NASA Contractor Report 172282

NASA-CR-172282
19840019666

Fault Tolerant Architectures for Integrated Aircraft Electronics Systems

Task 2

Karl N. Levitt
P. Michael Melliar-Smith
Richard L. Schwartz

SRI International
Menlo Park, California 94025

Contract NAS1-17067

June 1984

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

NASA Contractor Report 172282

Fault Tolerant Architectures for Integrated Aircraft Electronics Systems

Task 2

Karl N. Levitt
P. Michael Melliar-Smith
Richard L. Schwartz

SRI International
Menlo Park, California 94025

Contract NAS1-17067

June 1984

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Contents

Plates

Overall Introduction

This report documents work performed under Task 2 of NASA Contract NAS1-17067. The work continues the work of task 1 into the architectural basis for an advanced fault tolerant on-board computer that will be the successor to the current generation of fault tolerant computers (SIFT and FTMP).

The work is reported here in four chapters:

- Chapter 1 contains an extended study of the NETS Network Error Tolerant System architecture, with particular attention to intercluster configurations and communication protocols, and to refined reliability estimates.
- Chapter 2 discusses the diagnosis of faults, so that appropriate choices for reconfiguration can be made. The analysis relates particularly to the recognition of transient faults in a system with tasks at many levels of priority.
- Chapter 3 describes a novel architecture for computer systems, the demand driven data-flow architecture, which appears to have possible application in fault tolerant systems.
- Chapter 4 reports on work investigating the feasibility of automatic generation of aircraft flight control programs from abstract specifications.

Chapter 1

NETS: Network Error-Tolerant System

Introduction

For the past 10 years NASA-Langley has been supporting the development of two fault-tolerant computers (SIFT and FTMP), prototypes of which are currently under evaluation as part of Langley's AirLab test facility. Although both SIFT and FTMP provide a reliability in the presence of permanent and transient hardware failures that far exceeds what is obtainable with conventional unreplicated computers, there remain deficiencies that appear to be inherent to the underlying architectural concepts. Among the deficiencies are:

- Limited capability for expansion beyond approximately 16 processors
- Limited capability to accommodate different processor types, including special purpose processors
- No immunity to transient faults that temporarily disable several processors

The basic problems with SIFT and FTMP are that, although they are multicomputers providing reliability through redundancy, fault-masking and logical

removal of faulty processors, they employ the centralized computer technology available when the designs commenced in the 70's. In particular, they require reasonably tight synchronization among all processors and direct communication between each pair of processors. The solution to these deficiencies appears to be a more distributed concept, employing the newly available distributed system technology of the 80's.

For the past year we have been studying an architectural concept we call NETS (Network Error-Tolerant System) as a possible successor to the SIFT and FTMP class of fault-tolerant systems. NETS consists of *clusters*, each of which has a direct communication link with only a few other clusters; thus, as is standard in computer networks, communication between non-neighbor clusters requires the passing of data through intermediate clusters. Each cluster is intended to be responsible for 1 (or possibly a few) task, and is likely to be located physically close to external equipment (sensors, actuators, etc.) associated with the task. A cluster might have internal redundancy to enable it to continue operation in the presence of faults – in particular permanent or transient faults that only impact, say, 1–2 processors. It is anticipated that each cluster will be a SIFT configuration of 1-5 processors, 5 processors being required where higher reliability for a task is mandated and 1 processor where the task is not critical.

An initial design and analysis of NETS has been completed. In the process of carrying out the design, we identified a number of difficult problems, most of which we have solved at least to the point of pragmatic, if not theoretically optimal, solutions. NETS appears to address the deficiencies of SIFT and FTMP, and to achieve the high level of reliability required for aircraft electronics systems. It is recommended that consideration be given to carrying out a detailed design of NETS, leading to a simulation and/or a prototype that could be evaluated in AirLab.

Section 1.1 describes in more detail the goals of an aircraft fault-tolerant system that motivated the design of NETS. The overview of the NETS architecture is presented in Section 1.2, with emphasis on the intercluster communication protocols and the requirements of the various executive-level functions. Of

primary concern is failures in communication links: how to mask errors that follow such failures, how to identify faulty links, and what communication protocols can avoid the use of known faulty links. Formulas determining the reliability of NETS under various redundancy and communication assumptions are derived in Section 1.3. Section 1.4 discusses desirable properties of the interconnection network. One desirable feature is that, for a given *fan-out* **d** from each cluster and a given *diameter* **k** (the diameter being the maximum number of hops between any pair of clusters), the maximum number of clusters **n** should be accommodatable. This turns out to be a "classical" problem in graph theory, called the **(n,d,k)** problem, studied extensively at SRI and elsewhere during the 60's. We summarize the relevant results. This previous work, unfortunately, assumes no failures of communication links. Our discussion derives some initial results on the effect of faults on the diameter of certain networks. For the particular class of networks, due to Akers, we show that the increase in diameter due to the need to avoid a single faulty link occurring anywhere in the network never exceeds 2; even more encouraging, for most networks in this class, there is no increase in diameter in the presence of a single faulty link.

The graphs discussed in Section 1.5 guarantee the existence of a path, between any pair of clusters, whose length does not exceed d. Algorithms for identifying the shortest path, particularly following a failure in communication link, are discussed in Section 1.5.

Each cluster, being a SIFT computer, will employ our previously developed algorithm for achieving synchronization among the processors within a cluster. Extensions to that algorithm to obtain network-wide synchronization are discussed in Section 1.6.

Section 1.7 discusses the related problems of recovery from massive transients and initialization of a newly connected cluster. It is shown that a cluster suffering a transient failure that corrupts data in all of the cluster's computers can be reinitialized through the efforts of the cluster's neighbors. Conditions for recovery from simultaneous massive transients are also discussed. Section 1.8 presents unresolved problems and suggests experiments that could be conducted on AirLab

to confirm our initial findings on NETS and to identify optimal ways of using NETS in particular applications.

1.1 Requirements of Advanced Fault-Tolerant Systems Addressed by NETS

1.1.1 High Reliability

As assumed for SIFT, the probability of a critical computation yielding an incorrect or late result is not to exceed 10^{-10}/hour over a 10 hour period. Given the current reliability of processors (even those developed using VLSI technology), this low probability of failure can be achieved only with redundancy. Under certain simplifying assumptions, it is easily shown that for a given amount of redundancy, the smaller the replaceable unit the higher the reliability. This property is seen by considering a 5-fold replicated system (e.g. a centralized SIFT system). Assume the probability of failure of each processor is p, yielding a system failure probability $P_c = 5p^4$, for small values of p. This calculation assumes that a fault in a processor is detected and the processor is logically removed from the configuration prior to the occurrence of a subsequent fault. Thus system failure occurs upon the 4th failure, at which time there remains one failed processor and one working processor in the configuration. In contrast, consider a highly *partitioned* system which consists of n SIFT systems, each of which is 5-fold replicated, but where each of the SIFTS in this case performs $\frac{1}{n}$th of the work as compared with the original system to a first approximation we can assume that the probability of failure of a processor is proportioned to its size; hence the failure probability of an individual processor is $\frac{p}{n}$. For this partitioned system, the probability of failure $P_p = 5n^{-4}p^4$, or $P_c = n^4 P_p$. Of course, this simple calculation ignores the effect of any fixed, processor size-independent *overhead* associated with executive routines. Assume an overhead portion that contributes p_o to the failure probability of a processor independent of the size of the processor and also assume (very pessimistically) that $p_o = \frac{p}{n}$.† Then, $Pc = (2^{-4})(n^4)P_p$.

†With this latter assumption, half of the computation carried out by a partitioned system processor is overhead.

For all but very small values of n, then it is seen that there is significant gain in reliability to be realized through partitioning.

1.1.2 Expandable and Contractible

In the current SIFT architecture, each processor has a direct connection to every other processor through a broadcast link. This property limits the number of SIFT processors in a given system to about 16 – giving a range of about 3:1 from a minimal system to a maximal system. A larger range – perhaps of the order of 10:1 – would be desirable for a fault-tolerant computer to be useful for the full range of NASA applications.

1.1.3 Capability of Using Different Processor Types

One attractive feature of a network-based system is the capability to accommodate different processor types within a given system. Among the processor types could be special purpose processors (e.g., navigation computers, air data computers, etc.) in addition to general purpose processors. SIFT and FTMP, requiring tight synchronization among the processors, do not easily accommodate a wide range of processor types, particularly if the clock rates are different. Moreover, the overall reliability of the system can be improved by the use of different processor types. Our reliability computations assume processor faults occur independently. If faults are correlated, the actual reliability will be significantly lower than computed. Independence of faults is more likely if the processors have different designs and different manufacturers. Moreover, the use of different processor types will make certain software faults (e.g., in the implementation of compilers) more independent and less likely to result in system failure.

1.1.4 Immunity to Massive Transient Faults

The current fault-tolerant systems cannot tolerate a transient fault such that

values of data in a majority of the processors are modified. Such a fault could be caused by a lightning strike or by power surge. Although not absolutely precluding global damage from massive transient faults, the physical separation of processors afforded by a network-based system should help localize the corruption caused by such faults and provide the opportunity for recovery.

1.1.5 Ability to Interface to Distributed Smart Sensors and Actuators

The trend in aircraft electronics system design is to sensors and actuators which are "smart", i.e., which provide on-site computational power. The current SIFT system interfaces with such devices through conventional Input-Output channels. A more attractive approach is to consider these devices as part of the overall network, thus enabling the more effective use of their computational power.

As described in the following sections, the NETS architecture can satisfy all of the above goals.

1.2 Overview of NETS Architecture

This section presents a brief overview to the NETS architecture. Issues covered are

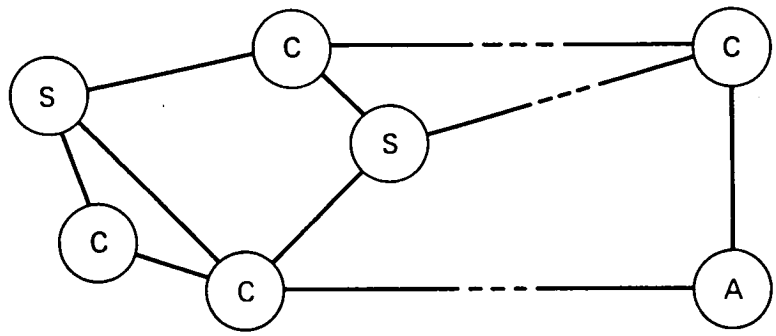
- The organization of the architecture as a network of clusters, and possible protocols for intercluster communication, including the accommodation to faulty links
- The combination of clusters of different redundancy
- Requirements for an overall network executive that is distributed among the nodes of NETS

1.2.1 Clusters and Their Communication Protocols

The computational unit in NETS is called a *cluster*; the clusters communicating with each other through a network. As illustrated in Figure 1, a cluster is a site that can be associated with a sensor, an actuator, or can be a computation cluster whose role is to generate outputs in response to inputs. A *sensor* cluster will have no logical inputs, and an *actuator* cluster no logical outputs. A *computation* cluster will have both logical inputs and logical outputs. The interconnection network need not be a *complete* graph; that is, each cluster need not have a direct connection to every other cluster. Hence intermediate hops will be required when a pair of nonadjacent clusters communicate with each other.

In generating the value to be delivered to an actuator in response to sensor inputs, all three types of clusters could be involved. A chain of tasks (in general a tree if it is assumed that more than one sensor is involved) cooperate to generate the output, the sensors providing the inputs, the computation clusters generating intermediate values, and the actuator generating the final value. In its most general form, the graph of these clusters will be as indicated in Figure 2. The sensors and actuator clusters are "stubs" hanging off a general graph (containing

loops). The loops are present to account for one (or more) computation cluster executing more than one task in the chain of tasks.



S: SENSOR CLUSTER
A: ACTUATOR CLUSTER
C: COMPUTATION CLUSTER

FIGURE 1 NETS IS AN INCOMPLETE INTERCONNECTION OF CLUSTERS

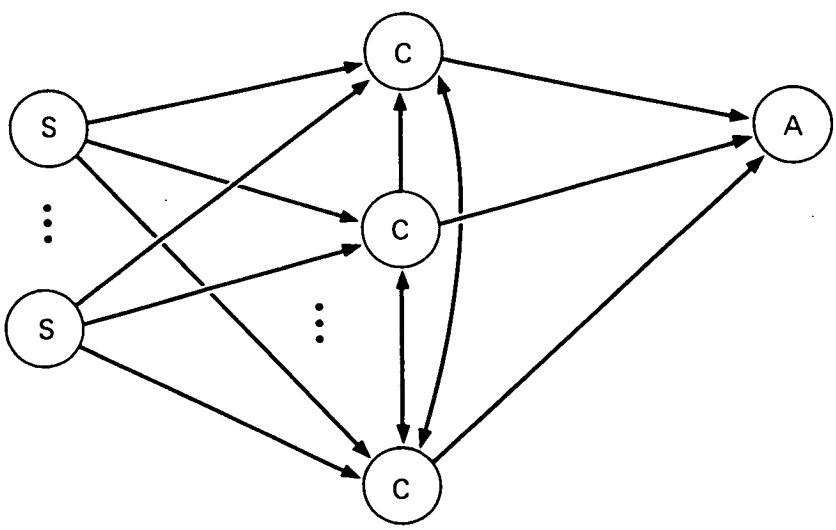


FIGURE 2 THE GENERAL FORM OF THE CLUSTER GRAPH

Each cluster is configured as a SIFT computer, i.e., a complete interconnection among a set of processors. It is expected that a given NETS system will have clusters of different size; we call such a cluster an *n-SIFT*. We will represent the internal structure of a n-SIFT as the schematic illustrated (for n=5) in Figure 3. It is likely that the maximum value of n needed for currently envisioned applications is 5, as the probability of failure of a 5-processor SIFT is quite low. Of course as discussed below, for less critical tasks clusters containing fewer than 5 processors will suffice.

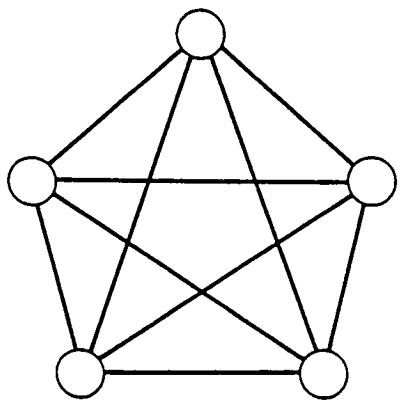
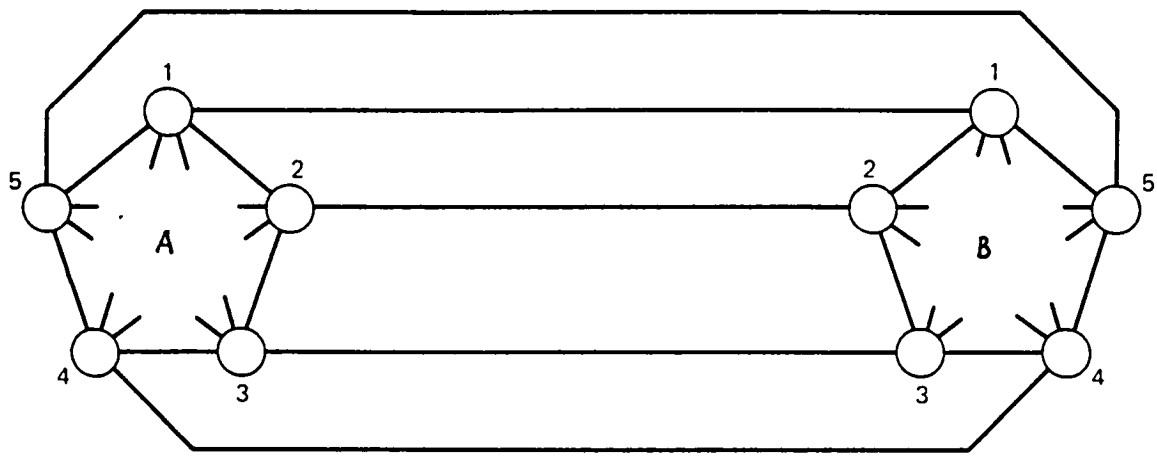


Figure 3. Within a NETS Cluster the Interconnection is Complete

Let us now consider the structure of the interconnection between a pair of clusters. As illustrated in Figure 4 for the two 5-SIFTs, there is a single link between corresponding processors; for the current discussion let us assume that the link is bidirectional, although unidirectional links are possible. Assume that a task **a** executing on cluster A is required to transmit data to a task **b** executing on B. In the absence of failures, each A processor will send the data to its corresponding B processor. When all of the B processors have received the data, they exchange the received values and vote. What if a link suffers a failure? Since a link is just a wire connecting a pair of processors, it is convenient to view a link failure as a failure in *either* of its associated processors. Accordingly, an error resulting from a link failure can be masked as long as there is adequate voting margin; what

constitutes an adequate voting margin for link failures is discussed in the next section. Furthermore, once the link failure is identified, that link can be avoided in future communication between A and B. The identification and reporting of link failures is quite straightforward, since it is simply the identification and reporting of processor failures. Thus the link, say, (A-2, B-2) will be assumed to be faulty under any of the following conditions:



Link 2-2 fails if A2 or B2 fail

Figure 4. Interconnection between two 5-SIFTs

1. The processors of cluster B (A) determine processor B-2 (A-2) to have suffered a permanent fault through B-2 (A-2) being outvoted on some computation.

