

DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: Functional Specification For The Advanced Autonomous Spacecraft Computer

AUTHOR(S): T. Gomi (Applied AI Systems Inc., Kanata, Ont.)
I. Cunningham
M. Inwood
I. McMaster

ISSUED BY CONTRACTOR AS REPORT NO: ESC-82-004

PREPARED BY: Eidetic Systems Corp.
P.O. Box 13340
Kanata, Ontario

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: 15ST.36001-2-0561
SERIAL No. OST82-00056

DOC SCIENTIFIC AUTHORITY: R.A. Millar

CLASSIFICATION: Unclassified

This report presents the views of the author(s). Publication of this report does not constitute DOC approval of the reports findings or conclusions. This report is available outside the department by special arrangement.

DATE: January 1983

C O N T E N T S

ACRONYMS

AASC	Advanced Autonomous Spacecraft Computer (AASC)
Ada	DoD defined Ada programming language
AI	Artificial Intelligence
APSE	Ada Programming Support Environment
ASM	Autonomous Spacecraft Maintenance
CSMA/CD	Carrier-sense Multiple-Access with Collision-Detection
FTC	Fault-Tolerant Computing
IEEE	Institute of Electrical and Electronic Engineers
IIU	Subsystem I/O Interface Unit (AASC/IIU)
ISO	International Standardization Organization
LAN	Local Area Network
MBPS	Mega Bits Per Second
MIPS	Mega Instructions Per Second
MTFF	Mean Time to First Failure
NASA	National Aeronautics and Space Administration
OAM	On-Board Autonomy Manager (AASC/OAM)
OSI	Open Systems Interconnection
PCU	Subsystem Processor Complex Unit (AASC/PCU)
VLSI	Very Large Scale Integration

ACKNOWLEDGEMENTS

This report is one of a series resulting from studies performed for the Federal Department of Communications, Communications Research Centre, Shirley Bay, Ottawa, Ontario, Canada under DSS Contract OST82-00056.

The authors would like to acknowledge the support of Dr. R.A. Rennels, of the University of California and also the Jet Propulsion Laboratory of the California Institute of Technology, and the assistance of Dr. L. Friedman, also of JPL; Dr. Tom Anderson, University of Newcastle-upon-Tyne, England; Dr. George A. Gilley of The Aerospace Corporation, El Segundo, California; Dr. Paul J. Heckman, Jr. of Naval Ocean Systems Center, San Diego, California; Mr. Lee J. Holcomb, Office of Aeronautics and Space Technology, National Space and Aeronautics Administration; and Dr. Nancy Leveson of University of California, Irvine. The authors also acknowledge the support of R.A. Millar, of the Communications Research Center, under whose guidance these studies are conducted.

SUMMARY

The following aspects of the Advanced Autonomous Spacecraft Computer (AASC) are specified: the functions of the on-board computer system in relation to the autonomy of the spacecraft; the constraints under which the development will take place; and methods of verifying compliance of the developed AASC to the specificiations.

Definitions and values are given for reliability terminology and parameters. Software fault-avoidance is examined and three approaches selected: N-version programming, recovery blocks, and ad hoc measures. Fault diagnosability is discussed. The objectives for system flexibility are outlined and performance criteria are established. The means of achieving on-board autonomy are introduced.

Design constraints, including the Fault Tolerant Computing rules, the application of an overall top-down developmental approach, software fault-avoidance methods and networking standards, are imposed to allow the maximum ability to adopt new fault-tolerant techniques and methods, as and when they are appropriate.

Compliance with specifications is to be tested throughout the system development by the appropriate application of testing techniques to be indicated in a test plan.

A detailed design plan is to be the subject of follow-on work.

1. Introduction

The Advanced Autonomous Spacecraft Computer (AASC) is a conceptual design for an onboard spacecraft computer. It was introduced as a proposed "ideal spacecraft system" during an examination of building block computer concepts [GOMI 82a] for the purposes of comparison with existing designs. It was, originally, an attempt to bring together the results of many developments which had occurred since the design of such building-block computers as the Jet Propulsion Laboratory's Unified Data System/Fault-Tolerant Building-Block Computer and the European Space Agency's On-Board Data Handling System. The concepts were elaborated in a further study [GOMI 82b], during the course of which it became apparent that the ideal of autonomous spacecraft management was the aim of a number of influential authorities in the spacecraft community. So certain were these authorities that autonomy was possible that they indicated a time frame for its achievement [MARS 80]. During this period of study, technological advances have been made which form a sound basis for achieving the fault-tolerance necessary for such an undertaking. These developments cover the disciplines of system engineering, artificial intelligence, and networking, as well as computer software and hardware. They have produced such innovations as the International Standards Organization's Open Systems Interconnection (ISO/OSI) protocols, software fault-tolerance, Ada programming language, highly configurable operating systems, and new computer architecture such as iMAX 432. They were examined in the report which precedes this and their relevance to the goal of autonomy was established.

It was hinted in the ASM Final Report [MARS 80] that the achievement of autonomy involved more than automated fault-tolerance. A high degree of decision making would be necessary, adding a new discipline to those already involved, namely Artificial Intelligence (AI). Over the past year, AI has had an increasing amount of publicity and has acquired some respectability, with much attention being focussed on robots of varied intelligence for various non-trivial applications, and practical Expert Systems such as MYCIN and PROSPECTOR. Research during the course of the present studies has clarified the part to be played by such systems in the achievement of autonomy. Work on expert systems in connection with autonomy is proceeding at NASA ("Babysitter"), Naval Ocean Systems Center ("EAVE WEST")[HECK 81] and JPL ("DEVISOR")[VERE 81a,b]. This discipline and its variations are to be an integral part of the

design to achieve autonomy in the AASC.

From the start of these studies, the approach has been the top-down/structured approach advocated very early in the 1960s by IBM and brought into the public domain by Yourdon [YOUR 75] in connection with software design and programming. This methodology has proved to be appropriate and valuable in many fields where the development of systems by professionals is involved. Its application has been the guiding principle throughout these studies and it has been applied to all areas of study in this particular phase. The top-down approach to design requires that first principles be dealt with before system details, as implied by the system structure shown in Figure 1. For this reason, it is not the intent, in this study, to differentiate between hardware and software in outlining the functionality of the AASC. Those detailed aspects will be covered in the later stages. This report does cover the results of a sifting process applied to the mass of information, papers and consultations made in connection with the latest research and activities in the Fault-Tolerant Computing (FTC) world. The results of this process have been applied in a specific manner to those somewhat abstract system attributes required for the achievement of autonomy. Thus the report defines in concrete terms reliability, flexibility and the application of autonomy.

Improvements in fault-tolerant techniques are occurring continually. It is intended that the design of the AASC should permit advantage to be taken of these revisions and introductions and its design should, therefore, be undertaken in a disciplined fashion. The constraints of top-down/structured development, Fault-Tolerant Computing rules, which were developed earlier in the study as guidelines for maintaining a high degree of fault-tolerance, and networking standards, have been imposed with the intention of making the design capable of accepting future changes. It is also felt that these constraints, together with software fault-avoidance techniques, will ensure the high quality necessary in a system before fault-tolerant techniques can be applied.

Finally, an examination of the latest thinking on program validation, verification and testing has been made. This, too, takes the top-down approach, with the recommendation that these techniques should be conducted in a planned fashion throughout system development. The test plan should be drawn up in conjunction with the design plan, to ensure