2. B-2 (A-2) reports itself to be faulty to more than one of its neighbors in B (A).

Note that in producing erroneous outputs B-2 (A-2) could be faulty itself or could have received erroneous data from its neighbor in the other cluster – A-2 or B-2. The *safe* policy here is to assume *both* A-2 and B-2 are faulty. Note, however, that a link could fail but the associated processors could still be capable of performing other activities, e.g., compute on behalf of tasks or transmit data

along other links (see below). Hence a policy less profligate in dismissing processors would be as follows: If a processor (say, B-2) is outvoted on data received from A, assume the failure could be either in A-2 or B-2 pending confirmation by subsequent error reports. That is, if no reports are received indicating that A-2 is unable to carry out its task processing activities, it is allowed to participate in all activities of A except the transmittal of data to B. Alternatively, future reports could indicate that B is likely to be working.

Once it has been determined, by either A or B, that a link is faulty, the cluster noting the failure informs its neighbor that the link is to be avoided. The direct exchange of failure information is possible if the links between processors are bidirectional; otherwise, as discussed below, the information must be transmitted through a path that contains other clusters.

The discussion above is concerned with the case where cluster B contains a task that requires data from its neighbor A. What if the destination of A's data is to be a third cluster C which is not a neighbor of A? For example, the transmission might require an intermediate hop through B. In this case, the working processors of A transmits the data to B using working links The working processors of B vote on the received values and then transmit the voted values to C, again using working links. For example, as illustrated in Figure 5, A would avoid the link (A-2,B-2)* and B the link (B-3,C-3), assuming these links were known to be faulty. By voting on the data received from A, B can mask any errors from newly failed links between A and B, thus increasing the chances for transmittal of error-free data to C. In addition, B can immediately take note of a failed link and inform A of the failure. We call this approach the *vote and forward* protocol. Through this protocol, errors are handled by the cooperation of the two processors connected by the failed link; no other clusters need participate. The disadvantage of the vote and forward protocol is the delay it introduces.

*Assuming A-2 has failed so as to emit garbage, B can ignore it by B-2 (if working) not reporting the value received from A-2 or by B-2's fellow clusters ignoring B-2.

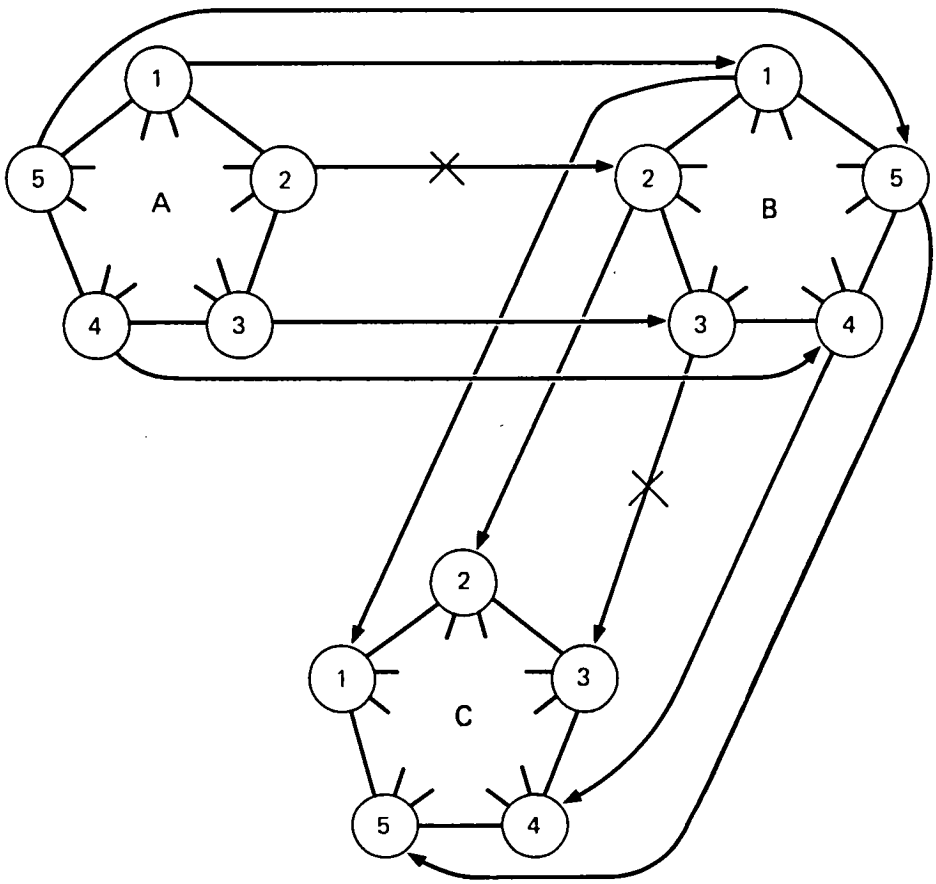


FIGURE 5 USE OF AN INTERMEDIATE HOP TO TRANSMIT DATA FROM A SOURCE TO A DESTINATION

A different protocol, which involves less delay, is called the *forward and vote at destination* protocol. Data received by an intermediate cluster is forwarded to the next cluster on the path *without* voting. Once all of the replicas arrive at the destination cluster, they are voted on. The successful masking of errors requires that the paths taken by the different replicas be *nonoverlapping*, otherwise a single link failure could cause correlated errors. A further complication is in locating faulty links, as a single error report can only locate the error to a path which might contain a number of links. It would be necessary to exercise, subsequently, each of the suspected links to try to locate the faulty link.

1.2.2 The Combining of Clusters of Different Redundancy

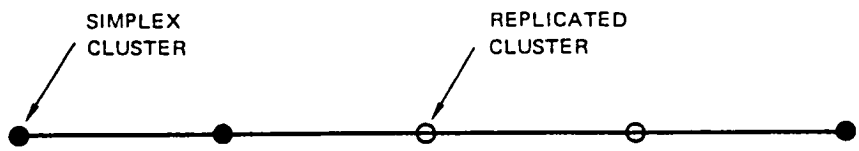
One of the goals for NETS that we indicated in Section 2 is the capability to handle tasks of different criticality without enduring the penalty of excessive redundancy for the less critical tasks. NETS can achieve this goal through the use of clusters containing different numbers of processors. Highly critical tasks would be assigned to clusters containing 5 processors; less critical tasks to 3-processor clusters; uncritical tasks to 1-processor clusters.

If clusters containing different levels of redundancy are to be combined in a single NETS, one approach is to segregate the clusters of a given redundancy to their own subnetworks of NETS. However, it is possible to combine clusters of different redundancy without compromising the reliability goals.

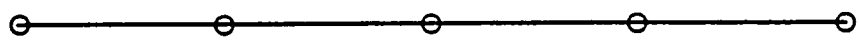
What are acceptable communication paths between clusters, particularly if the paths might involve clusters whose redundancies are not the same? Let us consider a source cluster A sending data to a destination cluster B through a path containing other clusters. Assume that A and B have the same redundancy n. If the *forward and vote at destination* protocol is used, the ideal is that the number of distinct paths from source to destination be equal to n. The minimum requirement is that at least 3 distinct paths be used to protect against any single point failure.

If the *vote and forward* protocol is used, all clusters on the path should ideally have the same redundancy as that of the source and destination clusters. Figure 6 summarizes the various possibilities.

One additional issue in combining clusters of different redundancy is balancing the load on each processor. That is, if m 3-SIFTS are to be neighbors of a 5-SIFT, what should the interconnection pattern be so that each of the processors of the 5-SIFT have approximately the same fan-out? The solution is quite simple, and is best illustrated through example.



(a) Acceptable Path
End nodes – simplex
Intermediate nodes – simplex or replicated



(b) Acceptable Path
All nodes replicated

(c) Unacceptable Path
End nodes – replicated
Intermediate nodes – simplex or replicated

Figure 6. Communication Paths containing Simplex and Replicated Clusters

Assume that the number of external links to each of the 5-SIFT processors is not to exceed 2. Then, as illustrated in Figure 7, three 3-SIFTS can be neighbors of the 5-SIFT using the interconnection pattern indicated. Each of the processors of the 5-SIFT, except 2, have a fan-out of 2. The interconnection pattern can be described using a matrix notation, as below.

	Processors of 5-SIFT				
3-SIFTs	1	2	3	4	5
A	x	x	x		
B	x			x	x
C		x		x	x

According to the matrix, the 3 processors of A are connected to processors 1,2, and 3 of the 5-SIFT respectively, etc.

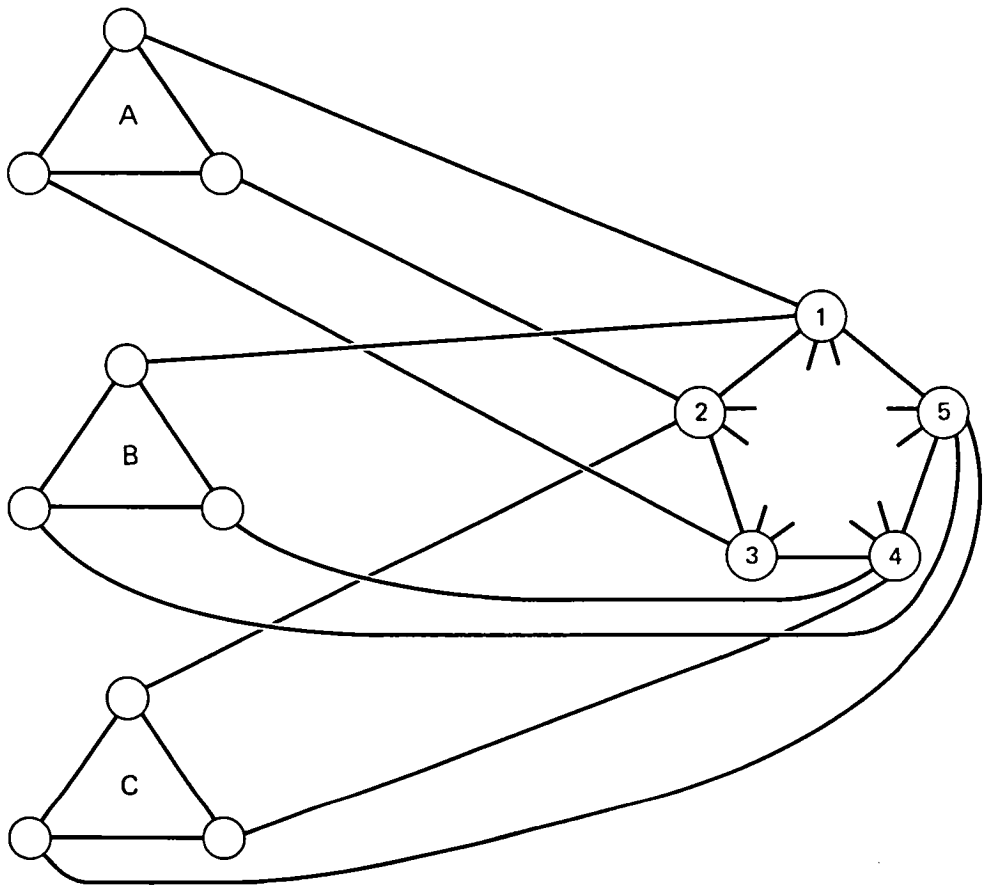


FIGURE 7 A BALANCED CONNECTION OF 3-SIFTs WITH A 5-SIFT

In general, if the allowed fan-out from each of the 5-SIFT processors is d, then the maximum number of 3-SIFTs that can be connected is given by the quotient $\frac{5*d}{3}$. Thus for d=3, the following matrix will apply.

	Processors of 5-SIFT				
3-SIFTs	1	2	3	4	5
A	x	x	x		
B	x			x	x
C		x	x		x
D			x	x	x
E		x		x	x

Note that the data passed to a 5-SIFT from a 3-SIFT will be subjected to only 3-way voting.

1.2.3 Requirements of NETS Executives

It is envisioned that each of the clusters of NETS will run the SIFT executive: local executive, error report, global executive, etc. To manage the network itself, a **network executive** is required. The network executive functions, distributed among the clusters, are the following:

- Apply the vote and forward protocol to messages destined for other clusters. It is envisioned that each message will have a destination tag, and each cluster will have a table indicating which neighbor to use for each possible ultimate destination.
- Receive and process error reports from neighboring clusters. The processing will identify faulty links, which are to be avoided in subsequent communications. The identity of a suspected faulty link is broadcast to the cluster at the other end of the link.

- Determine *optimal* paths to be used in the communication of data between clusters, where optimal means shortest. As discussed in Section 1.5, this function will be invoked when enough links between a pair of clusters have failed, thus precluding the reliable communication between these clusters. Many communication paths might have to be changed. As we show, the determination of new paths can be carried out *locally* in the sense that each cluster decides on the new optimal paths from information received from its neighbors.
- Participate in the initialization of newly connected neighboring clusters and in the recovery of neighbors that have suffered massive transients that temporary disable a majority of the cluster's processors. The approach to both of these problems is discussed in Section 1.6.

1.3 Reliability Assessment

In this section we consider the reliability achievable by NETS. We consider the following failure modes: (1) permanent faults – system failure due to exhaustion of spares, and (2) permanent faults – system failure due to buildup of faults beyond the voting margin before reconfiguration is completed. For each of the modes we consider separately the cases of (a) cluster failure, preventing it from performing tasks; and (b) link failures, preventing a cluster from communicating with *any* of its neighbors.

It is shown that acceptable reliability – better than the basic requirement of 10^{-10}/hour for critical tasks – can be obtained, even for relatively large NETS systems, assuming that (1) all critical tasks are executed on 5-SIFT clusters, (2) all communication between critical tasks is through 5-SIFT clusters using the vote and forward protocol, and (3) the fan-out from each cluster is at least 2. It is encouraging to observe that the reliability requirement is achieved with such a modest fan-out. Other requirements (e.g., keeping the communicating paths short) will probably dictate a higher fan-out.

1.3.1 System Failure Due to Exhaustion of Spares

We consider a NETS system to be a network in which each node is a 5-SIFT. (Clearly, we do not imply that all of the clusters must be 5-SIFTS; Our intention is to derive a lower bound on the reliability that critical tasks would experience.)

Let us assume, for this section, that faults become detectable errors that are handled very shortly after their occurrence. Thus a cluster will continue working through its third failure, as two working processors remain. However, the next fault spells the failure of the cluster, as one good and one bad processor would remain. Thus the probability of 4 (out of 5) processor failures is approximately $5p^4$, where p is the probability of a cluster failure. (Again, we are assuming faults occur independently of each other.) The probability of a failure of exactly one cluster in an N-cluster system is then $5Np^4$.