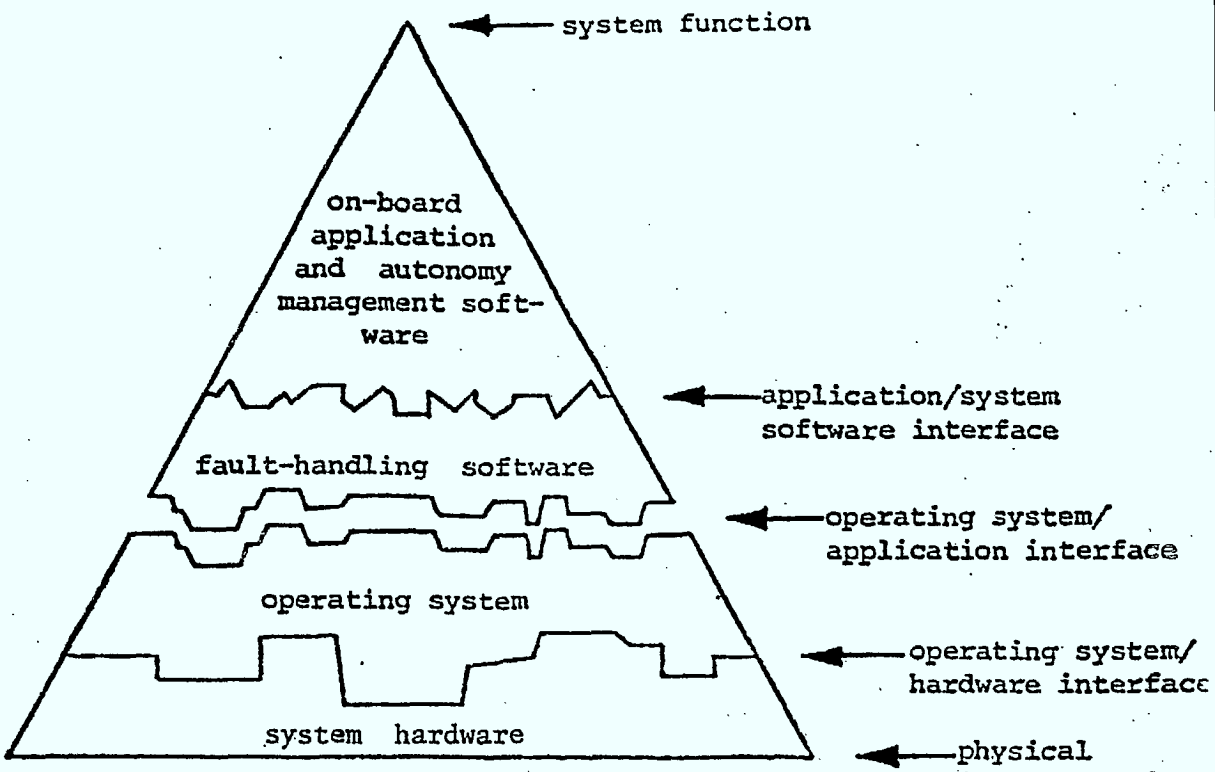


Figure 1. System Structure

system testability and completeness of testing.

2. Functional Objectives of the AASC

2.1 Reliability of the AASC

2.1.1 Definition and Assumptions

As any system is subject to failure, it is desirable to predict and to measure the degree to which the system performs its functions. However, there is a wide variance among the meanings attached by designers and researchers to the terms used to describe reliability parameters. For this reason, we define the following terms used in specifying the reliability of the AASC.

Valid State:
: the state of a system from which transitions produce the desired output.

Invalid State:
: a state from which transitions produce undesired output.

Failure:
: transition of a system from a valid state to an invalid state. For some period of time following the failure, outputs will be incorrect.

Fault:
: the internal condition of a system (component, program, etc.) which causes it to fail.

Transient Fault:
: a fault that exists for an interval T such that

$$4 <= O(T/M) <= 9$$

where M is the length of a mission.

Reliability:
: a function R(t), the probability that no failure has occurred to time t.

Failure Rate:
: A constant lambda.
Based on this assumption the reliability function is:

$$R(t) = Ro \exp(-lambda * t)$$

where $0 <= Ro <= 1$, the probability that the component is functioning at $t = 0$.

Mean Time To First Failure (MTFF):

the time T such that

$$Pr\{\text{first failure is in } (0,T)\} = Pr\{\text{first failure is in } (T,\text{infinity})\}$$

For exponential reliability,

$$MTFF = 1/(\ln (2 * lambda))$$

where ln = natural logarithm

Availability:

the proportion of time during which a system produces the desired output.

Error:

the occurrence of undesired symbolic output from a process. An error can be due to a hardware failure or a software fault.

Forward Recovery Procedure:

an algorithm which produces a transition from an invalid system state to a valid state that is not one of the states from which the invalid state was derived.

Backward Recovery Procedure:

an algorithm to produce a transition from an invalid system state to a state from which the invalid state was derived.

2.1.2 Reliability partitioning.

AASC reliability can be partitioned as shown in Figure 2. Whenever there is a choice of embedding a fault-tolerance procedure in hardware or software, hardware will be chosen. However, detection, isolation, diagnosis, and recovery at the subsystem level and at the network level will be accomplished by software.

The first line of defense in software is fault-avoidance, that is the minimization of the number of faults in software before mission implementation begins. Software fault-avoidance is discussed in Section 3.3.

The following techniques will be used to tolerate faults in software.

1. N-version programming: this technique has been described and implemented [LEVE 82, ADRI 82] at the system level. That is, two or more redundant, independently designed and implemented software systems run concurrently, voting on their outputs. Fault-detection occurs when the outputs disagree. In a multiprocessing and/or multi-tasking environment, redundancy can be extended to the process level. That is, redundant independently designed and implemented procedures can execute concurrently to perform the same function. Outputs of the multiple processes can be compared to detect failures, and can diagnose the failure if redundancy is triple or greater.

   The advantage of procedure-level redundancy is that it can be applied selectively to critical functions, reducing the cost of redundancy drastically.

2. Recovery blocks: this technique will also be extended beyond previous implementations. Existing implementations impose on each critical procedure the structure:

   - acceptance criterion;
   - list of alternate equivalent procedures;

   together with the implied control sequence:

```
i := 1;
loop
     alternate procedure i;
     i := i + 1;
     exit when acceptance criterion is met;
```

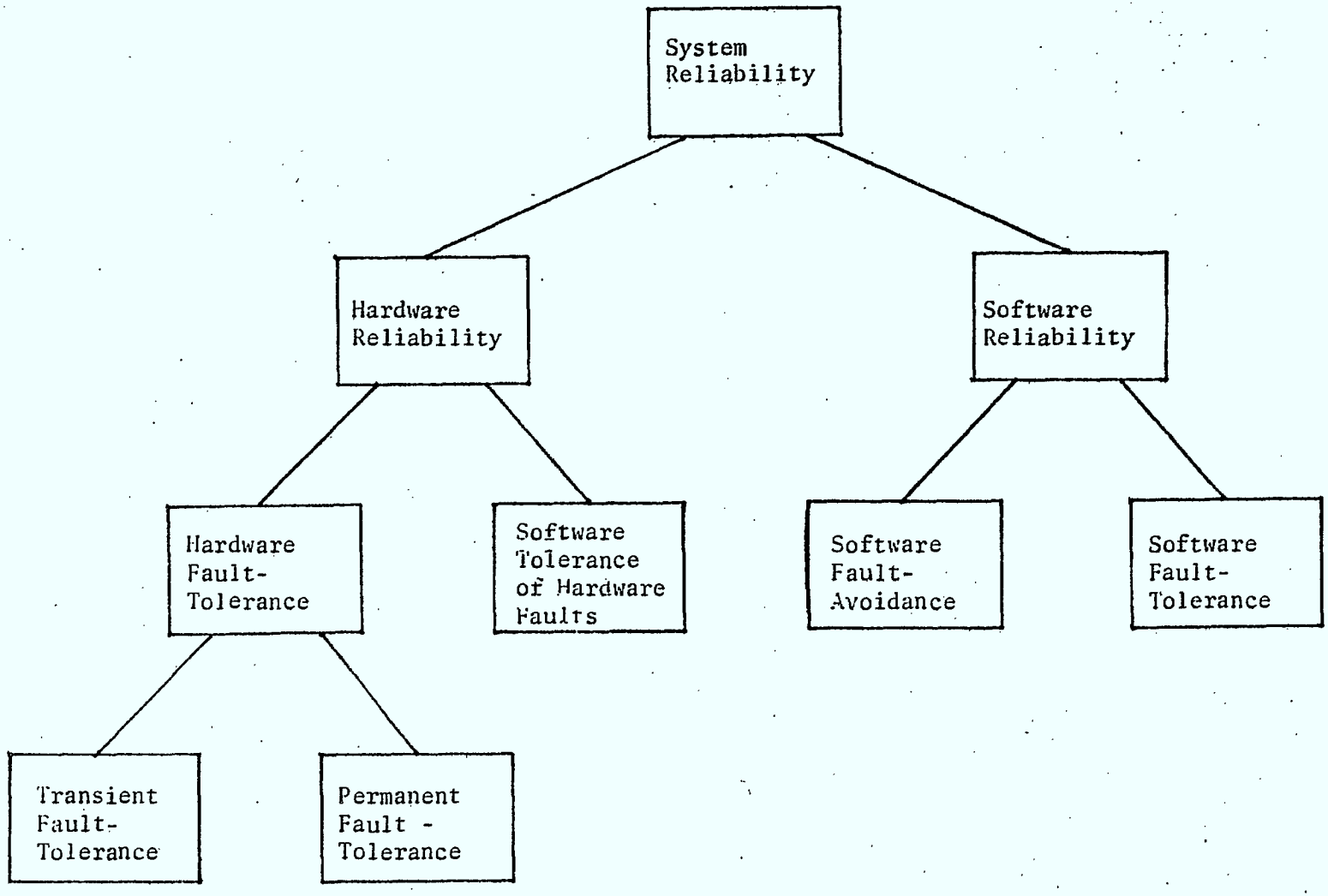


Figure 2. Reliability Partitioning

end loop;

The main criticism of this technique is its rigidity. In the AASC, the recovery block will be extended to include:

- a flexible, algorithmic control sequence for the alternate procedures;
- a final alternate procedure to be executed if no alternative meets the acceptance criterion;
- the ability of an alternative procedure to use the results of previous alternatives rather than return to the initial state;

3. Ad hoc techniques: there may be situations where neither of the above methods can be used, because of timing or cost constraints or irreversible error conditions. In these cases, ad hoc techniques for forward recovery will be used.

2.1.3 Reliability values

The parameters of importance to spacecraft missions are:

Mean Time to First Failure
System Failure Rate
Availability
Error Rate

MTFF is an estimate of the upper bound on mission length, while system failure rate is an estimate of confidence in a system's ability to complete a mission. Availability and error rate deal not with failures of the system but with the degree to which a system performs its functions over a mission.

A system may perform in a degraded mode for part of a mission or may, for short intervals, not perform any functions, even though this loss of function does not end the mission. Availability is a measure of this kind of reduction in function.

A system may produce information which is partially incorrect. If the errors do not cause disturbance to the he-

alth of the spacecraft, the mission can continue even though it does not perform functions perfectly. The error rate parameter is a measure of the extent to which information is corrupted.

The MTFF of the AASC will be 10**9 hours. The system failure rate will thus be approximately 10**-9 failures per hour. If the maximum expected mission length is 30 years, this implies that on average, 1 in 4000 missions will experience a system failure.

The error rate will be 1 incorrect symbol in 10**9.

The availability of the AASC will be (1 - 10**-9).

2.2 The Flexibility of the AASC

The AASC design will possess a high degree of adaptability at all levels of its structure, both in hardware and software. To this end, the following design objectives are found to be essential:

(1) mission-to-mission application changes: the ability to serve an arbitrary set of applications on each mission.

The ability to apply the same basic design of the AASC to several missions will result in an immediate reduction in development costs, and longer term costs including costs for operations, testing, upgrading, training, and maintenance. It is expected that the basic design will have a clean modular structure to allow substitution of components at all levels of its logical organization. A small application will have simpler networking and a smaller number of relatively simple clusters. The on-board system software for such a system will be equally bare-boned in a systematic fashion. A larger AASC implementation will have more capabilities implemented both in software and hardware, while sharing the same basic structure with the simpler version. There will be many sizes and complexities of the AASC between the two extremes. Furthermore, a mission may contain more than one AASC, each of different magnitude and characteristics.

Two examples of dissimilar missions might be a geo-

synchronous communication satellite and an eccentric orbit research probe. Applications in the former might include antenna deployment, orbit and attitude maintenance, communication channel control, power supply monitoring and control, while the latter might require multiple sensing device deployment, navigation, trajectory control, and telemetry.

(2) in-mission application changes: the ability to adapt in flight to changes in mission profile.

The diversification of space activities expected in the near future will lead to multiple mission objectives charged to many future spacecraft. This calls for hardware and software structures that are adaptable to in-flight, on-board reorganization.

The mission profile changes of the Voyager spacecrafts are typical examples. After successfully completing their activities around Jupiter, they used its gravity field to bring the orbit towards Saturn. The eventual successful completion of the Saturn mission led to the in-flight implementation of yet other missions to explore the outer-most planets of the solar system.

Interpreting this adaptability with respect to earth-orbiting satellite missions, a geosynchronous communication satellite would, for example, enjoy the ability to reassign on-board switching facilities from TV channels to data communication channels from time to time during its life.

(3) the ability to upgrade or enhance system components and relationships.

It must be possible, between missions, to replace one hardware component with another that is functionally similar, but has different performance levels, technology base, or other characteristic which is not related to logical function. Such changes will be necessary as a result of, among other things, changes in performance requirements, advances in hardware technology (for instance radiation hardness), cost, and availability.

The relationships among system components must be flexible so that, as experience accumulates, design

changes can be made without major changes in unrelated areas of the design. For example, if design changes require that Processor Complex Units (PCUs) be dynamically assigned to I/O Interface Units (IIUs)[GOMI 81b], instead of dedicated to them, this will be possible by changing the functional characteristics of a very small number of system modules.

Functions implemented in software are inherently more flexible than those in hardware. It will be possible to substitute for one software module another module which implements an enhanced algorithm for a particular function, without adversely affecting cost or effectiveness of the AASC and, hence, with little or no effect on other modules in the AASC. For example, if a new method for fault recovery is discovered, its introduction into the AASC should cause no side-effects.

(4) in-mission system changes and distribution of loads: modification of configuration to cover component failures during a mission and the ability to cope with non-homogenous load distribution.

While system level re-structuring may be achieved far more easily on software structures, hardware must also provide the facility to remove incapacitated components from the system and re-load the task to a spare. Hence, a mechanism to feed spares and disconnect invalidated elements is essential. Although present technology does not permit complete physical removal of rejected hardware components, there must be a scheme to keep them totally out of the way. Replacements, in most cases, must exist in the form of blank spares (as opposed to hot spares) so that the overhead is kept low. For the same reason, they must exist in a sufficiently fine granularity, preferably in several classes. For a given analyzed fault, spares of a finer granularity will be applied first (e.g. VLSI components are replaced before computer cards, cards before complete processor units, and so on).

The on-board software will have the ability to revise its execution environment. The revision may occur gradually by updating portions of the environment at a time; drastically, by rewriting the entire run-time world in a short sequence of events; or at any tempo

in between the two extremes. The system software must not only be able to modify itself but also be capable of managing application loads in the reconfigured physical and logical execution environment.

The interconnecting on-board networks must also be reconfigurable to cope with on-board emergencies. The ways in which they can be configured will include topology, capacity, protocol and the degree of redundancy.

A wide variation in the demands by on-board subsystems on resources such as computing or communication bandwidth will always exist, even during the course of one mission. For example, an on-board image analysis subsystem would require a greater processing capability than a subsystem that handles input from a strain gage which measures the external distortion of the spacecraft shell at discrete intervals.

2.3 Architecture of Fault-Tolerance

A distributed processing system with distributed fault-tolerance capabilities, such as the AASC, must include the following:

1. distributed fault detection - the ability of all nodes of the distributed system to detect faults in other nodes to which it is linked.

2. cooperative fault diagnosis - the ability of more than one node to exchange fault detection information in order to make a correct diagnosis.

3. node/link fault discrimination - the ability to distinguish between node faults and faulty communication links.

4. message/state conflict detection - the ability to determine that a correct message is being sent at an inappropriate time.

- physical isolation - the ability to physically prevent signals from a faulty node from reaching other nodes when such signals could prevent operation of the node (for instance because of the high rate of arrival of signals).

- logical isolation - the ability to prevent logical information from a faulty node from reaching good nodes.

- backward recovery - the ability to restore a faulty node to a previous correct state and to restore all nodes whose current states depend on the faulty node [WOOD 81].

- forward recovery - the ability to find an arbitrary correct state for a faulty node and transform that node into the correct state and to perform similar transformations on nodes dependent on the faulty node.

- physical reconfiguration - the ability to change physical links and node characteristics where necessary to allow forward recovery

- logical reconfiguration - the ability to change logical relationships among objects in the network, for instance, reassignment of processes to processors, to allow forward recovery.

Algorithms exist for all the above capabilities, given the above assumptions for fault occurrence.

2.4 Performance

The performance criteria of the AASC are described in terms of the following system parameters:

(1) the minimum throughput of a network linkage between two clusters.

For the transmission of data that is not a representation of an image or voice, 10 MBPS (Mega Bits Per Second) will be required as the minimum bandwidth between two clusters. In places where image or voice processing is involved, this may not be sufficient and an appropriate bandwidth must be provided. When the continuity of transmission is important, bandwidth alone is not sufficient to define a satisfactory operational environment (e.g., voice transmission or similar traffic on a 10MHz CSMA/CD network will have to face "glitch" problems). The combination of network access delays, routing delays

and bandwidth of the communication media must create a satisfactory communication channel to meet the access requirements set by the application or combination of applications.

(2) The logic and arithmetic operations required by an application (excluding extremely high-intensity computation such as high-resolution time series analysis and image processing) will be done by a processor complex unit. The through-put of this unit will be sufficient for all appropriate applications. In general, the unit must be able to yield a minimum of 1.0 MIPS (Million Instructions Per Second) of processing power when properly configured and measured in terms of a 16-bit instruction set or equivalent.