Now let us consider the probability of system failure due to a sufficient number of link failures occurring such that a cluster cannot communicate with any of its neighbors. We will only be enumerating those failure conditions that do *not* constitute cluster failure. Our initial assumptions will be as follows: (1) fan-out of two from each cluster, and (2) the forward and vote protocol; later we consider higher values of fan-out (which will provide improved reliability and the forward and vote at destination protocol.

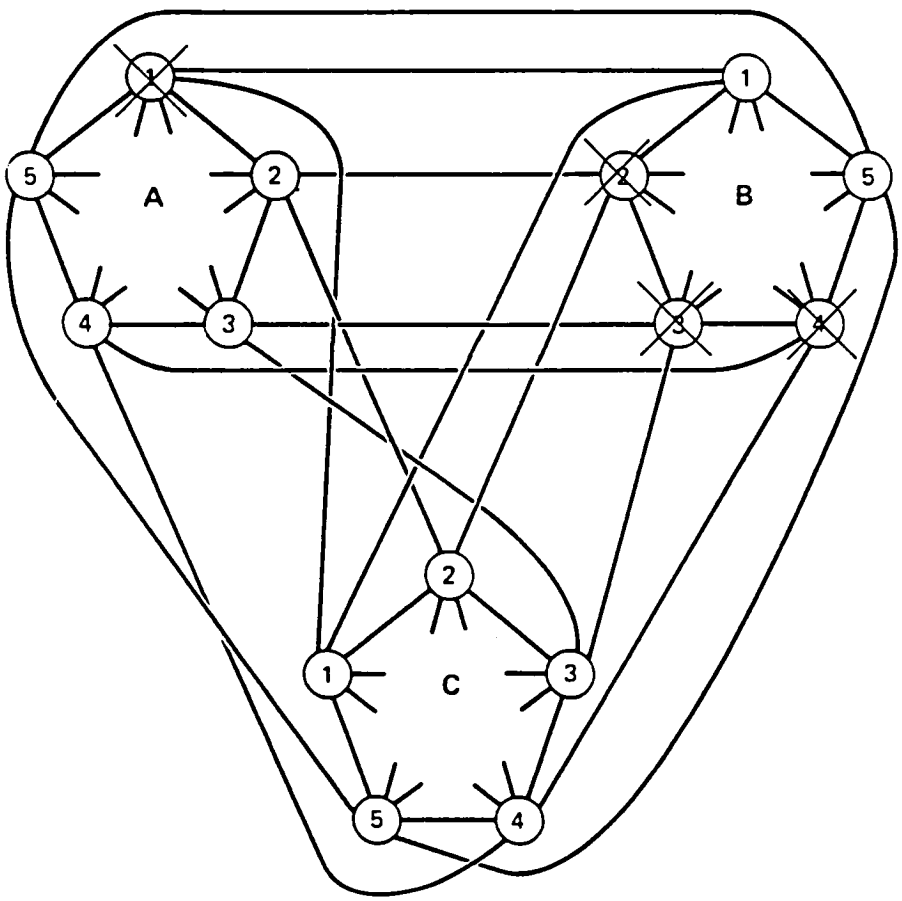


Figure 8. A pattern of Four Faults that does not cause Link Failure

Figure 8 shows 3 clusters; we will consider the probability that a fault pattern occurs that prevents A from communicating with either of its neighbors. A cluster I will be unable to communicate directly with cluster J, if 4 (or more) of the links connecting I and J are failed. Note that if 3 or fewer links have failed, the voting margin suffices to allow reliable communication under the assumption that a faulty link is ignored prior to the next occurrence of a faulty link. We first observe that all patterns of 4 failures spread over these 3 clusters are tolerated. (The reader is reminded that we are excluding from the enumeration those failure patterns that cause cluster failure, e.g., the failure of 4 processors in A.) An example of a pattern of 4 such tolerated failures is shown in Figure 8. The four failures indicated would prevent A from communicating directly with B, as only one good link (A-5,B-5) remains between these two clusters. However, 4 good links remain between A and C, and 3 good links remain between B and C, thus allowing communication between A and B to be through C. The other patterns of 4 faults among the 3 clusters are: two in each of two clusters, two in one cluster and one each in the other two clusters – all of which can be shown to be tolerated.

Now let us consider the patterns of 5 failures that are not tolerated. We will refer to Figure 9 for this discussion, where a pattern of 5 failures is distinguished. Assume a state where the failures of A-1, A-2, A-3, and B-4 have occurred and been noted. In this state A cannot communicate directly with B, but can communicate with B using C as an intermediary; A would use the links (A-5,C-5) and (A-4,C-4) in communicating with C. Now assume C-5 fails, but the failure is not detected. Clearly C might receive differing value on the two links it has been using to communicate with A, spelling possible failure of the system.

It can be shown that there is no pattern of five faults two of which are in A such that A will not be able to communicate with *either* B or C. Hence the only case of interest is three faults in a cluster, say A. If the two remaining faults are both in the same cluster, then A will have two working links on which it can communicate with the other cluster, thus avoiding isolation. Hence the enumeration need only consider three faults in A, one fault in B (such that A cannot communicate with B), and one fault in C (such that A cannot communicate with C). The number of

such patterns is given by

$$C(5,3) * C(2,1) * C(2,1) = 40.$$

Then, an upper bound on the probability of system failure due to link faults is $40Np^5$. This is an upper since some of the patterns covered for a cluster will also spell failure for a neighbor.

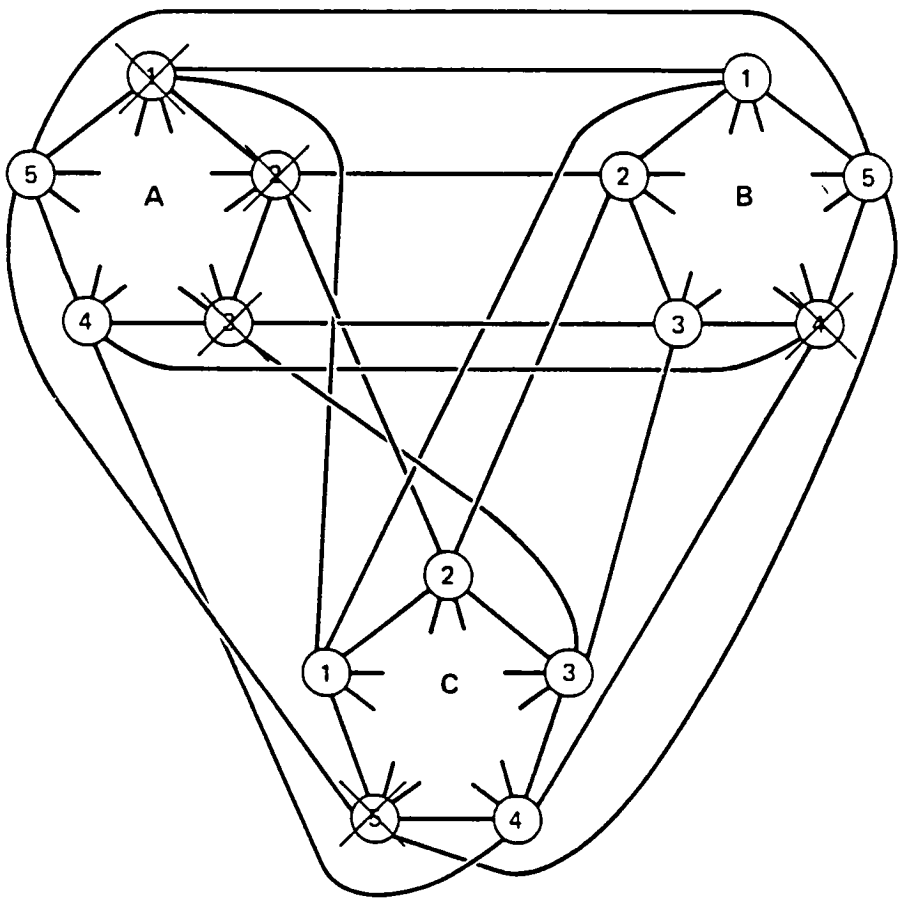


Figure 9. A Pattern of Five Failures that is not Tolerated

Note that for reasonably small values of p, the probability of system failure is dominated by the probability of cluster failure. Further note that the probability

of A not being able to talk with *both* of its neighbors is given by $100p^4$ (enumerating the patterns of four faults that cause either intercluster communication path to become unreliable). Exceeding the probability of cluster failure by a factor of 20, this is probably too high for critical computations. Thus it is necessary to allow for alternate communication paths in NETS.

If the intercluster fan-out is increased to 3, the probability of system failure due to link failure is decreased to $60Np^6$. It is probably not necessary to employ a fan-out of 3 from the standpoint of achieving reliable communication.

Now let us consider the use of the forward and vote at destination protocol. All that is required is that A have two or more working links – both to the same neighbor or one one to each of its neighbors. In this case all patterns of 5 failures are tolerated, but at the expense of the more complicated protocol.

1.3.2 System Failure due to Fault Buildup Prior to Reconfiguration

Unlike the adaptive voting approach assumed in the previous section, we are assuming here that faults are not detected and handled. Thus system failure will occur whenever three bad inputs are generated – either within a cluster carrying out a computation or in the passing of data between clusters. The probability of three faults in a 5-SIFT is given by $10p^3$.

On the other hand, the communication between a pair of clusters (A and B) will become unreliable when A suffers 2 faults (say in Aj and Ai) and B suffers one fault in Bl, l =j or i. The number of such fault patterns is

$$C(5,2) * C(3,1) = 30.$$

Thus the probability of system failure due to cluster failure and that due to link failures are comparable. Furthermore, there is no alternative to improving the reliability in this case short of increasing the redundancy level of the clusters.

1.4 Structure of the Interconnection Network

In this section we consider the interconnection network through which clusters communicate with each other. One key property of the network is that it allow clusters to communicate with each other with minimal delay. For the moment let us assume that each cluster has the need to communicate with every other cluster. (It is understood that this assumption ignores the possibility of assigning collections of tasks that communicate with each to collections of clusters that are close to each other; this possibility is discussed later.) Hence a measure of the quality of a network is the *diameter* k of the network. Here diameter is taken in the graph-theoretic sense to mean the following:

> Let the distance between any pair of adjacent nodes be 1. Let the distance between any pair of nonadjacent nodes i,j, be the length l_{ij} of the shortest path between i and j. The diameter k of the graph is the length of the longest shortest path, where the maximum is taken over every pair of nodes. Thus for a diameter k graph it is assured that no more than k hops need be taken in going between any pair of nodes.

As might be expected, the diameter of a graph generally decreases with the fan-out d permitted from each node. In the limit, if every node is connected to every other node, the diameter is one. However, we are seeking graphs in which the fan-out is much less than the number of nodes. In this case, a more comprehensive measure of the quality of the graph is the number of nodes n, for a given d and k, the general desire being to find graphs with maximum n. A graph having n nodes, diameter k, and fan-out d is called an *(n,d,k)* graph. A graph having the largest n for given d and k is called an $(n,d,k)_{max}$ graph.

One further complication is the impact of faulty links. We want the diameter to be low despite the occurrence of faulty links – say t such faults; whenever a fault occurs it is necessary to find a new shortest path that does not include the faulty link. A graph of n nodes, fan-out d, and diameter k in the presence of t or fewer faulty links is called an *(n,d,k,t)* graph. Again, we are, in general, interested

in maximizing n for fixed values of the other parameters – leading to $(n,d,k,t)_{max}$ graphs.

First let us consider the fault–free case. As discussed by Elspas[11], it is relatively easy to compute an upper bound on n for $(n,d,k)_{max}$ graphs; this bound has become known as the Moore bound. Consider the maximum number of nodes in a graph such that the distance from *one distinguished* node to *any* other node is no more than k, assuming a fan–out of d. Let **a** be the root node of a tree. Let there by d successors to **a**, as allowed by the fan–out limitation. Let each of these successors have d-1 successors, again as constrained by the requirement of fan–out of d. This construction can be continued to yield a tree of k levels below the root; the distances from the root node to the leaves of the tree is k, and the total number of nodes N_{max} is

$$N_{max} = 1 + d + d(d-1) + d(d-1)^2 + \cdots + d(d-1)^{k-1}$$

Simplification of this series yields

$$N_{max} = \frac{1 + d[(d-1)^k - 1]}{d-2} \quad \text{for} \quad d \geq 3$$

This tree construction is seen to maximize the number of nodes such that the distance between *one* node and the other nodes in the graph does not exceed k. Of course, the construction does not guarantee that *every* pair of nodes is within distance d — hence the construction yields only an upper bound.

During the 60's, a number of researchers searched for (n,d,k) graphs, in particular for families of such graphs with a large value of n, for particular values of d, and k. Unfortunately, all of this work was aimed at the fault-free case. Below we discuss several such families and show how the diameter is affected by a single link failure occurring anywhere in the graph – the case t=1. As discussed in the previous section, the occurrence of more than one link failure is so unlikely that we need not consider it.

Let us consider a particularly interesting family of (n,d,k) graphs – – due to Akers[10]. For this family the key parameters will be as follows:

$$k = d - 1$$

$$n = C(2d - 1, d)$$

the fan-out for this class of graphs is relatively high, but at the benefit of relatively low distance. The following table gives the key parameters of the Akers graph for a few values of d, and compares n with the Moore bound.

d	k	n	n_{max}
2	1	3	3
3	2	10	10
4	3	35	53
5	4	136	426

Although these graphs do not come close to satisfying the Moore bound (except for small values of d), we will observe that the distance increases only slightly (if at all) when link failures are taken into account.

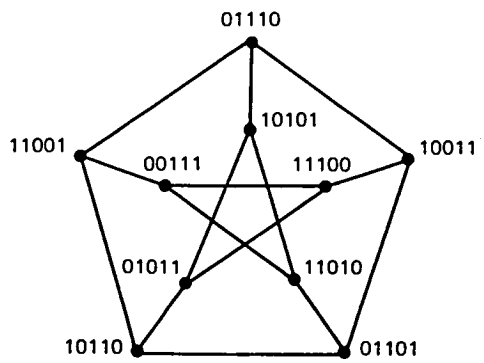


FIGURE 10 AN AKERS (n, d, k) GRAPH

Akers' construction technique is as follows. There are C(2d-1,d) binary words of length 2d-1 each of which have exactly d 1's. Associate a node of the graph with each of these words. For d = 3, there are C(5,3) = 10 such words of length

5, each of which have exactly three 1's. A branch is drawn between each pair of nodes whose corresponding words have exactly one 1 in a common position. The graph for d = 3 is shown in Figure 10.

We claim that this construction yields a diameter of k = d-1 — 2 for the case d = 3. The proof by construction given by Akers, is as follows:

> Consider any two nodes P_1 and P_2 that agree in q places and, hence, disagree in r = 2(d-1-q) places. It can be shown that r is even and q odd for all node pairs. When r < q, P_1 and P_2 are distance–r apart, limited by a sequence of r branches found as follows. Take in turn each of the r disagreeing places and complement the digits in the other 2d–2 places, assuring that the position chosen to be unchanged at each step contains a 1. Similarly, when q < r, P_1 and P_2 are distance–q apart. The process is to take each of the q agreeing places and complement the other 2d-2 places, assuring again that the position chosen to be unchanged contains a 1. The number of branches traversed is at most d-1.

Below we illustrate the construction for two nodes of Figure 10, distance–2 apart.

Positions	1	2	3	4	5
P_1	1	0	1	1	0
P_2	1	1	1	0	0

We observe that q = 3 (positions 1,3,5) and C = 2 (positions 2,4); hence we follow the step associated with the conditions C < q. We start with P_1 (although the process would yield the same path if we started with P_2), seeking the node P_i that will define the path of length 2. By the rule above, position 4 of P_i is to be left at 1; the other digits of P_i are complemented, yielding P_i = 01011.