(3) For floating-point computation, the processor unit must have at least 0.25 MIPS of throughput when executing a mix of basic floating point operations (add, subtract, multiply, divide and modulo) on floating point values in IEEE standard single precision (32-bit) format.

(4) The memory transfer rate between the processor in the cluster and on-unit memory array must be more than 5.0 megabytes per second. Similarly, the data transfer rate between the unit and off-unit memory (secondary memory) must exceed 2.0 megabytes per second.

(5) The operating system for the processor unit will provide a multitasking environment which is transparent to any multiprocessing scheme the processor unit may have. It will support basic multitasking functions (e.g. inter-task communication and control, context switching) with less than 25% overhead. The maximum time requirement for context switching must be less than 50 microseconds.

2.5 Autonomy Requirements

The On-board Autonomy Manager (OAM) will maintain the well-being of the on-board operation of the spacecraft. This will include proper handling of minor faults. In the case of severe on-board faults, it will attempt to solve the crisis in cooperation with ground control.

In order to satisfy ground control's audit, override and reporting requirements, the OAM will establish communication with the ground asynchronously - i.e. no fixed time windows, or synchronization restrictions - when one of the following conditions exists:

(1) ground control wishes to query or monitor any aspects of on-board operation

(2) ground control attempts to take over part or all of the on-board management

(3) the OAM decides that a significant on-board event requires reporting

The thresholds implied in (3) above will be determined case by case, as system parameters, for each application. The OAM will maintain sufficient on-board archiving storage so that information subject to ground audit may be kept for the period of time determined for each mission. Such storage shall be organized in a hierarchical fashion so that on-board archiving of data will occur in diminishing frequency and quantity in respect to time (e.g. the on-board storage will retain detailed data on the events of the previous few hours, a summary of the events for the previous week and a condensed summary of events that occurred several months ago). The anticipated relationship between the OAM, the on-board system and ground control is indicated in Figure 3.

Since the technology involved in developing the OAM is new, several aspects of its design may evolve rapidly during the early years of its implementation. For this reason, the initial design of the OAM must consider the following:

- The OAM must be able to, at a later time, incorporate an explanation subsystem that will describe how the on-board expert system achieved its deduction, or other forms of decision making, from a given set of conditions.

- In the event that suitable techniques are developed in the field of so-called "knowledge acquisition" or "learning", the OAM will be able to gracefully include them in its structure.

- The on-board expert system will maintain a reasonable architectural distinction between the knowledge-base

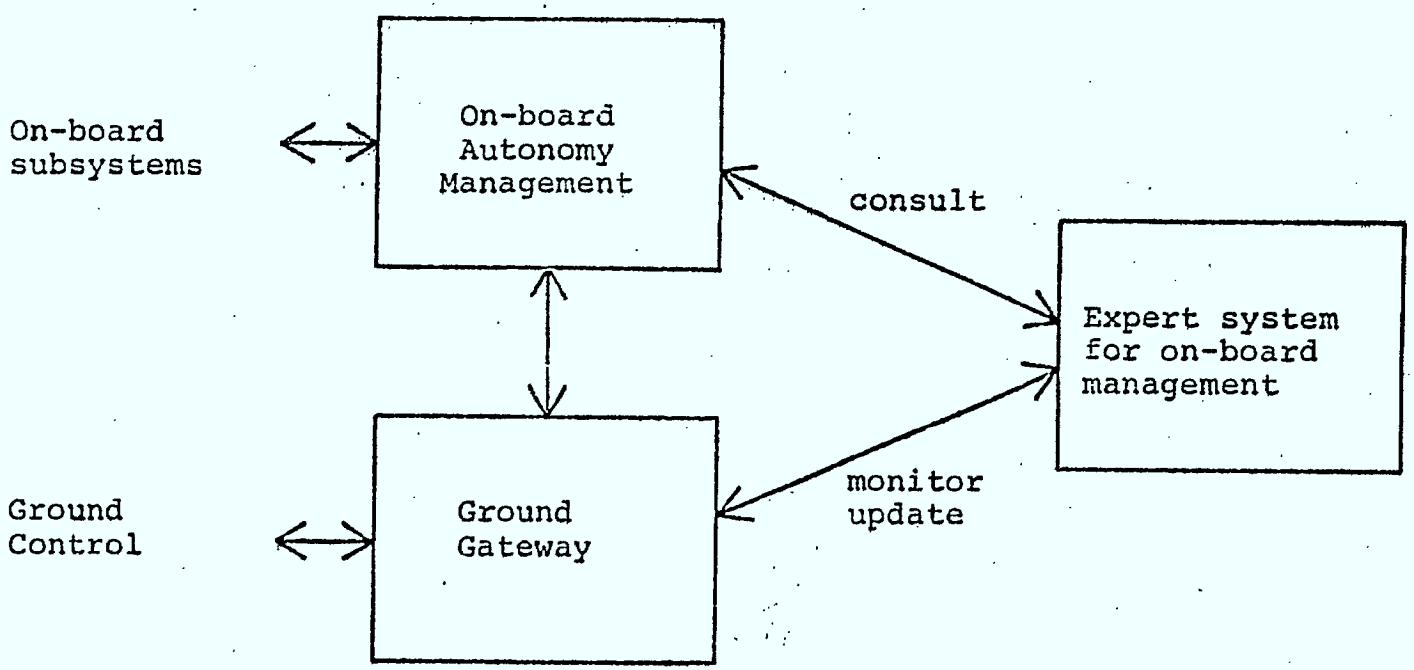


Figure 3. On-Board Autonomy Management

and the inference mechanism. First, this is to permit different applications to use the same OAM structure by substituting a knowledge-base with one that is appropriate for the new application domain. Second, such a modular structure will allow the system designer to take advantage of improvements in the implementation of these elements.

- The "global data base" or "scratch-pad" used during the operation of the OAM will have an appropriate internal structure and capacity to store not only current data being used in the process of deduction but also data concerning the immediate past operation of the OAM, such that backtracking can be performed when necessary.

- The OAM will contain hooks to incorporate a mechanism to deal with "fuzzy situations". Heuristic techniques will be introduced when the OAM is forced to process fuzzy conditions or apply unclear deduction schemes. When these heuristics take place the OAM will report their occurrence and the procedures followed to ground control.

3. Design Constraints of the AASC

The following constraints are applicable to the design of the AASC.

3.1 The Fault-Tolerant Computing Rules (FTC Rules)

Reliability of a system can be increased in two time domains: development time and mission time. The use of techniques during system development that tend to decrease the number of faults in the system at the start of a mission is called fault-avoidance. In particular, during the design phases of system development, there are a number of constraints which will reduce the number of faults in a system, as well as ensuring that the flexibility criteria (section 2.2) are met.

The following are the proposed fault-tolerant computing design rules (FTC design rules):

[1] There shall be no, or as few as possible, single points of failure in the system (the redundancy rule).

A chain is only as strong as its weakest link. If the system is dependent on a single item, channel, or methodology, system reliability equals the reliability of such single points of failure. For instance, if a star configuration is used in a multiprocessor system, failure of the central node causes inter-processor communication to fail. If there is only one software module available for executing a crucial algorithm, a fault in that module means the algorithm cannot be relied upon.

[2] There shall be no fixed master-slave relationships among processing units (the democracy rule).

The use of dedicated redundancy means an inefficient use of resources and loss of flexibility. In a fixed master/slave processor relationship a faulty master can propagate errors throughout the system before the damage is discovered.

[3] There shall be no permanent fault arbiters or judges in the system (the modesty rule).

A permanent fault arbiter runs the risk of harbouring faults. If the ability to detect or diagnose faults is incorporated in individual modules, the chance of caus-

ing widespread failures due to a faulty judge can be avoided.

[4] Interconnection among functional modules must be minimal and of the simplest type (the decoupling rule).

Whenever a function is supported by processors, processes, tasks, subprograms, or other form of sub-functional modules, the method of inter-connecting them shall obey the module decoupling rules proposed by Glenford Myers [MYER 75,78].

In a dynamically configurable system, which permits its own reconfiguration in operation, the ability to manipulate component modules is important. In order to achieve the addition or removal of a module with the minimum of disruption, modular interfaces should be as simple and clean as possible. While the principle is clear in hardware terms, its application to software often requires discipline on the part of the designer. The simplest example is the minimization of the number of arguments passed between calling and called programs. Similarly, variables in a block-structured language should be declared as locally as possible, because a global declaration gives all procedures access to the variable, thus providing the possibility of side-effects and coupling procedures unnecessarily.

The concept, along with the concept of module strength given below, was developed in software engineering. However, it is judged that these concepts are equally applicable to non-software system entities such as hardware components and processes.