Although the Akers' graphs were not necessarily designed in consideration of fault–tolerance, they have attractive diameter properties in the presence of single link failures – – and multiple failures. In particular, as proved below, the diameters of the graphs in the presence of single link failures is as follows; for

convenience the diameter for the case of no link failures is also shown:

d	$k_n f$ (no failure)	$k_s f$ (single failure)	n
2	1	2	3
3	2	4	10
4	3	5	35
5	4	5	136
6	5	5	462

For those graphs where the nodal degree d is greater than or equal to 6, there is no increase in diameter due to the presence of an arbitrary link failure.

The proof involves exhibiting a path that avoids any possible single link and whose length does not exceed $k_s f$. To carry out the proof, we require the following property, which we name the *alternate path property.*

Alternate Path Property (APP): In the Akers' graph, if two nodes p_1 and p_2 are at least distance-3 apart, then there are at least 2 disjoint distance-3 paths linking them.

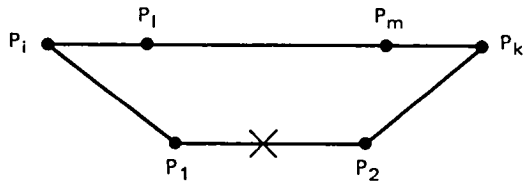
The proof of the APP follows from the construction outlined above for the fault-free case. If p_1 and p_2 are at least distance-3 apart and $r<q$, then there are at least two 1's in positions of p_1 for which the positions of p_2 contain 0's. Hence in finding there are 2 nodes, p_i, p_j, that are M neighbors of p_1 and on the shortest path between p_1 and p_2. (Similarly, if $q<r$, there are at least two 1's in positions of p_1 for which the positions of p_2 contain 1's.) The neighbor p_k of p_i (or p_j) next on the path to p_2 is shown to be unique. Hence for the case of 2 nodes *exactly* distance-3 apart, there are exactly two shortest paths, and they are disjoint.

It now follows that if p_1 and p_2 are at least distance-3 apart, an alternate path can be found to any primary path that avoids any (failed) link and that has the same distance as the primary path. What about the case of nodes that are distance-1 or distance-2 apart.

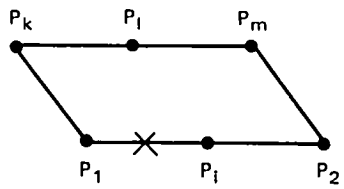
Let us first consider the case of p_1 and p_2 being distance-1 apart, by referring to Figure 11(a). Assume the link between p_1 and p_2 has failed, necessitating the location of an alternate path. Let p_1 and p_k ($k \neq 1$) be a neighbor of p_2. Clearly, p_i and p_k are distance-3 apart. One of these paths, of course, involves p_1 and p_2, while the other goes through two new nodes p_l and p_m. Hence the alternate path between p_1 and p_2 (p_1, p_i, p_l, p_m, p_k, p_2) has distance 5. Note that the graph of Figure 10, having diameter-2, requires special consideration. Since the diameter $k=2$, the neighbors of p_1 and p_2 (p_i and p_k) are distance-2 apart, yielding an alternate path at distance-4.

Figure 11(b) considers the case of a failure in a path at length 2. Here we take a neighbor p_k at p_i, and indicate two distance-3 paths from p_k to p−2, one of which contains the original distance-2 path as a subpath. Again, the graph of Figure 10 is a special case, yielding a path of distance-3 that is the alternative to a path of distance 2.

Figure 12 summarizes these two cases.

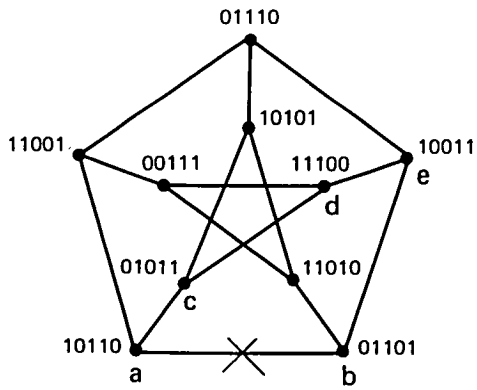


(a) DISTANCE-1 CASE



(b) DISTANCE-2 CASE

FIGURE 11 ALTERNATE PATHS IN AKERS GRAPH



LINK a-b IS FAULTY

1. ALTERNATE PATH BETWEEN a, b IS (a, c, d, e, b) AND HAS LENGTH 4.
2. ALTERNATE PATH BETWEEN c, b IS (c, d, e, b) AND HAS LENGTH 3.

FIGURE 12 AN AKERS GRAPH WITH A SINGLE LINK FAILURE

1.4 Real Time Identification of Shortest Path

In this section we consider the problem of identifying shortest paths in a network; in particular, we consider the question: should the shortest path be precomputed and stored for all possible link failures or should it be dynamically computed? In either case, we assume that once the shortest path has been determined, each node p_1 stores for each ultimate destination node p_2 the neighbor p_i of p_1 that is on *a* shortest path from p_1 to p_2.

Let us first consider the possibility of pre-storing all of the shortest paths, in consideration of all possible single-link failures. That is, at each node p_1 a set of triples $<p_2, p_i, p_l>$ is stored, indicating that the shortest path to p_2 is the presence of a fault in p_l is to involve p_i's neighbor p_l. Once a link has been determined to have failed, its identity is broadcast to all nodes in the network, each of which then applies the appropriate triples. Since the number of such triples at each node grows as the square of the number of nodes n in the network, the storage can quickly become excessive. For n=100, there are 10^4 triples; for n=200, there are 4.10^4.

Hence for large (n>75) networks, it is recommended that the shortest path be recomputed subsequent to a link failure and prior to continuing normal operation. This process is quite straightforward for the case at the Aker's graph, each cluster participating in the determination of alternate paths as follows. Assume nodes p_1 and p_2 define a failed link and that this link is on a particular shortest path. If there is an alternate path starting at p_1 that avoids p_2 and has the same length as the primary path, then the alternate path is selected. The Akers construction algorithm is used, in which case the alternate path from p_1 will be selected only if the distance from p_1 to the destination is at least 3. Otherwise, it will be necessary to seek the alternate path starting from some node preceding p_1. In this case, p_1 reports to its predecessor that it should seek the alternate path.

1.5 Synchronization in NETS

The successful operation of a SIFT cluster requires that its processors be synchronized to within approximately 50 microseconds. Such synchronization is necessary to prevent a processor from changing its rate of processing tasks to the point where it is working on an iteration that is different from its neighbors, and thus producing different results and destroying the exact match required for voting.

In NETS, of course, each SIFT cluster would have to be internally synchronized and that cluster must itself remain synchronized with other clusters, though possibly the permissible intercluster skew may be greater than the permissible intracluster skew. If each cluster contains at least 4 processors and if the fan-out from each cluster is 3, then the current SIFT synchronization algorithm can be used by each cluster to synchronize itself with its neighbors.

It should be noted that the algorithm can also be used for clusters containing fewer than 4 processors. In this case, each processor will synchronize itself with its neighbors within a cluster *and* in other clusters.

1.6 Recovery from Massive Transients

We assume the occurrence of a fault that corrupts the state of every processor in a cluster, but does not cause damage to prevent its subsequent processing. This kind of fault we call a *massive transient* fault. A cluster c suffering such a fault may not be able to effect recovery without outside assistance. For example, the information indicating working processors and, perhaps more fundamentally, indicating the neighbors of c might be have been lost due to the fault. The following are requirements to permit the recovery of a cluster through the assistance of its neighbors:

- Restart Box (rb)
- Checkpointing of global data
- Detection of Massive Transient
- Recovery process

We will require a modest-size special purpose circuit in each cluster, which we call a *restart box* (rb). A rb will accept inputs from each of c's neighbors, requesting c to return to a reset state. To prevent a neighbor, perhaps one that itself has suffered a massive transient, from maliciously trying to restart its neighbors, restart will only be carried out if at least 2 of c's neighbors send restart signals. The immediate effect of being in the reset state is to execute the clock synchronization algorithm and run an initialization program that will continue recovery (see below).

Certain critical data of a cluster must be checkpointed. This includes the identity of the working processors, the identity of neighbors (assuming such information is not hard-wired), the pairs and triples needed to communicate with other clusters on shortest paths, and the identity of which links with neighbors are working. Such information can be given to a neighbor each time it is updated. Not required to be checkpointed would be task data (it is regenerated each iteration) and task schedules (they are likely to be stored in microcode).

When a cluster has suffered a massive transient it is assumed that its behavior becomes erratic. This could involve the processors becoming unsynchronized, the loss of data such that there is little agreement among the output values of the cluster's processors, or the absence of any output data. Any of these events, when observed by a neighbor, are evidence of a massive transient. In any event, a cluster exhibiting such behavior cannot produce any useful work. It would also be possible for a neighbor to submit test data and, based on the return, decide that the cluster has suffered a massive transient.

The recovery of a cluster is as follows. It must be given all of the data it previously checkpointed and be moved to a state where it starts to execute the tasks on its schedule. The data will come from its neighbors, once the cluster indicates that it is in its reset state. The final input from the neighbors will move the cluster to the state where it commences doing useful work.

1.7 Remaining Problems and Recommendations

We believe that the work conducted to date demonstrates the feasibility of the NETS concept for the aircraft environment. Some additional problems, solution to which would optimize the NETS design are the following:

- Determination of Near-Optimal Interconnection Networks. Based on the work carried out in the 60's, adequate <n,d,k> graphs are known. However, the situation is not as encouraging when faults must be handled – the <n,d,k,t> case. We have shown that the Akers' graphs have good fault-handling capabilities – at least for the case of single link faults. It is recommended that other families of graphs be sought. Moreover, reasonably tight upper bounds on n should be determined to guide the search for such graphs.

- Assignment of Tasks to Clusters. The motivation for the search for <n,d,k,t> graphs was that tasks could be assigned to clusters in an arbitrary manner, hence the need for graphs with low diameter. However, it should be possible to take advantage of the structure inherent in task communication to determine optimal assignments of tasks to clusters. The problem is as follows. Assume that the computations to be carried out are expressed as graphs, the nodes of which are tasks and the edges indicate communication between tasks. For a set of such computations, find an optimal embedding onto the underlying network graph. It is not obvious just what constitutes optimality, but minimizing the longest communication path seems to be a good choice. It is noted that this problem is related to the VLSI placement problem, although our problem does not have the rectilinear structure of the VLSI problem.

- Effects of Combinations of Failures. Our design effort so far has assumed that faults are handled shortly after their occurrence. We have avoided considering the recovery from multiple failures, e.g., a massive transient at the same time as a link failure. Some effort should be given to this more general case.

It is recommended that work continue on NETS to further refine and optimize its design. Furthermore, we recommend that experiments be conducted on the

AirLab facility to verify some of our "conjectures" and to determine if some of the remaining problems admit to experimental solutions. It should be relatively easy to use AirLab VAX's to emulate NETS. Each VAX could be a cluster. The interconnection network could be overlayed on the VAX local area net in an easy way. Any direct connection of clusters A and B would involve the direct transfer of data from A to B via the net. Where an intermediate hop is required, two transmissions via the VAX net would be effected, and so on. Among the important properties to be monitored are the following:

- Transport delay
- Communication load, searching for any imbalances among the links
- Time for recovery from massive transients
- Time to compute shortest paths following identification of link failures
- Ability of the system to handle multiple errors.

Chapter 2

The Analysis of Transient Faults

Introduction

Faults in computer systems are of two kinds, solid and transient. A solid fault is one in which some component of the system fails and will continue to fail for all subsequent uses. A transient fault is one in which some component of the system is temporarily deranged and fails in use, but in which that component subsequently recovers, without repair action, and in subsequent use the component does not fail.

Transient faults may be caused by thermal noise in a marginal component, by cosmic rays or alpha rays, or by electromagnetic interference. For typical transient faults, the faulty component is deranged for only microseconds or at most milliseconds, though the errors resulting from the fault may persist for much longer. It is difficult to obtain dependable information on the frequency of transient faults under operational conditions, because current systems are not instrumented to distinguish between solid and transient faults. However such information as is available indicated that transient faults will occur more frequently than solid faults, at perhaps ten times the rate.

It is important to distinguish between solid and transient faults, since processors suffering from a solid fault are removed from the system configuration. In contrast, processors subject to a transient fault are permitted to remain in the configuration. In other designs, the initial action taken for both solid and transient faults is the same – the processor is temporarily removed from the configuration pending tests to determine whether it is working and can be readmitted. This approach is not used in SIFT because:

- Processors diagnosed as having a solid fault are never readmitted to the configuration after they have been removed, even if the off-line diagnostic tests cannot detect any fault. The coverage of the diagnostic tests is not high enough to ensure that the benefit from readmitting good processors to the configuration outweighs the loss in reliability from readmitting defective processors.

- Processors diagnosed as suffering from a transient fault are not removed from the configuration, even temporarily, since the short duration of transient faults ensures that the actual faulty condition will not last even as long as the time required for error recognition and reconfiguration.

The reliability modelling results of the SIFT project [1] analysed the ability to distinguish between solid and transient faults. Assuming that transient faults are substantially more frequent than solid faults, it is important for the error diagnosis of the system to be able to recognize transient faults.

- If transient faults are incorrectly diagnosed as solid, resulting in working processors being deleted from the system configuration, the rate of system failure due to exhaustion of spares is greatly increased.

- If a solid fault is incorrectly diagnosed as a transient, the effects on system reliability are much less deleterious. The solid fault will generate further errors and provide further opportunities for repeating the diagnosis and recognizing the solid nature of the fault. The system is at risk to the occurrence of a second fault, whether transient or solid, during the time interval before the fault is correctly diagnosed.

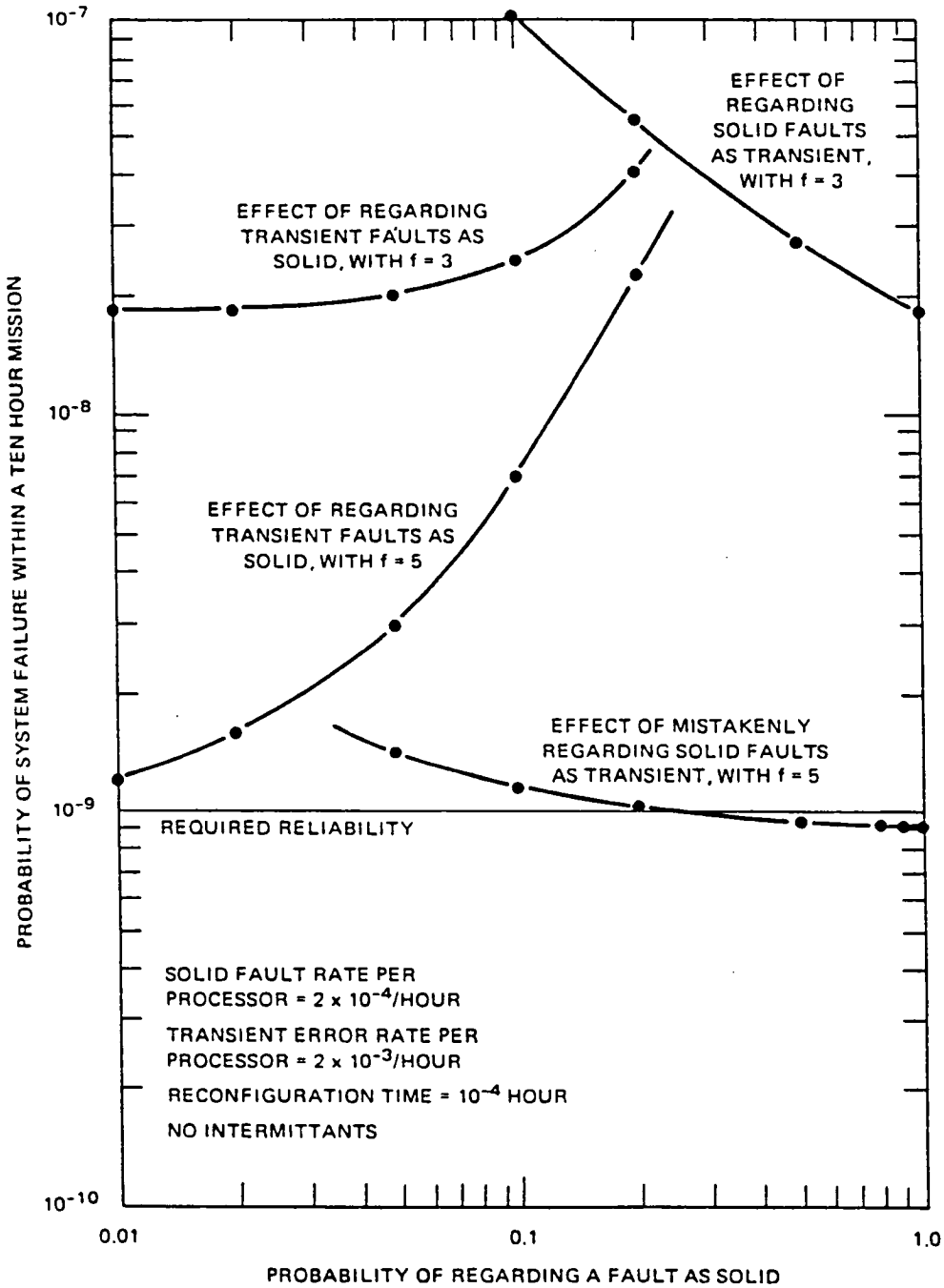


Figure 13. The Effect of Discrimination between Solid and Transient Faults

Figure 12 contains results obtained from the reliability model for SIFT, showing the probability of system failure within a 10 hour flight*. It is evident, particularly where critical functions are protected by five-fold voting (f=5), that a relatively small probability of regarding a transient fault as solid has a much bigger effect on the probability of system failure than does the corresponding probability of regarding a solid fault as transient. But, of course, it is essential to recognize solid faults; regarding all solid faults as transient is devastating to the reliability of the system.