[5] Every functional module must follow Myers' module strength rules (the module strength rule).

The strength of a module lies in its raison d'etre. A functionally cohesive module will be easier to recognize and manipulate, and will not disintegrate in a dynamic environment. A module must perform a recognizable, coherent function. A module which attempts to execute two or more disparate functions will require a more complicated interface with other modules, breaking Rule [4]. Furthermore, since there is more than one function, the probability is increased that a modification will be required of that module, so system testing and modification will become harder and more error

prone.

[6] System function must be broken down vertically into layers of decreasing levels of abstraction. Decoupling between the layers must be observed (the layer rule).

Since hierarchical thinking is natural to humans, complex structures arranged in orderly layers are more readily understood. Such a structured representation of concepts will be more accurate, revisions will be fewer and more readily implemented. Examples of this thinking are seen in the designs of operating systems, computer networking, and computer graphics.

3.2 Top-Down/Structured Development

In order to adhere to the FTC rules described above, system development will use techniques collectively known as "top-down/structured development". The methodology applies a consistent approach to phases of development commonly known as system design, module design, implementation, and testing. "Module" here implies software, hardware, or a combined software/hardware entity that may exist in a system statically (e.g. processors) or dynamically (e.g. processes). A module performs a predefined function and contributes to the global function of the system. The common characteristic at each step is that the individual functions (or modules, or test units) are defined by a process of successive refinement, starting with the most global functions (or modules, or test units).

Since there are several versions of the interpretation of this technique, the following sections describe briefly the application of top-down/structured techniques to each development activity. While four activities are described, it cannot be emphasized too heavily that these activities are not consecutive in time, but are overlapping and largely concurrently executed in much smaller units. That is, once the decomposition of system functions has passed the second level, implementation of the first level function and its interfaces can begin. Once the first level is implemented, testing can begin. The extent with which implementation of a level and the design of the next overlap may vary depending on the nature of the implementation and other circumstances. The point is that the implementation, integration and testing of a level will not have to await the completion of the next lower level or any subsequent levels. Feedback from the

level immediately below the one being tested is encouraged in order to remove distortions at the earliest possible chance, while long-hop feedbacks are discouraged to maintain an effective development cost.

3.2.1 Top-down system design

The first step is to define the overall function of the system, characterized as accurately and as succinctly as possible. This document, for example, is aimed at achieving this objective. This function is then decomposed into a small number of subfunctions, each of which is required, at some point in time, to accomplish the global function. Each of these subfunctions can, in turn, be decomposed into a simple set of "lower-level" subfunctions, and so on, until a set of primitive, (or "low-level" or "terminal") subfunctions is arrived at, whose simplicity does not require further decomposition.

The above process will produce a set of functions whose relationships form a tree, or hierarchy, the root node of which is the global function and the leaves of which are the primitive subfunctions. The next step is to determine the input and output for each of these functions, associated data stores, and timing requirements. Finally, the interfaces between the functions, that is the data passed between functions, must be defined.

The choice of which functions in the hierarchy are to be implemented in software and which in hardware can be made at this point. However, if Rules [4] through [6] of the FTC Rules have been followed, it will be possible to change these assignments later with minimum disruption. Similarly, partitioning of functions among processor classes is done at this point, but can be altered later with minimum side-effects.

3.2.2 Top-Down/Structured Module Design

A subset of the hierarchy of system functions can be implemented either in software or hardware. In the case of software functions, a subset of software functions will be implemented as separately-compiled programs. Each of these modules is decomposed, top-down, into subfunctions which are implemented as internal procedures (software) or sub-units (hardware), a process known as top-down module design. The inputs and outputs (parameters) of each procedure are defined, as well as its data stores (software) or signals

(hardware), in the same way as for system functions. As the modules are decomposed into submodules, proper structured design techniques will be used to maintain the structural soundness of the over-all system.

3.2.3 Top-Down Implementation.

Implementation of a module designed top-down is done by implementing the top-level or global procedure first, with the procedures of the next level implemented only as dummy procedures containing no usable body. These are called "stubs". Following the integration of the implemented module into a skeleton system and the testing (Section 3.2.4) of this procedure, the stubs are "expanded" or developed in full, with their dependent procedures written as stubs. This method is repeated at each level of procedure until the low-level or terminal procedures have been implemented.

3.2.4 Top-Down Testing

As mentioned in 3.2.3, the global procedure in a system is tested while the second level procedures exist only as stubs. As each stub is expanded, the system can be tested with possible lower-level stubs. Errors detected during testing can, in most cases, quickly be identified with the particular stub that has just been expanded.

Top-down testing will, in fact, be applied not only to local module testing but to system testing. As soon as the system functional hierarchy has been defined, a test plan will be designed. The top-down module of the system will be implemented, and testing will begin with stub modules representing the second-level functions of the hierarchy. Testing will proceed top-down, with implementation of the modules proceeding level by level. This process produces a very localized feed back loop consisting of module design, module implementation, module integration, module testing, error detection, module correction. System integration testing as a major development phase will be non-existent.

3.3 Software Fault-Avoidance

The following methods will be given serious consideration in the design, implementation, and testing of software for the AASC.

1. Structured system design and analysis.

2. Top-down program design (see Section 3.2)

3. Structured coding. These well-documented techniques impose discipline on coding and enhance verifiability of procedures.

4. Top-down Testing (see Section 4.)

5. Software fault-tree analysis. This technique promotes the coverage of software faults by formalizing the design of acceptance tests for recovery blocks.

6. Software fault-seeding/mutation analysis (see Section 4.)

Use of the above techniques will reduce the probability that a software module contains a fault when the AASC begins a mission.

The concept of fault tree analysis requires some clarification. Since any system may be recursively decomposed into functions (this is the underlying principle of top-down design), it follows that a similar decomposition can be applied to system failure.

As an example, the hierarchy chart for a simple real-time system for providing temperature readings based on readings of transducer channels is given in Figure 4. A fault-tree analysis of the same system is shown in Figure 5. The starting point in fault analysis is to assume that the system can fail. From the first-level functional decomposition, we can deduce the failures at this level that can cause system failure. Each first-level failure can cause system failure, so there is an implied disjunction among all these conditions. In addition to faults which are intrinsic to each function, there are general faults that can cause failure at any level: loss of context (the loss of information about what task to do next) and incorrect parameter-passing between functions.

Each of the first-level failures can in turn be analyzed for failures in second-level functions, and similarly at the third level. Carrying the functional decomposition to a more detailed level in the hierarchy chart would allow a more detailed fault-tree analysis.

Fault-tree analysis can be used to:

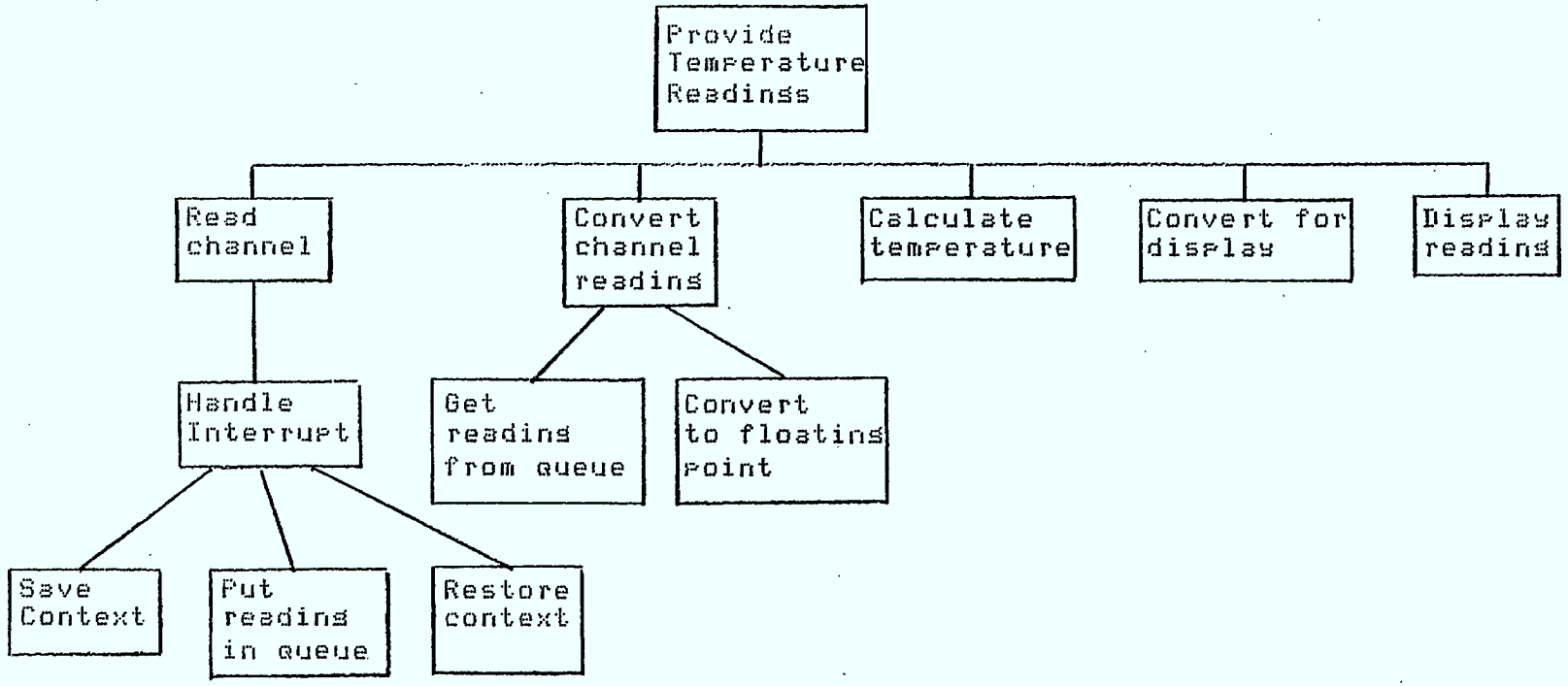


Figure 4. Hierarchy chart showing functional decomposition of a simple process.

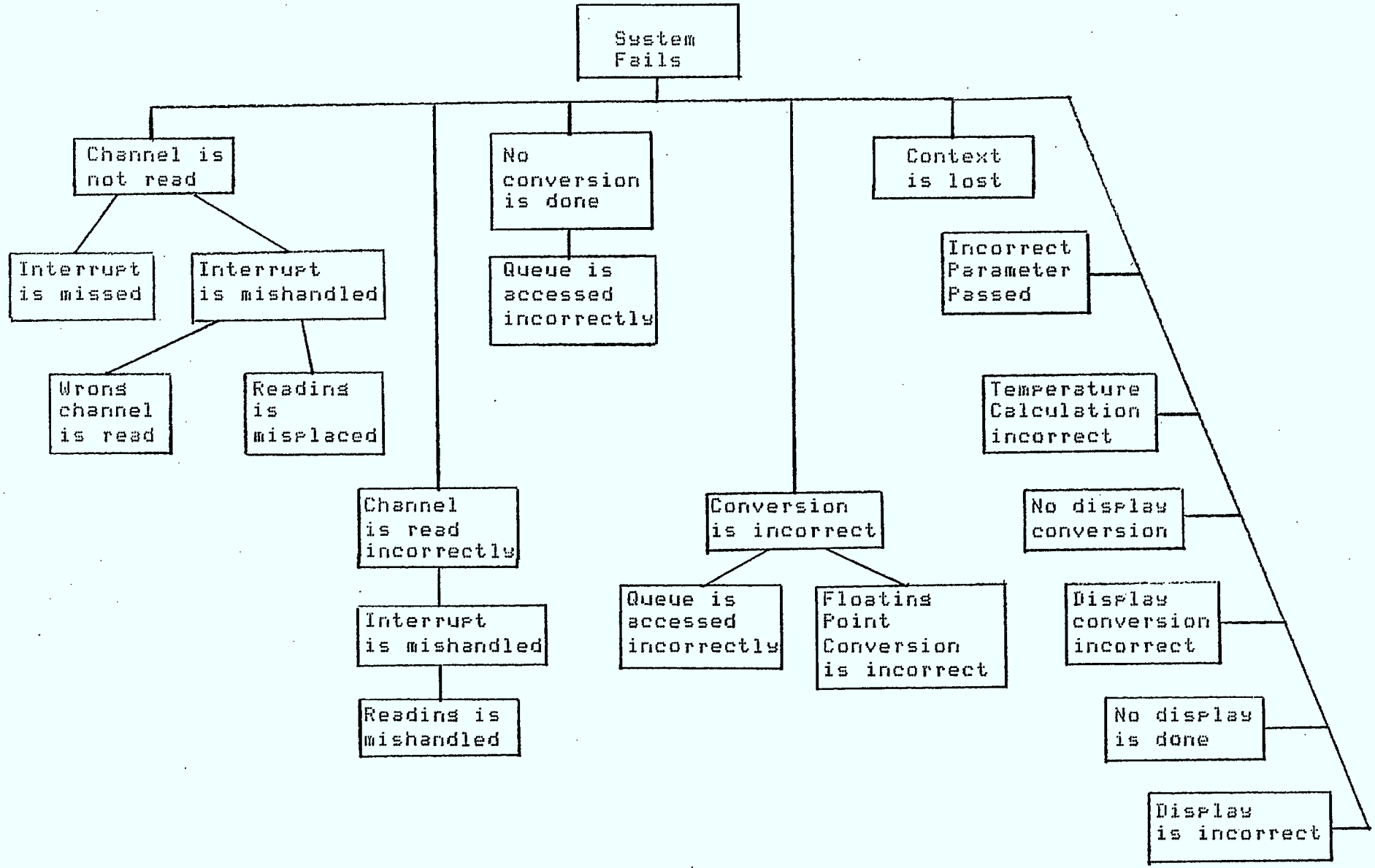


Figure 5. Fault-tree analysis of simple process

a) determine partitioning for recovery blocks

b) determine acceptance tests for recovery blocks

c) determine sequence and criteria for testing

d) determine isolation regions for software fault-tolerance.

Fault-tree analysis can be applied at the system, node, processor, process and module level.

3.4 Networking Design Principles and Standards

The structure and design of communication architectures has been researched in depth by the telecommunication and distributed data processing industries. From this work, a number of fundamental design principles have been identified which the authors recommend be applied to the development of the AASC. These principles are as follows:

(1) Layered Communications Structure

A layered structure be required such that the communication procedures used between two systems (e.g. a satellite and a ground computer) be partitioned into a hierarchical set of procedures (i.e. protocols). In this structure, each protocol provides a set of services (e.g. sequencing, flow control, check point signalling etc.).

Each partition in this structure is called a layer. Each layer uses the services provided by the layer below, plus the functions performed by the protocol in its own layer, to offer enhanced services to the next higher layer.

(2) Criteria for Selecting Layers

There are a number of criteria for partitioning the communications structure into a set of layers. The number of layers should not be too high to avoid inefficient communications. Too few layers can result in protocols that perform many functions and thus are very complex. The consequence is that proof of protocol correctness and validation of implementation are difficult.

Layers should be selected where alternate technologies (and hence protocols) are anticipated. This criteria will allow the use of new communication technologies without requiring the re-design and implementation of the entire communication structure.

(3) Performance

The communications structure should allow performance criteria to be dictated by the needs of the applications. The following are statements of objectives associated with the major performance criteria:

a. Throughput

The communications structure should manage the sharing of limited communication resources across applications. If necessary, the full bandwidth should be available to a single application. The simultaneous use of multiple connections to achieve the required capacity should not be precluded.

b. Reliability and Availability

The communication structure should offer a range of transmission integrity checking and recovery services.

Recovery either through error detection and re-transmission (or re-generation) or reconnection of a logical link (perhaps using a different physical circuit) should be an intrinsic feature and should be a service provided to application processes.

c. Transparency

The application should be able to send arbitrary sequences of bits without concern that they will be interpreted by lower layer protocols.

d. Recovery from End Application Failure

The structure should facilitate the recovery of information when one of the communicating applications fails. Techniques such as check point/restart protocols provide this feature.

(4) Interconnection to Ground and Other On-Board Systems.

Multiple applications in the AASC must be able to communicate with application processes in a variety of computer systems distributed on the ground and elsewhere in space. An example of such interconnections is seen in Figure 6.

The standard communication architecture called Open System Interconnection (OSI)[ISO 79] satisfies the above layering, performance and interconnection requirements. Given the trend to use this architecture when interconnecting heterogeneous computing systems, we recommend that the use of the OSI architecture be an AASC design requirement.

3.5 Designs for Fault-Tolerance

While adherence to FTC rules and top-down/structured development has implications for all design decisions, the design of fault-tolerance mechanisms deserves particular attention.

FTC Rule [1] implies that a distributed processing solution will be chosen for the design, since it removes single points of failure. Similarly, the functions of diagnosis, containment, and analysis of, and recovery from errors should be distributed, so that no single module is responsible for fault diagnosis. There are at least two classes of designs for distributed fault-diagnosis - hierarchical and "democratic" in which all nodes in a network have the potential for diagnosing one or more other nodes. The FTC rules bias design towards the democratic model, but it is possible that the choice will be constrained not by design rules but by the existence of appropriate algorithms for diagnosis of node faults and link faults [KUHL 81, MCPH 81].