Consequently, the ability of the system's error diagnosis routines to distinguish between solid and transient faults is very important.

2.1 Solid Fault Types

Many solid faults are catastrophic and either prevent the computer from generating any results at all or cause almost all results generated to be erroneous. However some faults, though solid, produce erroneous results only in rather specific circumstances. Such faults generate periodic errors, produced more or less frequently whenever those specific circumstances occur. A solid fault that generates errors only infrequently can be difficult to distinguish from a succession of transient faults.

Some faults will never yield an erroneous result, for the particular components are never actually used to produce the results in question. Other faults yield errors, not on every execution of the task, but only for specific data values, resulting in errors every few milliseconds, or seconds, or minutes. Experiments have been performed at NASA Langley Research Center to investigate the proportion of solid faults that do not generate immediate errors in the results of ap-

*It is important to note that these results are based on plausible but arbitrary component failure rates. Consequently the results can only be of qualitative significance. Quantitative measures of the reliability must be derived from careful measurement of actual component failure rates under operational conditions.

plication programs. Unfortunately these experiments have, for obvious economic reasons, considered only the proportion of faults that result, or do not result, in an error within a relatively short time interval. It is important to extend this work to determine the shape of the tail of the error-generation-frequency distribution, and it is to be hoped that the team at NASA Langley might consider such an experiment for AIRLAB.

During the period between the time when a solid fault occurs and the time when the fault causes an error, the fault is "latent". Latent faults are of course undetectable. The duration of latency of the fault is not significant, and latent faults are no more damaging to system reliability than simple faults, provided that the fault is "uncorrelated". While the latent fault remains undetectable so long as it is latent, it can also do no damage so long as it is latent. Only when the fault generates an error is there any risk to the system, and the duration of the previous period of latency is of no significance, provided that the error is generated at a random moment in time.

Correlated latent errors present a significant risk to the reliability of the system. A correlated latent error remains latent until some other error also occurs, and thus is manifested only in a double error situation. There are two ways in which this can occur:

- The latent fault can be such that the only circumstances in which errors are generated are those in which other errors are already present. Such a fault might damage only the operation of the error detection, diagnosis, or reconfiguration.
- The latent fault can be such that errors are generated only during a specific infrequently performed, but critical, function (for instance the autolanding functions). There is a risk that two processors might each be affected by such a latent fault, undetectable until the function is invoked and then yielding a double error.

This analysis, and indeed the whole SIFT design, does not address correlated faults.

There can also exist faults that are solid in that their defect is due to a physical cause that is permanent, but which generate errors only infrequently due to some physical aspect of the nature of the fault, rather than due to the nature of the processing being performed. Such faults are referred to as "intermittent" and are sometimes caused by cracks in conductors or by loose particles in packages. An intermittent fault resembles a succession of transient faults. The duration of each error generating event of an intermittent fault is usually rather longer than for a transient fault, and the frequency of such event is usually much greater than the frequency of transient faults in a properly working processor. The reliability analysis for SIFT indicated that the transient fault analysis algorithms are well able to protect the system against intermittent faults.

2.2 Error Generation and Detection

Faults, whether solid or transient, are manifested only through the errors that they generate, whether those errors are in the results of the application tasks or errors in the results of a diagnostic test sequence. This immediate error is of course only an incorrect result. To act on the error requires that it be detected, that a checking mechanism be capable of recognizing that the result is indeed incorrect and thus that an error, and by implication a fault, exists. Once the error is detected, it must be diagnosed that a specific type of fault is the cause of the error, and that some recovery or reconfiguration action is appropriate.

In typical low reliability systems, error detection is very poor and the degree of confidence that any particular erroneous result will be noticed is low. High reliability systems, such as SIFT, in contrast have very good error detection and almost any error will be detected.

Even though a fault may cause errors to be generated 'immediately', the errors are not detected, and thus the existence of the fault is not recognized, until the erroneous results are subjected to the voting or other error detection checks. The results of high priority tasks are needed for use by other tasks within a short

period of time, and thus must be voted or checked very promptly, certainly within a few milliseconds. Thus, a fault that causes errors in the results of high priority tasks can be detected soon after the fault occurs.

However, many systems contain background tasks whose results are not needed immediately. The execution of such tasks may be spread over several seconds, or even longer, and the results may not be voted until some convenient moment long after they were generated. A fault that yields and error in the results of a background task may not be detected until seconds, or even minutes, after the fault occurs.

This has two effects:

- During the interval between the generation of the erroneous result for the background task and the masking of that error by voting, the system is vulnerable to the occurrence of a second fault. Fortunately, background tasks are usually not very critical and a rather higher risk of failure of such a task can be accepted. Results that are very critical must be voted at frequent intervals to ensure that errors are masked promptly, thus reducing the risk of error accumulation between masking. This frequent voting of critical results is necessary even if processing of those results is required only infrequently.
- A single transient error may occur and damage the results of several tasks. The erroneous results of high priority tasks will be detected quickly, but further error reports will continue to be generated for some time as other results of lower priority are voted. This might confuse the error diagnosis routines into thinking that the error that has occurred is persisting in generating errors, and thus may be solid. It might also confuse the error diagnosis routines into thinking that multiple faults had occurred.

It might be hoped that the nature or appearance of the error detected might provide an indication as to the location and type of the fault that caused it. Unfortunately, the errors detected are often of the form of an incorrect result and it is difficult to ascribe a cause from such meager information. Further, a "malicious" fault may masquerade as some different type of fault. It is es-

sential that such deception should not permit the successive removal from the configuration of working equipment until system failure results.

Any one error report originates at a single point in the system and the report must be replicated for analysis by the necessarily replicated global executive routines. As for any other information that originates at a single point, interactive consistency or interactive convergence techniques must be used to ensure that the replications are consistent. Even when a component reports itself to be faulty, it is essential to use interactive consistency techniques to detect that a processor has reported itself faulty to one neighbour and not to others, and situation indistinguishable from that in which the neighbour falsely claims that the processor has reported itself faulty.

2.3 The Analysis of Error Reports

An error report is certain information that a fault has occurred, but less certain as to what fault and when. If interactive consistency techniques are used, a report by processor A of an error in the results of processor B for iteration i of task k provides the information that:

- the fault existed in either processor A, or processor B, or the link between them
- the fault existed at some time since the start of the data-window for iteration i of task k

The global executive routines must make use of the combination of many error reports to deduce the true nature of the underlying fault. The basic algorithms used in SIFT are described in [1]. We discuss here three aspects of fault diagnosis:

- identification of, and action on, link failure,
- identification of transient faults,
- identification of low error rate solid faults.

2.3.1 Identification of Link Failure

When a processor fails, it will probably generate erroneous results and broadcast them to all of the other processors, resulting in a large number of error reports from which it is easy to diagnose which processor has failed.

Less probably, a processor might suffer from a fault that causes it to generate erroneous error reports even though the results being voted were correct. If that processor is detected by the global executive to be generating many error reports, claiming errors in several other processors, all unsupported by reports from other processors, the diagnosis is again relatively easy.

But failure of the link between two processors results in error reports in which one processor systematically reports errors in the results of just one other processor, without any corroboration from other processors. The exact location of the fault may be:

- in the physical link itself,
- in the transmitting circuitry of the broadcasting processor, after the point at which the common broadcast signal has fanned out into separate signals for each destination,
- in the receiving circuitry, or the result buffering, or the voting software, or the error reporting software, of the processor reporting the error.

Because continued operation of SIFT requires full connectivity between all processors of the configuration, and because continued operation with a faulty link exposes the system to failure should another fault occur, it is essential to reconfigure the system to a reduced configuration in which the faulty link is not required, i.e. to a configuration without one or other of the two processors at either end of the link.

If the fault is simple, it matters little which of the two processors is to be reconfigured out of the system. But it is very important that a malicious fault should not be able to exploit the choice to remove systematically a succession of

other processors.

The algorithm recommended is:

following a link failure event in which processor B reports errors in the results of processor A, without corroboration,

- if processor A is not on probation then:
  - processor B is removed from the configuration,
  - processor A is recorded as being on probation,
- if processor A is on probation then processor A is removed from the configuration.

The choice is made to favor removing processor B, rather than processor A, from the configuration because there is very little logic in processor A after the fanout point at which the common broadcast signal is split into separate signals for each destination. Consequently is is relatively improbable, though not impossible, for a malicious fault to develop in that small amount of logic within processor A. In contrast, the amount of logic, both hardware and software, in processor B that is capable of producing the symptoms is quite large.

However, the algorithm must guard against the possibility of a malicious fault in the small amount of logic in processor A. Consequently, processor A is placed on probation. Thus, if a malicious fault in processor A should succeed in causing processor B to be removed from the configuration, any subsequent attempt by processor A to repeat the attempt, say on processor C, results in the removal of A rather than C. Consequently, the rather improbable malicious fault in processor A can cause two processors to be lost, but no more than two.

2.3.2 Identification of Transient Faults

The identification of transient faults is based on their short duration. It is assumed that a fault is solid if it persists, i.e. continues to generate errors, for more than some period of time (known as the *solid/transient discrimination*

interval). Typically that period of time might be set to say 200ms or 300ms, for almost all transient events are much shorter. Faults that generate only single isolated errors, or short bursts of errors, are assumed to be transient.

In systems in which all tasks operate at iteration rates shorter than the solid/transient discrimination interval, such as SIFT Mk I, it is relatively easy to distinguish solid from transient faults. Such system detect all errors within one iteration, and any fault that causes errors to be detected in two or more iterations can be assumed to be solid.

Future systems will contain tasks that operate at very different iteration rates, and some of those iteration rates will be much longer than the solid/transient discrimination interval. The detection of errors in the results of slowly iterating tasks may be delayed for a period comparable to the iteration interval of the task. For certain navigation and fuel management tasks, this delay may be many seconds or even minutes. Thus, even for a transient fault of short duration, if the results of lower priority tasks have been affected then error reports may be generated periodically over a relatively long interval of time. Consequently it does not suffice to assume that a solid fault is indicated by error reports spread over an interval longer than the solid/transient discrimination interval.

The proposed algorithm is based on the concept of *fault windows*, the interval of time somewhere within which a fault must have existed to cause the observed error symptoms. We will consider two types of fault windows:

- error report windows,
- fault event windows.

An error report window is the interval within which a fault must have existed to result in the observed error report. The error report window, for an erroneous result from task A, extends from the earliest time of voting of any input value to task A until the completion of interactive consistency on the error report.

A fault event window is the interval within which a fault must have existed to cause several error reports, and thus is the intersection of the of the error report

windows for each of the error reports.

The proposed algorithm for discrimination between solid and transient faults is:

- initially the fault event window is set to empty.
- when a fault report is received,
  - the error report window for that report is computed,
  - if the fault event window is empty then an new fault event window is created equal to the current error report window,
  - if the fault event window is not empty, but the intersection between the fault event window and the error report window is empty, then again a new fault even window is created equal to the current error report window,
  - if the intersection of the fault event window and the error report window is not empty, the fault event window is set to that intersection.
- if the end of the fault event window is so long ago that no current fault could generate an error report window to intersect it, the fault event window can be set to empty.
- every time that a new fault event window is created, analysis is made of the frequency of fault event to determine whether reconfiguration action is required.

The behavior of this algorithm is illustrated in Figures 13 - 16. In Figure 13, the fault event window is initially empty. Thus the error report window is computed and a new fault event window is created and set equal to the error report window.

In Figure 14, a further error report has been received whose window overlaps the fault event window. We assume that the same fault generated both error reports. Thus the fault event window is reduced in size to the intersection of the two windows. In Figure 15, it is still possible that the same fault caused all three errors and thus the fault event window is again reduced in size.

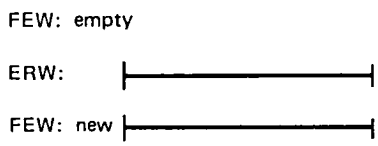


Figure 14. An Error is Reported when the Fault Event Window is Empty

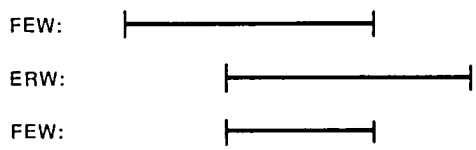


Figure 15. An Error whose Window overlaps the Fault Event Window

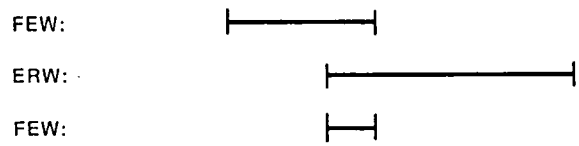


Figure 16. A further Error whose Window overlaps the Fault Event Window

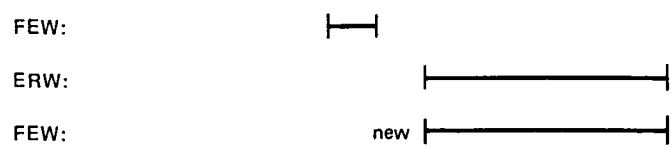


Figure 17. An Error whose Window does not Intersect the Fault Event Window

But in Figure 16, the next error report window no longer intersects the fault event window, and it is not possible for a single transient fault of short duration to have caused all of the errors that have been reported. Thus we create a new fault event window, equal to the error report window, and start the analysis to determine whether the frequency of faults requires reconfiguration (see the section below).

2.3.3 Identification of Low Error Rate Solid Faults

The discrimination between solid and transient faults depends on the observation that a transient fault is of short duration, and thus on the assumption that a set of errors, generated in some short interval and not followed by other errors, have probably been generated by a transient fault. But some solid faults are such that only occasional results are damaged by their presence. The equipment is definitely broken, but the nature of the fault is such that many correct results can be generated and only a few are erroneous. Unfortunately, it is unlikely that in service it will be possible to diagnose faults sufficiently to distinguish solid faults that generate errors only occasionally from transient faults. Consequently we do not distinguish between transient faults and low rate solid faults, but rather aim to determine whether the rate of occurrence of such faults is such as to damage the overall system's reliability.

If a processor suffers from a solid fault (or a transient fault) generates errors only occasionally, we must consider whether retaining that processor in the configuration improves the reliability of the system or reduces it. Occasional erroneous results from the processor$_i$ can be masked by the voting algorithms, but retaining the processor in the configuration increases the risk that its erroneous result will coincide with some other erroneous result, causing system failure. Removing the processor from the configuration eliminates the risk of coincident errors, but increases the risk of exhaustion of spares should several other processors fail.

A preliminary analysis of this problem was performed using the reliability model for SIFT. It is important to note that the results here are only indicative, and that the modelling should be repeated with more accurate data. The model was used to compare the reliability of two SIFT configurations:

- A five processor SIFT, with four normal processors and one processor set to generate occasional (transient) errors,
- A four processor SIFT, with all normal processors.

A normal SIFT processor was assumed to have a solid fault rate of 2×10^{-4}/hour and a transient fault rate of 2×10^{-3}/hour. The error rate of the 'special' processor was varied to investigate the effects of a higher than normal transient event rate.

For a SIFT system in which only three way voting is performed on critical functions, it was found that the 4 processor system became more reliable if the transient rate of the 'special' processor exceeded 10^{-1}/hour. In effect, a processor that suffers even a single fault per fight damages the reliability of the system, a rate that is only slightly greater than the expected transient rate for normal SIFT processors.

For a SIFT system in which critical functions are five way voted, it was found that the 5 processor system, containing the 'special' processor, remained more reliable even for special transient rates as high as one per few seconds. When ample error masking was available, the risk of coincident errors was not significant.

This investigation should be extended. For instance, no investigation was made of the effect of remaining mission duration of the discrimination, nor was consideration given to situations in which more than one processor has a high transient fault rate. The analysis should also be performed for systems with initial numbers of processors other than 5.

Chapter 3

Novel Fault-Tolerant Architectures

Introduction

This task involved the study of novel, unconventional architectures from the point of view of fault-tolerance. In particular, the relevance of dataflow architecture was studied. The main conclusion reached was that fault-tolerance can be added to a dataflow architecture more simply and for less cost than it can be added to a conventional Von Neumann, imperative, sequential architecture. In fact, it is worthwhile implementing conventional programs in a dataflow manner, in order to then add fault-tolerance capabilities. Since in this case the use of dataflow is not motivated, as it usually is, by a desire to use parallel computation to lower program execution times, it is not out of the question to consider translating conventional programs into a dataflow language (compilation), or interpreting conventional programs using an interpreter written in a dataflow language. In either case the fault-tolerance capabilities could be added in the way to be described in this report. Nevertheless, if the programs were actually written in a dataflow language, using a dataflow machine would give decreases in execution

time (increased efficiency) that would more than compensate for the increased resources (extra processors) that are needed for fault-tolerance.

In order to simplify the later discussion, and clarify the meanings of terms to be used later, it is essential to give a short introduction to the concepts of dataflow.

3.1 Dataflow

It is useful to distinguish between two different ways of looking at dataflow. One way is to consider dataflow as a programming methodology. The other is to consider dataflow as a way of using multiprocessor machines. The use of dataflow as a programming methodology has been around for some time and has been very successful in applications like data processing (see, for example, the Transform Analysis of Yourdon and Constantine, even though the final programs produced tend to be in FORTRAN or COBOL and run on conventional machines. (This is not a requirement of the methodology. In fact, the transition from dataflow to conventional programs is currently just a tedious, error-prone, but necessary, final stage in the methodology. When dataflow machines become available, it will, presumably, be dropped.) Here dataflow will be considered at a more detailed level, in terms of dataflow networks and the implementation of such networks by dataflow machines.

The basic idea in dataflow is that computations proceed not by flow of control around a flowchart but by flow of data around a dataflow network. A dataflow network is a directed graph whose nodes represent operations or (user-defined) functions to be performed on the data that come to the node along the edges that terminate at the nodes. If the operation has several arguments, there should be that number of edges terminating at the node, one per argument. The data resulting from an operation leave the node along the single edge that originates at the node.

There are three non-operation nodes. One such node is the *split* node with one incoming edge and an arbitrary number of outgoing edges, which simply splits its incoming data items into several copies, sending one on each of the outgoing edges. (Actually it is sufficient to only have split nodes with two outgoing edges, and cascade them to get more than two copies.) There is also a merging node, in which several edges converge on a single node, but do not correspond to the different arguments of a polyadic operation. This is the *select* node, which has three incoming edges, along one of which boolean data items will arrive at the node, and on the basis of which the items arriving along the other two edges are merged together into the stream of values that leave the node along its single outgoing edge. There is also the *distribute* or *switch* node, the inverse of the *select* node, having two incoming edges, one of which is for boolean data, and two outgoing edges. The boolean input determines along which of the two outputs the other input is to be sent.

There are several existing projects to build or design dataflow machines, but they all differ in their approach to how dataflow network computations proceed. (All of them agree, however, that computation should proceed in parallel, with various operations being performed at the same time, asynchronously.)

3.1.1 Data-Driven vs. Demand-Driven Computation

There are different views as to the way computations are *driven*. Most projects adopt the data-driven approach, which says that the operation of a node will be performed as soon as there is a data item on each of the incoming edges of the node. (In fact, most researchers assume that this is the distinguishing feature of dataflow, whereas we feel that the term "dataflow" encompasses more than this.) On the other hand, the demand-driven approach says that the operation of a node will be performed only if, as before, there are data items on all the incoming edges and, in addition, there is a demand for the result of the operation. If there is a demand but there is an incoming edge of the node that does not have a data item on it, then a demand for a data item is made of the node at the

beginning of that edge. When, as a result of such demands, there is a data item on each incoming edge, the operation of the node is performed (consuming the inputs), and its result is sent out along the node's output edge.

The demand-driven approach is out of favor for two reasons. Firstly, just in propagation of demands, there is extra computation that needs to be performed before any operations actually get executed. Secondly, it seems that the amount of parallel activity is less than would be produced by the data-driven approach, because operations are only executed after it has been determined that their results are definitely needed. It appears that there must be a sequential *ebb and flow* effect. Demands are propagated, from the edges leading out of the network, back through the dataflow network until they reach the nodes representing constants (nullary operations) or edges leading into the network from outside (along which inputs arrive). This is followed by a wave of computed values that go roaring forwards through the network until data items finally come out along the edges that lead out of the network. This is then followed by another wave of demands in the backwards direction (resulting from demands for the next output values), and so on. This means that the operations that eventually lead to an output value cannot be performed until all the computations that will lead to the previous output value have been completed. This contrasts sharply with data-driven computation, where these different computations can be proceeding simultaneously, in different parts of the network.

This *ebb and flow* effect is actually caused by not demanding an output until the previous output has been produced, and would disappear if demands for output were made spontaneously, that is, if computations were made to result from an endless stream of demands for output being "initially" injected into the network. The rate at which these demands for output are injected into the network could be crucial in terms of getting as much parallelism from the demand-driven approach as can be obtained from the data-driven approach.

In any case, it is not really obvious that the demand-driven approach is less efficient. The computational savings to be made, by only performing computations that are definitely necessary, may, when the number of processors is bounded

(as of course it will be in practice), result in lower execution times for demand-driven computation than for data-driven computation. There are other reasons for using the demand-driven approach. For example, the implementation of certain languages seems to require it, and the computational behavior is more predictable, possibly allowing the use of dataflow in real-time applications.

3.1.2 Pipelined Dataflow vs. Tagged Dataflow

There is another way in which the various dataflow machine projects differ. Some use *pipelined* (*static*) *dataflow* and some use *tagged* (*dynamic*) *dataflow*.

In pipelined dataflow, the edges in a dataflow network are considered to be pipes along which data items flow, in a first-in, first-out manner; in other words, there are queues acting as buffers between the nodes (some projects require these queues to be of bounded length).

In tagged dataflow, the edges in a dataflow network simply indicate the routes that data items must take. The buffers between the nodes are not queues, they are unordered sets. Some discipline must be imposed on the order in which the operation of a node takes the data items from the incoming edges, and this is achieved by associating the data items with *tags*, so that the operation looks for data items with appropriate tags. Often the tag simply indicates the order in which the data items are added to the set. If the operation then can only take data items from the set in increasing order of tag, and the data items produced have the same tag as the data items consumed, tagged dataflow simply simulates pipelined dataflow. On the other hand, if we allow the operation to be applied simultaneously to many data items, and the operation terminates sooner for some inputs than others, the tag will no longer correspond to the order in which the data items are added to a set (we have assumed that the tags on data items produced are the same as those on the data items consumed, and the time of production is unpredictable). There are definite reasons for wishing to allow such simultaneous application of operations (for example, to get increased parallelism, if dataflow

is being used as a way of getting increased computational power), so we must abandon the idea that the tag indicates the order in which items are added to a set. Rather, the tag indicates the position of the data item in the *conceptual* sequence of items emitted by a node. Thus it is still useful to think of sequences of data items, but we needn't think that a node must consume the data items in order, or produce them in order.

Tagged dataflow was invented to allow increased parallelism, but it has many other useful applications, particularly in the dataflow implementation of higher-level languages. For example, demand-driven tagged dataflow allows an elegant way of dealing with conditional evaluation. After all, *if-then-else* is just a ternary basic operation, which happens to be nonstrict, and such operations are automatically handled when using demand-driven tagged dataflow. If the i-th value of *if P then A else B fi* is demanded, then a demand is generated for the i-th value of *P*. If the value produced is *true*, then a demand is generated for the i-th value of *A*; if the value produced is *false* then a demand is generated for the i-th value of *B*. In either case, the value subsequently generated is the desired i-th value of the expression. The conventional, data-driven way of dealing with conditional evaluation involves the use of *distribute* and *select* nodes. (In fact, *select* nodes are not needed with tagged dataflow, because the tags do the appropriate selection.) This is one of the few uses for such nodes. The other main use of *distribute* nodes is to extract the final values from a loop.

3.1.3 User-Defined Functions

It is possible to let some nodes in a dataflow network represent user-defined functions rather than operations. The definitions of user-defined functions can be thought of as pieces of dataflow network. How these pieces of network are used depends on whether the network is being evaluated in demand-driven or data-driven mode. If it is data-driven, when data items first arrive at all the edges leading into a function node, the node itself is replaced by the piece of network that corresponds to the function definition (this piece of network must have the

appropriate number of incoming edges to match the number of arguments of the defined function, and a single outgoing edge), and the data items (the arguments of the function) move on, into the new piece of network (the body of the function definition). (This corresponds to call-by-value parameter passing - the arguments of the function are evaluated before the function is called.)

If it is demand-driven, when a value of the function is first demanded, the node is replaced by the defining piece of network and the demand travels up along its single outgoing edge. Eventually this demand will result in demands being sent out along the incoming edges of the piece of network, to get the arguments of the function-call. (This corresponds to call-by-need parameter passing - the arguments of the function are only evaluated when needed.)

Recursively defined functions naturally give rise to networks that grow larger and larger, as more and more recursive calls are made. This changing of the network is essentially "graph reduction".

The above explanation really only applies to pipelined dataflow. In tagged dataflow, the network doesn't actually grow, but rather the tags on data items get more complicated. For example, the tag for a particular data item that corresponds to a local variable or formal parameter of a function call would indicate that the call was reached by some particular calling sequence, i.e. by a particular sequence of places in the program where function calls were made that eventually led to the current function call.

3.2 Dataflow Machines

A dataflow machine is an actual piece of hardware that "runs" dataflow networks. We have talked of the various approaches to network evaluation, and mentioned that evaluation proceeds in parallel, asynchronously. All this activity could be simulated on a single-processor machine, if we were using dataflow simply as a programming methodology. A true dataflow machine, however, should actually employ asynchronous parallel computation, using many processors, and

be a real "supercomputer". The independence of the computational activities of the various parts of a dataflow network, produced mainly by the lack of side effects of the operations, means that such dataflow machines are actually feasible.

As previously mentioned, there are several projects that are currently constructing dataflow machines. They are all experimenting with different architectures, mainly because of the different approaches to evaluation of networks: pipelined dataflow versus tagged dataflow, data-driven versus demand-driven, and bounded queues versus unbounded queues. The designers of the machines were left with the problem of programming them; because it was clear that the programs could not be actual networks, some linear textual representation was needed. The languages they came up with tended to be conventional functional languages or single-assignment languages, with the addition of some sort of iteration construct. There is one dataflow language that is different from the others, called *Lucid.* The other languages do not have the idea of infinite sequences built into the semantics of the language, as Lucid does, and are therefore less intimately linked with a main idea of dataflow, the idea of continually processing unbounded streams of input data. Although it is not essential to use Lucid to get the benefits of fault-tolerant dataflow, the language has some features that make it attractive, in its own right, for the sort of applications that underlie this project, namely aircraft flight control. This language makes certain requirements on dataflow implementations, namely that the dataflow be tagged and demand-driven. We will assume that this is the computational mode we are given when we consider the ways of adding fault-tolerance to dataflow. (Later we will consider how fault-tolerance could be added if the computational mode were otherwise.) Firstly, however, we will discuss the main features of tagged, demand-driven dataflow machines. The important features of Lucid, as far as dataflow and this project are concerned, will be discussed in a later section.

The architecture of a tagged, demand-driven dataflow machine might be as shown in Figure 17. The machine consists of two unidirectional rings, around which demand packets and term-value packets flow. Both rings pass through a "pool" of processors, where processors are acquired to execute particular basic

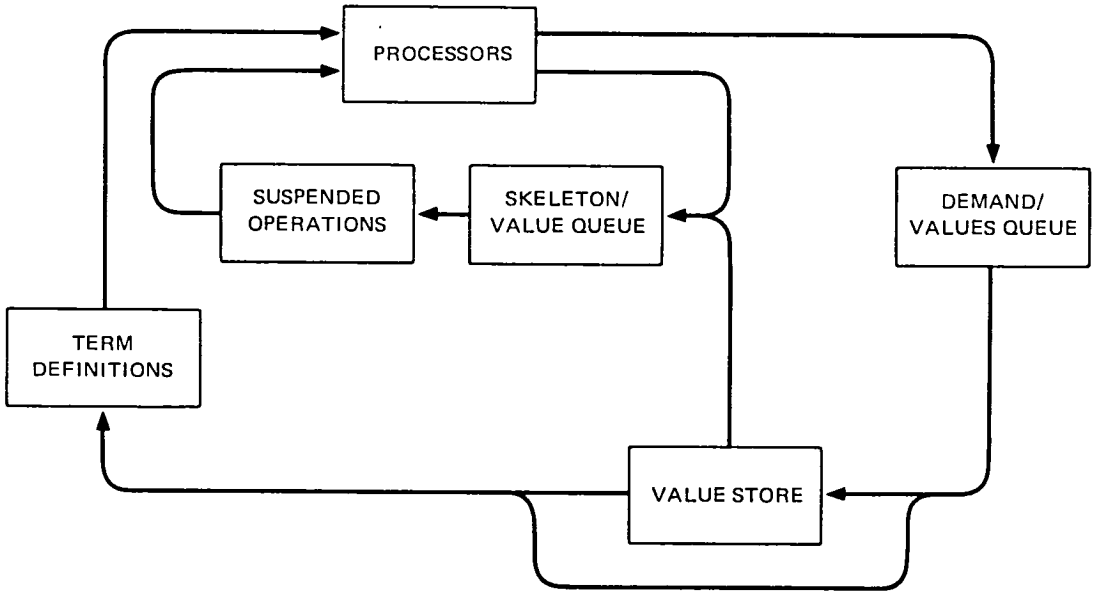


Figure 18. The Architecture of a Dataflow Machine.

operations of the language, to perform certain tag manipulations, and to translate demands, for the tagged values of basic operations applied to arguments, into "suspended operations" and demands for the tagged values of the operands. (When the tagged values of the operands are produced, they are put into the suspended operation and held there until the operation has a full complement of operand values, at which time this completed suspended operation travels to the processor pool to get a processor to execute the operation.) The important thing to realize is that the computations proceed in a very piecemeal way, with processors being picked up to perform very small computations, no bigger than a single execution of a basic operation. Moreover, when the processor is picked up the operands of the operation have already been evaluated, which means that the processor is kept for only a few microseconds.