FTC Rule [1] has implications for the choice of fault-tolerant data structures. A single point of failure in a list, linked list, tree, etc. is discouraged by the rule. The concept can be extended to encourage tolerance of N failures in the data structure, since it is quite easy for a software or hardware error to destroy a set of data items in a data structure in a single event. Data structure design should, hence, incorporate robustness [BLAC 81] as a fault-tolerant feature.

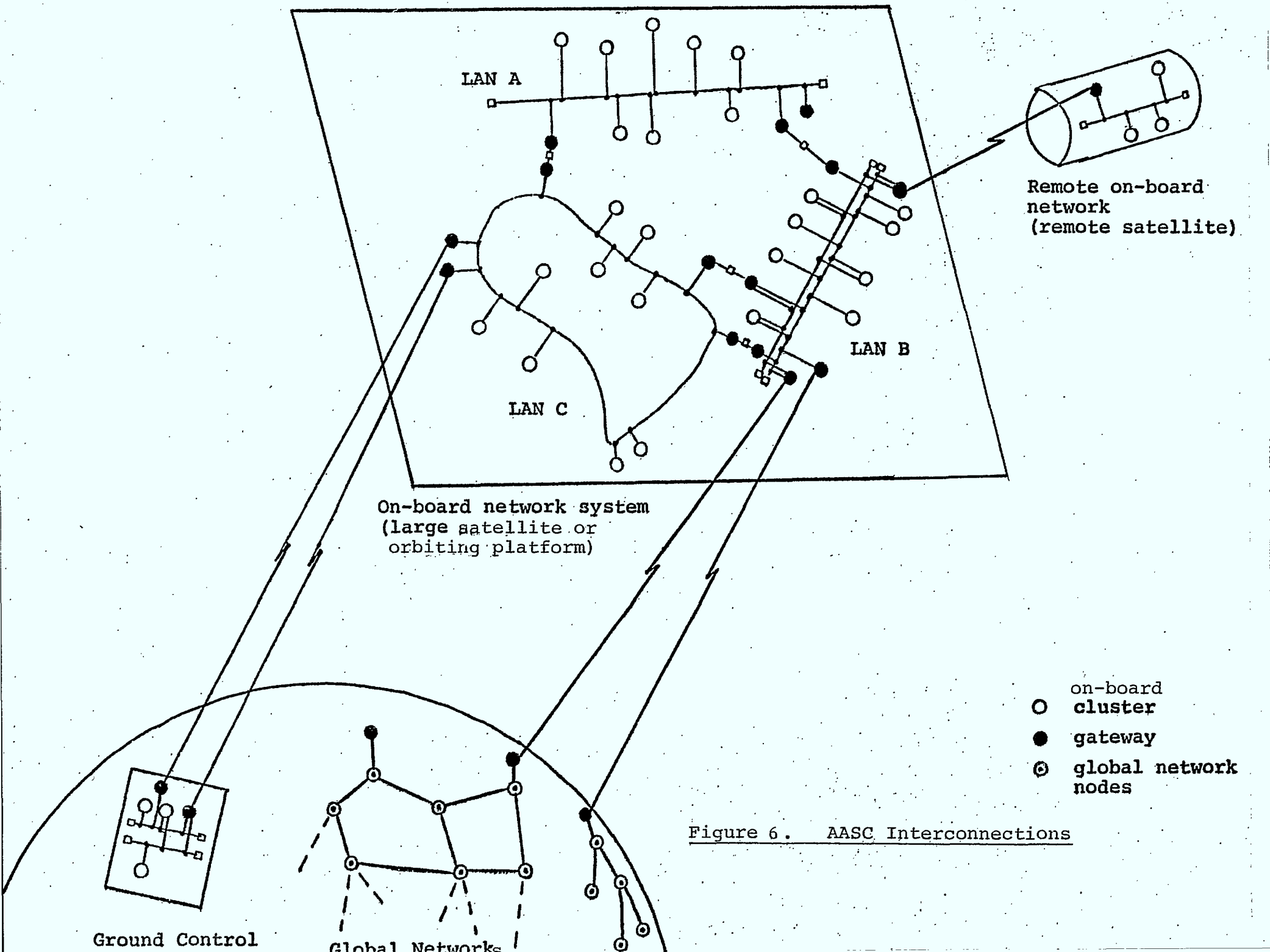


Figure 6. AASC Interconnections

4. The Functional Test of the AASC

Purpose

The purpose of testing the AASC is to establish:

- compliance with the functional specification
- confidence in the design
- consistency of performance.

Principles

It is widely acknowledged [ANDE 82, LEVE 82a] that it is not possible to achieve 100% avoidance of faults in building a system. Similarly, it is impossible to ensure that system testing will expose all the faults remaining after the application of fault-avoidance techniques. However, certain principles have been applied during this study to the system development process in order to obtain a high degree of software quality assurance and thus to maximize fault-avoidance. These principles are equally appropriate to the area of testing and validation, for which the same goals apply. A top-down, structured approach to testing will be used and the application of redundant methods will be considered, particularly in areas recognized as most likely to harbour critical faults.

Concurrent Testing

Testing is seen as a continuous process involving all levels of the system hierarchy and all stages in its evolution [HOWD 82]. Tests should be applied at each phase of development and to each concept, level or module as it is formed, with reference to the real-world environment in which the system is designed to operate. The nature of this process is shown in Figure 7, which is derived from Howdon [HOWD 82].

Test Plan

An attempt must be made to validate the system formally. The achievement of high system quality requires the incorporation of verification into each phase of development [ADRI 82]. To this end, the test plan is considered to be an integral part of development. It will take into consideration these factors:

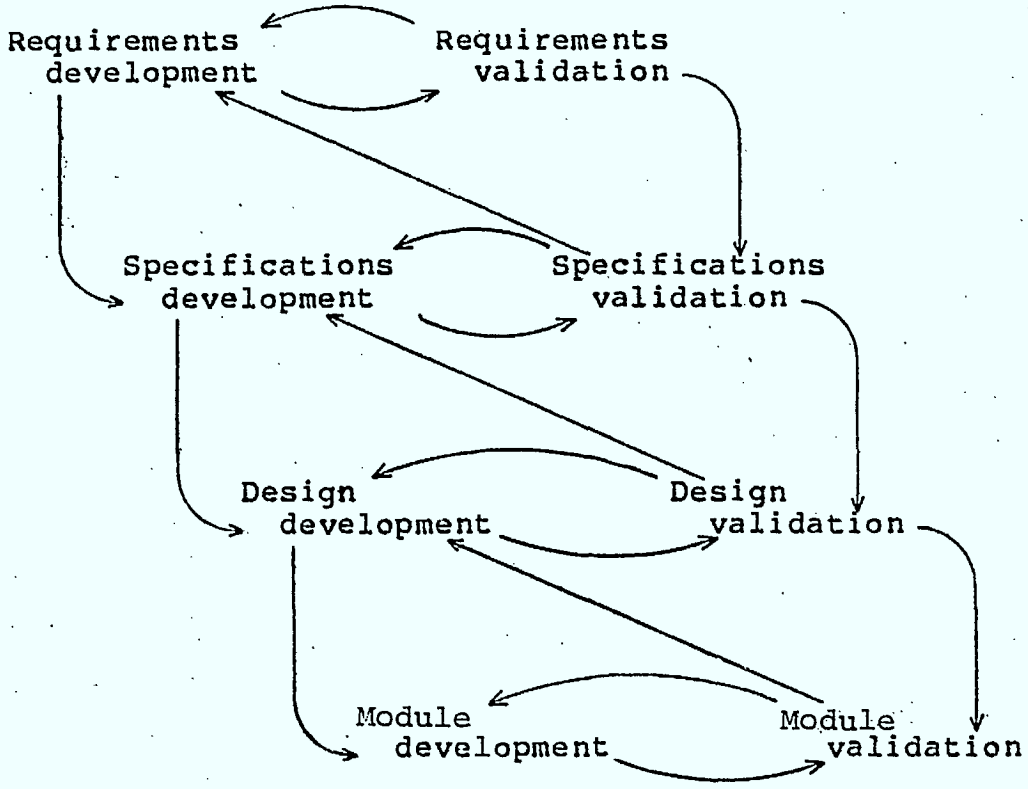


Figure 7. Integrated validation and development.