3.2.1 Fault-tolerance

It is not really appropriate in this report to go into the detailed behavior of dataflow machines. The last paragraph explained enough of the operation of a machine to make the modification for fault-tolerance easily understood. The idea is simply this: when a processor has to be picked up, to execute a basic operation, modify a tag, produce new demands, etc., *two* processors should be picked up, rather than one, and both should be set to work doing the same calculation. In a few microseconds both processors should be finished. (If either of them is not finished within a reasonable time, it can be assumed that the processor has failed, and the corresponding result will be taken to be some failure flag.) The two results can be compared by a third, distinct, processor. If they both give the same result, this result is sent off around the appropriate ring. If they disagree, a fourth, necessarily reliable, processor should be immediately grabbed and set to work on the same minute computation. In a few microseconds it, too, will have finished, and its result is the result which is sent around the rings. (This result can also be used to determine which of the two original processors was in error (or even if the third, arbitration processor falsely indicated that the first two processors disagreed). The errant processor can then be immediately removed from the pool of processors.) The computation can then proceed with a delay of a few microseconds, well within the limits imposed by the fault-detection requirements.

One of the advantages of this technique, over the "conventional" fault-detection technique for Von Neumann machines, is that only double redundancy, not triple, is required. (We are not counting the arbitration processor because it does need not have all the capabilities of the other processors; all it does is check whether two bit-strings are identical.) Only when a fault is detected, by two processors disagreeing, is it necessary to call for a fourth, distinct, processor. (This should happen rarely.) It is possible to do this because of the way that dataflow machines naturally break their computations into minute pieces, and because there are no side effects of evaluating these pieces. The only result of such an evaluation is a single value, which can easily be compared with the result of a duplicate evaluation. If the results disagree, no effects will have occurred to

prevent a third processor from performing a fresh evaluation, from the original data.

Another advantage is that dataflow machines naturally have several processors, and if one processor becomes defective and has to be taken out of service, the machine will be essentially unchanged, and be ready to detect more errors. Only when a large fraction of the total number of processors has been removed from service will the machine become too slow to be able to function effectively.

The dataflow machine consists of more than just the pool of processors; it has several other components, and data packages will be transmitted around the rings in great numbers. Can not errors occur in the rest of the machine? The answer, of course, is yes, but these errors will be different in kind than the errors produced by defective processors. These errors will be data transmission errors, that can be detected, and corrected, by conventional single error correction codes. The special-purpose processors needed to do this error detection and correction can be built into the hardware of the machine, with enough redundancy to be immune to faults. (There has to be some level at which possible faults are disregarded. For example, in this discussion we are assuming that the communication channels between components of the machine are not going to be broken.)

For this to be a viable fault-tolerance technique in the context of applications such as those underlying this project, the dataflow machine, when running without any processor failures, must be able to perform real-time computations, where the machine has to respond to inputs within certain precise timing constraints. This will depend crucially on the language used to program the machine, on the way programs are written, and on the degree to which the time behavior of the machine can be determined by the programs. There is not a large number of dataflow languages to choose from, but the language Lucid appears to be the best language to choose. We will return to these questions after a discussion of the language Lucid.

3.3 Lucid

Although it looks imperative at first sight, Lucid is actually a functional language. Therefore, as with all functional languages, it is a *value* language: it is concerned with the values of variables. Normally this is as opposed to an *object* language, which deals with objects that change with time. (In a value language the values of variables are unchanging.) Lucid, however, can often be viewed as an object language even though it is a value language.

3.3.1 Operational Ideas in Lucid

The (unchanging) values of variables in Lucid are actually infinite sequences, but the variables should be thought of as objects that change with time. For example, the Lucid definition

$$x = 1 \text{ fby } x + 1$$

(the operator fby means *followed by*) defines the value of x to be the sequence <1, 2, 3, 4, ... $>$ but it is natural and useful to think of the value of x as an object, which is initially 1 and subsequently changes to 2, 3, 4, etc.

This simple idea has many consequences. It means that the language enjoys all the advantages of a functional language, such as referential transparency, provability etc., while retaining an operational iterative flavor. It means that defined functions in Lucid are functions from infinite sequences to infinite sequences, which, in operational terms, means that they are *filters* that produce outputs as inputs are fed in. Moreover these filters do not need to act 'pointwise'– the output at any time need not just depend on the input at that time, it can depend on all the previous inputs. For example, we can write a function that outputs a running sum of the inputs, as follows

```
Sum(x) = s where
                s = first x    fby    s + next x;
        end;
```

Operationally, the filter *sum* keeps around a local variable *s* (for each invocation of *sum*, i.e., for each expression *sum(e)*), which "remembers" the running sum (of *e*).

It also means that all Lucid programs are essentially continually operating programs, which are given infinite sequences of inputs and produce infinite sequences of outputs. This doesn't mean that Lucid is only suited to special applications like process control or data processing; it just means that the operating system would have to take special action if a Lucid program were to be a 'one off' job, if it were required to give only a single result. (In the current implementation of Lucid, this action is invoked when the system gets *eod* as input (*eod* stands for *end-of-data*).) Nevertheless, it does mean that Lucid is well suited for the sorts of applications that are relevant to this project.

3.3.2 Lucid as a Dataflow Language

If we consider the input to a dataflow network to be unending sequences of data items, there will be unending sequences of data items flowing along the edges of the network. Infinite sequences are at the semantic heart of Lucid, and this suggests that there might be a correspondence between Lucid and dataflow networks. In fact there is such a correspondence, and it is very close indeed. Lucid has the pleasing property that every Lucid program is simply translatable into a dataflow network, and every dataflow network is simply translatable into a Lucid program. This is only true if we consider a particular class of dataflow networks, but this class is sufficiently general to be considered as *the* class of dataflow networks: all other classes of dataflow networks appear to be translatable into this class.

In fact, the dataflow networks in the class corresponding to Lucid programs do not use *select* and *distribute* nodes at all. (No *select* nodes are needed because Lucid, as we will see, requires tagged dataflow, and *distribute* nodes are not needed because computations are demand-driven and conditional evaluation is done using

the operation *if-then-else* and the extraction of values from a loop is done using the operation *asa* (which means *as soon as*).) This is a pleasing property, because the *distribute* node is the only node, other than *split*, with more than one output edge. Now dataflow networks are more syntactically uniform: a dataflow network is any directed graph formed from operation nodes, function nodes, and *split* nodes. In fact the dataflow network corresponding to a Lucid program uses only nodes that come directly from the program itself (which is why the translation from Lucid to dataflow networks is so trivial).

The Lucid networks use tagged dataflow, with the tags including a component for Lucid-time. (We cannot consider Lucid-time as actual time; we have seen that tagged dataflow allows us to have *conceptual* sequences that are not necessarily evaluated in order, which is exactly what is required when implementing Lucid.) Also the networks are demand-driven, rather than data-driven, for various reasons which we will not go into here.

This adoption of the demand-driven approach is rather radical, since it is contrary to the approach taken by most other dataflow researchers (although several of them are coming to feel that they have to be able to handle demand-driven computation). To some extent the demand-driven approach may require more computation, to propagate demands, and, in many cases, the data-driven approach causes no problems. This suggests that some sort of hybrid approach would be better: a basically demand-driven approach with some data-driven computation when it is heuristically determined that no computational excesses will result. Some heuristics for use in such a hybrid system are easy to think of, but further research is needed to come up with a comprehensive set.

3.4 Lucid, Demand-driven Evaluation and Fault-tolerance

In a Lucid program, the values of variables are actually infinite sequences, and these sequences are generated as the computation of the program progresses. Variables with no definitions are *input variables*, which take a sequence of values

from the "outside world". Unlike the other variables in a program, the elements in the sequence which is the value of an input variable are always demanded in order. (This is ensured by the interpreter, which generates extra demands if a program tries to access the elements of an input variable out of order.) This means that the elements in the sequence for an input variable must actually be inputted in order. For real time applications, it would be appropriate to modify the dataflow machine so that every input variable v has associated with it another variable t_v which records the actual times (in milliseconds since system start-up) at which the elements of v are inputted to the program. Using these variables it is possible to make the behavior of a program depend on actual time.

Here is a simple program fragment which takes readings from an accelerometer and produces values for velocity and distance (in the same direction as the acceleration is measured). The original velocity and distance, when the readings of the accelerometer are started, are *V0* and *D0*, respectively.

vel = V0 fby vel + acc× delta;
dist = D0 fby dist + vel× delta;
delta = (next t_{acc} - t_{acc})/1000;

The variable *delta* will give the times (in seconds) between successive readings of the accelerometer. (It probably will be constant, i.e. be a constant sequence, but the program works just as well if it is a varying sequence.) The variables *acc*, *delta*, *vel* and *dist* can be used in any way one wishes. For example, if the program contained the following definition

avg(x) = mean where
mean = x fby mean + d;
d = (next x - mean)/(index + 2);
end;

(the Lucid constant *index* is the sequence <0,1,2,...>) then the expression *avg(vel)* will give the running average velocity, which may well be a useful thing to know. If there were readings of the rate of fuel consumption, taken at the same times as

the accelerometer readings, represented by a variable *fuelrate*, then *vel/fuelrate* will give the instantaneous "milage" (actually feet per gallon) and *avg(vel/fuelrate)* will give the average milage. If the fuel tanks originally hold F gallons of fuel, the expression

```
                    vel/fuelrate× left where
left = F fby left - fuelrate× delta;
                                    end
```

will give the distance that can be travelled if the present velocity is maintained. (Actually, if the present fuel consumption is maintained.)

Thus there are many values that may be calculated, in a continuous way. It is possible to have values that are conditionally updated. For example, we could keep track of the distance travelled, once every five seconds. (This would be appropriate for a visual display, which would be unreadable if it changed as often as the accelerometer produced readings.) This could be defined by

```
display = if sawtooth eq 0 then dist else display fi
        where
            sawtooth = 0 fby if sawtooth >= 5
                             then 0
                             else sawtooth + delta fi;
        end;
```

We now must consider the operational behavior of the dataflow machine produced by these definitions. The machine will be demand driven, but to get real-time behavior we will link the actual time when demands for output are made to the times that readings of acceleration and rate of fuel consumption are made. (This is a simplifying assumption for this report. In fact, this assumption does not have to be made, but the extra complications in explanation would just obscure the points that are relevant here.) The demands for output will filter back, and the actual readings will be inputted, *but the actual readings will have been made at the time the demands for output were made.* Now, we are assuming that the output is required very quickly, at least before the next demand for output is made. Thus it must be assumed that the demands for output are made at sufficiently great

intervals of time to allow the computation of each output to be completed before the next demand for output is made. The problem comes when one of the outputs required involves conditional evaluation, as is the case with *display* here. If the variable *dist* is not used anywhere else, it would only be demanded when *display* needs it, namely once every five seconds. To then get the required value of *dist*, all the intervening values of *dist* must be calculated, corresponding to the times since *dist* was last needed, five seconds previously. This will be a longer calculation than usual, and may, in fact, be too long, and overlap with the next demand for output. To get around this, *dist* should be evaluated continuously, as are *dir* etc. This seems to go against the idea of demand-driven dataflow. In actual fact, it is a good example of the benefits of having a hybrid type of evaluation that involves data-driven evaluation when it is certain (as in this case) that the values in question will eventually be needed anyhow, or when it is *possible* that the values will be needed, and calculating them will not involve an excessive amount of computation.

So what are the advantages of demand-driven computation in this context? Surely we have just demonstrated that data-driven evaluation would have been better? The advantage of demand-driven computation is that we can have different types of demands. We can have urgent demands, for values needed right now, and we can have less urgent demands for values, such as might be generated by the part of the hybrid system that simulates data-driven computation. In this case, values for *dir*, say, could be evaluated quickly, and the values of *dist* would be kept up to date at a more leisurely pace, before the next demand for output comes in but not necessarily as soon as possible after the previous demand for output came in. (The advantages of having the two types of demands are more pronounced when the demands for output are made at a lower rate than the rate at which readings are made of the values on which these outputs will be based. Also, it will be natural in practice to have some values whose speedy evaluation is more critical than others, and this naturally fits in better with a multi-priority demand-driven evaluation scheme than with a single priority data-driven scheme. (It is more natural to have a multi-priority demand-driven scheme than a multi-priority data-driven scheme

because the urgency attached to a particular computation comes from the top, from the use that is going to be made of the results of the computation, rather than from the bottom, from the basic operations comprising the computation.))

This is just one simple example of the way in which demand-driven evaluation can be exploited to give greater real-time control over computations than is possible with data-driven evaluation. As such, it is one reason for using a Lucid dataflow machine. Another reason for using such a machine is that the machine runs Lucid, which is a natural language for expressing dataflow computations.

3.4.1 Data-driven Fault-tolerance

The scheme for adding fault-tolerance to a dataflow machine did not depend crucially on the machine being demand-driven. Exactly the same technique could be used with a data-driven machine. The only drawback to using a data-driven machine is that sufficient problems have been found with the data-driven approach that there is now a trend towards the demand-driven approach, at least in some contexts.

3.5 Conclusions

An elementary scheme has been outlined for adding fault-tolerance capabilities to dataflow machines. The scheme will be tolerant of single failures of processors, either failure to produce results at all, or failures which result in the wrong answers being produced. The recovery time when failures are detected is on the order of microseconds, and the overhead for adding fault-tolerance is essentially that twice as many processors are needed (otherwise the machine will compute at approximately half the speed).

The benefits, in terms of reduced execution time, to be obtained from using dataflow, by translating conventional programs into dataflow programs or by interpreting conventional programs using an interpreter written in a dataflow

language, together with the fault-tolerance capabilities that can be obtained, are probably great enough to justify the effort and expense of doing the translation, or of constructing the interpreter. The benefits of writing application programs directly in a dataflow language like Lucid are even greater, and are definitely great enough to justify the effort involved in learning to use such a language.

Chapter 4

Automatic Generation of Aircraft Control Programs

Introduction

As digital electronic devices become cheaper, faster, more reliable and more compact, there will be a continuing trend toward the replacement of the analog electro-mechanical systems used in aircraft applications with newer digital systems that provide similar functions. In particular, the aircraft control functions that have traditionally been implemeted on an individual basis by "black box" analog devices now can be realized using digital computation methods, and if desired, consolidated into one or a few processing units. A given control function then becomes one of a set of programs that are executed on a general-purpose computer, possibly along with other programs that are unrelated to control processes.

The designer of new digital control programs faces much the same set of problems as the analog system designer with respect to meeting the control function requirements that relate to process-dynamics, stability, etc. However, he has a different set of problems with respect to the implementation of the system. The limitations imposed by the characteristics of physical devices (e.g., speed, linearity, and dynamic range of system components) translate into program properties such

as execution speed and accuracy of arithmetic operations. The stability of feedback loops in the analog world becomes a problem of the numerical stability of calculations in program loops. Questions of correct hardware implementation at the circuit level correspond to issues of the correctness of a program with respect to its flow-of-control structure and the details of its arithmetic processes.

In the subsequent sections we will examine the possibility that some of the work involved in the development of aircraft control programs might be automated or semi-automated. The plan initially would be to create a specialized programming environment for the control system designer. This environment would consist of a collection of software tools, each of which was designed to support one or more of the different phases of control program specification, design and implementation. At some future stage one would hope to integrate these tools with some form of knowledge-based "expert system" that could provide advice on all aspects of the program generation problem.

The main motivations for the employment of semi-automatic program generation methods are to reduce the time and cost of program development, and importantly, to increase the confidence that control programs are correct with respect to their specified performance requirements.

4.1 The Nature of the Problem

The stages of development of either a process control system or a process control program involve three distinct phases of activity that need to be carried out in the following sequence.