Test Scope:

All parts of the system hierarchy and aspects of its development should be covered by testing and validation. Testing should include a check of the basic system concepts against a real-world model. These concepts include reliability, availability, flexibility, reconfigurability, performance, and general conformity to the FTC rules. It should continue throughout development to make reference back to the previous stage. This will enable any changes made to be tested as they are made and increase confidence in the implementation of the specification.

Exhaustive testing, the technique of testing every element of a domain, is the only known analysis technique which will guarantee the validity of a program [ADRI 82]. This is the ideal but where this is practically impossible, criteria must be selected to enable economical, feasible and yet representative testing to take place.

Testability:

To be testable, software must be understandable and measureable [ADRI 82]. Understandability requires that each stage be structured, concise and self-descriptive. Measurability requires the possibility of instrumenting, probing, testing, and evaluating at each stage of the development. The requirement for testability at each stage calls for the creation of the test plan at the earliest possible point. This should be done in conjunction with other design plans to ensure that measurability is, indeed, a feature at all levels.

Timely detection of errors:

Emphasis should be placed on the early detection of errors. Errors occurring in the requirements or specification stages are harder and more expensive to catch and persist longer than other types [HOWD 82, LEVE 82b, SCHI 82]. Validation should, therefore, be applied at the earliest opportunity.

Fault Types:

Knowledge of what faults are being tested for is essential. An analysis of fault-types should be made prior to making the test/design plans. This should cover the criticality of fault types and

areas of high fault concentration [SCHI 81]. Awareness of these factors should lessen the chance of building faults into the design. Fault-tree analysis will considered for this, as well as other purposes.

Redundancy:

The use of redundancy should be considered when planning tests. Redundancy is appropriate in the use of:

- methods: redundant, independent methods may well ensure more complete error-trapping. Fault-type analysis will aid the selection of areas in which redundant tests should be applied, namely areas deemed most likely to contain critical fault types and high concentrations of faults.

- models: use of the same simulation model for testing purposes throughout will produce valid results only to the extent to which the model itself is fault-free. N versions of a model, preferably originated by different people, should be used. N different, independent versions used at each stage of a system's evolution will result, by being tested both against each other and against the previous stage, in the emergence of a preferred version(s).

Test Tools:

Serious consideration should be given to the acquiring and appropriate use of test and validation tools when the test plan is drawn up. They should be selected in accordance with the above requirements and their relevance determined in terms of functionality, module size and language.

The use of Ada as a specification and implementation language and an Ada Programming Support Environment (APSE) may obviate the need for some forms of testing. For example, type checking would be an inherent part of the APSE. However, this should not cause the requirement for redundancy of testing methods to be overlooked.

Tests are divided into static and dynamic [SCHI 82]. Static tests can be conducted during the construction stage of development on incom-

plete programs and will yield faster results. Dynamic tests are more productive in the later stages of development during execution of the code. Static testing tools include data-flow analyzers, path analyzers, coverage analyzers, interface analyzers, and cross-referencers. Some dynamic testing tools are assertion checkers, simulators, path-flow tracers, symbolic execution tools and mutation analyzers.

Requirements, specification and design documents will be subjected to document analysis in the form of document inspection and structured "walk-throughs". A checklist of properties such as consistency, necessity, sufficiency, feasibility and correctness should be drawn up and applied to each document. Walk-throughs, although expensive in terms of man-power, would be appropriate at any stage, and must be conducted as frequently and informally as feasible.

Fault-seeding, referred to earlier in Section 3.3, is a technique which "seeds" known errors into the implementation in a statistically similar manner to that of actual errors. Test data are then applied and the number of seeded and original errors determined. With the assumptions that seeded and unseeded errors are equally findable and that seeding and testing are statistically unbiased, the proportion of undetected, unseeded errors is ascertained. These assumptions, however, are open to question [ADRI 82]. A further development from this method is mutation analysis, which also involves error seeding. In this method, several mutant programs are derived from the original, each containing different errors or error sets. The program and mutants are run interpretatively on the test set. Results so far have shown that test sets which showed scores of 0.95 or more did not produce any further errors in subsequent use. It is recommended that this method be seriously considered for use in the test plan.

Tests of specific concepts will require the use of estimation tools, for example the use of Aries 82 [MAKA 82] in the evaluation of reliablility. Constant referral should be made to the FTC rules, particularly in the specification and design

stages.

5. CONCLUSION

The main objectives of the AASC have been laid down and definitions and assumptions regarding reliability have been formed. A basis for partitioning reliability into hardware or software has been indicated. Software fault-avoidance techniques were examined. Findings in this area were that:

- N-version programming can be used selectively and would, therefore, be cost-effective in use;
- use of recovery blocks provides an extension beyond N-version programming but is rigid in application. Recommendations to counteract this rigidity were made;
- Ad hoc measures can be useful in circumstances when neither of these two methods apply.

Reliability values important on spacecraft missions have been established. Conditions for the achievement of diagnosability and recovery are determined.

The essential objectives were outlined for the achievement of flexibility of system hardware and software. These involve mission-to-mission application changes; in-mission application changes; enhancement of components and relationships; and in-mission modification.

Performance criteria were established, including minimum throughput, floating-point requirements, memory transfer rate and multitasking capabilities and constraints.

Autonomy will be achieved through an Onboard Autonomy Manager and conditions have been established which apply to its operation.

Design constraints will be imposed during development to minimize the introduction of faults and ensure that the design criteria are upheld. These consist of the Fault Tolerant Computing rules; the application of a top-down/structured approach to development, system and program design, implementation, and testing; and the use of software fault-avoidance methods.

The design principles and standards which are applicable to networking have been outlined, namely a layered communications structure, performance criteria and interconnection to other systems.

Top-down principles were also applied to functional testing. The application of testing should be concurrent with the

development of the system. To ensure system testability, a test plan is to be created at the outset of development. Reference points for this plan should be system scope and testability, the timely detection of errors, redundancy of techniques, types of faults and testing tools.

REFERENCES

ADRI 82 Adrion, W.R., Martha A. Branstad, and John C. Cherniavsky, "Verification, Validation, and Testing of Computer Software", ACM Computing Surveys, Vol. 14, No. 2., June 1982.

ANDE 82 Anderson, T. and P.A. Lee, "Fault Tolerance: Principles and Practice", Prentice Hall. 1982.

GOMI 82a Gomi, T. and M.Inwood, "FTBBC - The Fault-Tolerant Building Block Computer", Eidetic Systems Corp., 1982.

GOMI 82b Gomi, T. and M. Inwood, "A fault-Tolerant On-Board Computer System for Spacecraft Applications", Eidetic Systems Corp., 1982.

HECK 80 Heckman, Paul J., Jr. "Free-Swimming Submersible Testbed (Eave West)", Technical Report 622, Naval Ocean Systems Center, San Diego, Calif.

HOWD 82 Howdon, William E., "Validation of Scientific Programs", ACM Computing Surveys, Vol. 14, No.2, June 1982.

ISO 79 "Reference Model of Open Systems Interconnection", ISO TC97/SC16/N537

LEVE 82a Leveson, Nancy G., and Shaula Yemini, "An Evaluation of Software Fault Tolerance Techniques in Real-Time Safety-Critical Applications", Technical Report 192, University of California, Irvine, November 1982.

LEVE 82b Leveson, Nancy, "Software Fault-Tolerance" AIAA/NASA Workshop on Applied Fault Tolerant Computing for Aerospace Systems. Texas, November 1982.

MAKA 82 Makam, Srinivas V. and Algirdas Avizienis, "Aries 81: A Reliability and Life-Cycle Evaluation Tool for Fault-Tolerant Systems", Proc. of 12th Ann. Symp. of F.T.Computing.

MARS 80 Marshall, Michael H. and G.David Low, "Final Report of the Autonomous Spacecraft Maintenance Study Group", February 1, 1981. JPL Publication 80-88

MYER 75 Myers, Glenford "Software Development by Composite

Design", 1975.

MYER 78 Myers, Glenford "Software Development", 1978.

SCHI 82 Schindler, Max "Software Testing - A Scarce Art Struggles to Become a Science", Electronic Design. July 22, 1982.

VERE 81a Vere, Steven, "Planning in Time: Windows and Durations for Activities and Goals", NASA/JPL, Nove. 1981.

VERE 81b Vere, Steven and Brad Wallis, "A Full-Scale Demonstration of Autonomous Spacecraft Sequencing", Information Systems Research Section (364), JPL.

YOUR 75 Yourdon, E. "Techniques of Program Structure and Design", Prentice-Hall, Englewood Cliffs, N.J., 1975.

CACC / CCAC
81631

CUNNINGHAM, I.
Functional specification for the advanced autonomous spacecraft computer

P
91
C655
G646
1983

DATE DUE
DATE DE RETOUR

SEP 07 1984

LOWE-MARTIN No. 1137