1. Specification of the process to be controlled
2. Design of the control system
3. Implementation of the design

For a given required aircraft control program it may be the case that the specification, or even the specification and the design already exist. This would be the situation

if, for example, a satisfactory control system had already been constructed using an analog realization, and what was needed was a control program that carries out exactly the same function. For any completely new application, however, it will be necessary for a control program designer to address each of the three tasks above.

What is involved in these three activities? First, with respect to system specification, in Tou's book on control theory [2], he lists the parts of a control system specification as including:

a. Dynamics of the control process

b. Input signals and desired output signals

c. External disturbances

d. Tolerable error and degree of stability

Note that these properties are essentially independent of any assumptions about the method of control system design or its form of implementation. The above items actually comprise four areas of functional specification, and would usually be expressed in a mixture of formal and informal descriptive terminology. For example, input signals may be easy to characterize very precisely by referring to manufacturers data on sensor devices. Other properties such as the dynamics of the process may be harder to describe in a formal way. Such a partial specification might be: It should "feel good" to the pilot. Informally expressed specifications are difficult to deal with in a semi-automatic way because they usually lack precision and completeness. One possible solution to the partial automation of the specification task would be to provide the designer with a formal specification language such as SPECIAL [3]. Another approach employing programming "templates" will be described later.

The design of a control system to meet the functional specifications may or may not presume a particular form of implementation. For example a design could be specified by exhibiting a set of differential equations and associated boundary values that express the desired time dependency of the system inputs and outputs–

but without saying just how the equations should be solved by hardware devices. Alternatively, a design could be expressed by giving a circuit diagram description of the actual hardware to be used to carry out the control process. In either case the design itself contitutes a second level of specification of the desired control behavior (now in a more formal terminology) that must be satisfied by the chosen form of implementation. The design phase is probably the most difficult of the three to automate because it involves engineering knowledge, experience, intuition, and physical and mathematical reasoning. Some of this expertise can be captured in useful form by maintaining a data base of descriptive material and programs corresponding to previous successful designs. We will comment on this later.

Implementation of a design carries out the transition from a specification in terms of a system design to a physical realization. In the case that the implementation is a control program rather than another "black box" then there is some presumed underlying hardware device e.g., a minicomputer that the program needs for its execution. The control program designer should not need to influence the design of that processor. All that he should need to know is that it has the capabilities to execute his program at a desired speed and with satisfactory numerical accuracy. The transition from the design to a program implementation is perhaps the easiest of the three phases to accomplish in an automatic fashion. Once the design is expressed in terms of formal mathematical notation or (network terminology) it is not difficult to construct one or more translators that can accept such descriptions and produce as output a high level language program that "realizes" the design. These translators are nothing more than compilers that convert from one formal language into another.

4.2 Concerning Hardware–and Why?

When considering possible approaches to the automatic generation of flight-control programs, it should be kept in mind that this activity probably will not produce any practical results for three to five years. During that time one can expect a considerable improvement in the available digital hardware that

eventually would be used to implement control systems. It is certain that there will be substantial increases in hardware speed, improved memory capabilities, more specialized computational devices (e.g., signal-processors, floating-point arithmetic units, etc.) and, of course, lower costs. These projections have a strong impact on the feasibility of using automated schemes for control program construction and code generation. Two important aspects of the expected hardware advances are:

4.2.1 Machine Arithmetic

Consider the question of the arithmetic capabilities of a processor that supports and runs a control program. In the very early days of digital computation, the programmers of numerical software suffered intensely by not having fast floating-point arithmetic. They were forced by machine limitations to work with fixed-point number representations, and this lead to many tedious problems with the scaling of variables used in computations. Until quite recently the situation was similar for those intending numerical work on small computers based on, say 8 or 16 bit microprocessors. To write control programs to run on such a machine one must use very careful and rather sophisticated reasoning to make sure that the problems of

- Out of range values
- Overflow/underflow
- Catastrophic cancellation
- Loss of accuracy

will not occur in the running program. The skills, judgement and experience necessary to construct correct programs for fixed-point numerical calculations are difficult to capture in a form that could be used in automatic program generation. For example, such a simple program statement as

$$x := a + b;$$

may be "safe" to use only because the programmer knows the physics of the problem and realizes that the values of a and b will always be in a range such that arithmetic overflow will not occur on addition of the two quantities. If an automatic program generator were to create many similar arithmetic statements in the course of computing some state-variable, then it would be unreasonable to expect it to employ human-like intuitive physical reasoning to decide whether the statement was safe.

The situation is quite different if one assumes that future control programs will be executed on hardware that supports fast floating-point arithmetic. Currently there is an IEEE standard, see Ref. [4,5], for a reasonable set of floating-point representations, and a precisely defined semantics for the associated arithmetic operations. A number of semiconductor manufacturers are already offering IC parts that conform to these rules and provide very fast floating-point capabilities.

For control programs that deal with real physical processes it is rarely the case that the state-variables of the process can be measured (or need to be known) to more than three or four decimal digits of accuracy. A possible exception is the task of inertial navigation, if indeed that can be called a control process. In this instance, the FAA requirements for navigational accuracy lead to distance calculations needing about six decimal digits of precision. Otherwise, if control programs can be assumed to use floating-point arithmetic having moderately accurate numerical representations (say 32 bits), then almost all of the subtleties of programming caused by the the above mentioned difficulties with fixed-point calculations dissappear. For most practical purposes the execution of a program statement involving arithmetic operations on program variables can be assumed to yield results that conform to the "mathematical" semantics of the statement operating on a domain of program variables that is equivalent to the "mathematical" reals.

The above facts have a pleasant impact on prospects for automatic program generation. The crucial point is that one may consider a process to be well specified if it is described in terms of mathematical symbology, formulas or equations. One has the prospect of doing various formal symbolic manipulations of mathematical statements into equivalent program statements without any concerns about purely

arithmetic problems. Moreover, as will be developed in more detail later, there are possibilities for using just a few standard methods for handling a wide variety of control problems.

4.2.2 Speed of computation

Now consider some of the consequences of the trend toward the availability of devices with higher and higher speeds of computation. The computational load on an aircraft control program usually will be proportional to the rate at which control outputs must be generated in order to maintain stability of the aircraft. These output rates differ considerably, depending on the control function. For, say pitch control on a large aircraft, rates of a few iterations of the control program per second are adequate. For something like dynamic flutter-control of an aerodynamically unstable wing, rates as high as 50 to 100 iterations per second have been considered. Since these computational iteration rates depend essentially on such mechanical properties of the aircraft as the resonant frequencys of the wing structure, it is not likely that they will increase significantly in the near future. On the other hand, the available speed of computation for microprocessors and other digital devices can be expected to increase several fold during the next few years. As a consequece, one can expect that for future control programs there will be ample time for computation even at high iteration rates.

The importance of high computation speed to automatic control program generation is that it presents the opportunity to create programs that are not particularly elegant or efficient but do rely on simple computational processes that are much the same for differing control problems. For example, most control system implementations depend on integration of time varying state-variables. When such integrations are carried out numerically (rather than by an analog device) the accuracy and numerical stability of the computation depends strongly on the time interval between samples of the integrand. At the expense of more computation, a smaller interval can be used, and a simpler integration rule. This can simplify the process of translating a specification of function into a program

for doing the calculation. Another case of interest occurs when a device like a compensator or filter is specified in terms of a given electrical network. Here, rather than attempt to derive a mathematical expression for the network transfer function, one might construct a program that repeatedly computes all of the network's node voltages and mesh currents. The same effect is achieved as that of computing a derived transfer function but at the expense of a greater amount of computation. If the extra computation can be afforded, then the passage from a specification in terms of a network to an equivalent program realization becomes more straightforward and probably less error prone.

4.3 An Environment for Control Program Generation

In a programming environment designed specifically for control program composition, the separate phases of control program generation discussed above call for different forms of support tools for each of the activities of

- System Specification
- System Design
- Design Implementation.

Taken together, the tools should provide a programming environment that makes it easier to carry out the different steps necessary for the program development.

In creating the tools, it is important to consider how they can be structured to take maximum advantage of knowledge that resides in previously solved control problems and their implementations. One does not want to have to start from scratch with each new control problem. Instead, the programming environment should support the capabilities to recover and reuse any relevant prior work, and to do so in ways that are easy for the user to manage. Several methods and facilities that could be provided to support this policy are

- The use of several standard forms of template-like data structures that can be used in program development, and also used to store either partial or complete

information about parts of a control program.

- Structure oriented text-editors that help one to make incremental modifications to previously existing programs or program segments, and incorporate these "parts" into a developing program.
- A library of standard numerical procedures or frequently used code-fragments that can be recalled and inserted as function calls or macros in a program that is being composed. Included should be reliable routines for matrix eigenvalue computation and for the solution of polynomial equations. The latter are useful (perhaps essential) to determine values for those design parameters that affect system stability.
- One or more compilers (or translators) that accept program descriptions in symbolic structured-data format and generate executable programs expressed in a high order language such as FORTRAN or Pascal.

The notion of using standard templates for storing and composing control programs is attractive because there are many similarities that extend across all aircraft control applications. For example the programs are all iterative in execution, they all require an initialization phase, they usually have several modes of operation depending on aircraft state, and always read information from sensors and issue commands to actuators. Other similarities occur in the nature of the internal calculations that are carried out, where often the same general type of dynamic process is being modeled by different control programs. In the latter case, two programs might differ significantly only with respect to the parameterization of some standard computational procedure.

Three forms of template suggest themselves as being useful for different aspects of the program development. First, a data structure based on a two or more dimensional partitioning of "space" into pigeon- holes in much the same manner as the business-oriented "spreadsheet" programs. This would be useful in the specification and design phases of program generation. The idea is to use the spreadsheet mechanism to name, record, and identify numerical relationships between the "objects" that will be referred to in the eventual program. Such

objects are, for example, input and output devices, state-variables, and modes of control. The spreadsheet format can be used to associate values, scale-factors, dimensions, limits, etc., that are part of a specification, with the corresponding identifiers in the actual control program. This format also is convenient for specifying or describing the flight mode conditions. For example, one can fill in names and mathematical relations of the form

$$mode_a \quad = \quad (airspeed > 120) \quad \wedge \quad (altitude < 200)$$

This gives a name to the mode contemplated and states the conditions under which the mode obtains. In the case that many modes are specified with complicated dependencies, it would be possible to have an automatic tool that checks the supplied data for consistency of the mode descriptions.

A second form of template that may be useful is a set of standardized control program formats. These could be used either to construct complete working programs or, partially completed, to store useful parts of programs future use. The plausibility of constructing these program templates depends on the fact that control programs tend to have rather similar forms of structure and content, so that several programs actually may have the same generic pattern. To illustrate, a program template written in a Pascal-like syntax might look something like the following.

```
Program Template 1.

procedure control(var firstime: boolean);

   var mode: (m1..mn);                                    (1)
       exitcondition: boolean;
       otherlocals: ....;

   procedure initialize;                                  (2)
      begin ... end;
```

```
procedure compute-mode;                                    (3)
   begin ... end;

begin
     if firstime then begin                                (4)
        initialize;
        firstime := false
     end;
     repeat                                                (5)
        read(input,sensors);
        compute-mode;
        case mode of
           m1:  begin code1 end;
           m2:  begin code2 end;
           ...
           mn:  begin coden end
        end;
        write(output,effectors);
        wait(epsilon)
     until exitcondition
end;
```

This template (or something rather similar in form) could "match" the control structure and computational style of many simple control programs. The var declarations beginning at (1) give names and types to variables to be mentioned in the program body. These declarations could be created with the help of the spreadsheet type of specification discussed above. Similarly, the procedure initilize, starting at (2), can be automatically generated by using initial values supplied when variables were specified and named in the first phase,or created by the second phase. The procedure compute-mode, starting at (3) also can be automatically constructed from the spreadsheet representation.

At program entry, point (4), initialization occurs for the first call of the control procedure and is subsequently disabled. Following this there is a repeated loop that iterates the computation corresponding to a selected mode. The segments designated as code1, code2 ... carry out the actual computations that correspond to different control laws for the different modes. These segments must be supplied by "borrowing" code from previous programs or by a creative process that only partially can be automated. More discussion of this phase will be found in the following section. The use of program templates should be valuable because automatic checking mechanisms can be devised to prompt the user into supplying all of the information necessary to produce a syntatically correct program in the high level language of choice.

A third form of template that would be useful as a semi-automatic programming aid is a data representation form for symbolic matrices. The motivation is that most of the standard methods of control process design involve reasoning about state changes of the system that are expressed in notations of the general form

$$X' = M \times X$$

Here, X and X' are vectors whose elements are input, output and internal state-variables (and their time derivatives) and M denotes a state-transition matrix that relates the current state X to the state at the next "instant" of time. The above formulation can be viewed either as a system of first order differential equations or as a set of finite-difference equations where the implicit time variable t increases by discrete increments delta. In the latter case the increment delta may be the interval between iterations of the main loop in the control program.

If a given control law or design is expressed in symbolic form as a matrix equation representing the discrete time-step interpretation, then it is particularly easy to translate this representation into a sequentially executed high level language program. All that is necessary is that the translator accept the symbolic data structure representing the matrix equation and use the standard matrix-product rule to generate program statements that implement the corresponding

arithmetic on the symbolic program names. Since the matrix M is usually sparse, the translator can omit compiling statements involving multiplication by the zero-elements of M, and thus produce a more efficient code implementation than would be provided by a "full" matrix multiplication algorithm.

Another reason for wanting symbolic matrix templates is that useful parts of prior control system designs may exist that are expressed in the form of state-diagrams, circuit-diagrams or electrical networks. The easiest way of capturing this knowledge in a form that is suitable for automatic translation into program code is to treat the diagrams or circuits as directed graphs where nodes of the graph are given names identified with state-variables, and edges of the graph correspond to relations between the connected pair of nodes and their associated state-variable values. With some help from symbolic data-manipulation routines, it should be relatively easy to transform a state-diagram representation or a network realization into a transition matrix that expresses the "next state" of the system in the same form as that required by the discrete time-step calculations described above. For passive linear electrical networks expressed in either circuit-diagram or Laplace transform style, there are standard techniques that can be used to obtain the transition-matrix representation.

4.4 An Illustrative Example

In order to illustrate these ideas in a somewhat more concrete context, let's examine one typical form of aircraft control problem to see how the use of the proposed programming environment in the system design process leads to formal representations that could be used by an automatic program generator. This will be a very oversimplified "toy" example, but it will contain the essential ingredients of many control system design problems.

Consider the problem of flutter-control for an aircraft that has an aerodynamically unstable wing structure (see Figure 18). In flight the wing is subjected to a time-varying perturbing force, shown as P(t), that creates a bending moment

about the wing root. The designer/user wants to compensate for this disturbance and the known wing instability by varying the angle of a small aileron positioned somewhere along the trailing edge of the wing. He might know that it is possible to make the simplifying assumption that the wing can be physically modeled by an elastic cantelever beam with center of mass CM. He might conclude that the mechanical properties of the wing are such that it is only necessary to consider the fundamental mode of vibration of the wing in the y direction and that all of the higher-order vibrational modes, torsional modes etc., can be ignored in the control system design. He might also conclude that a single sensor that measures wing deflection will suffice as a control input.

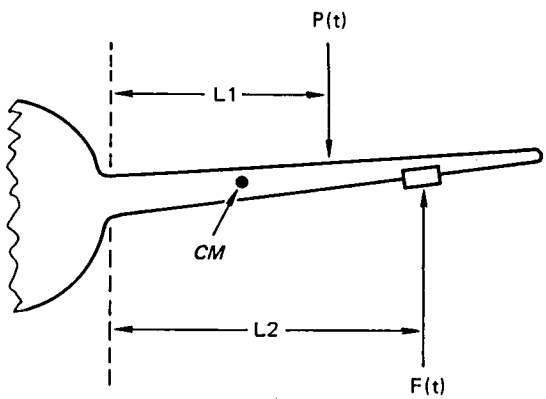


Figure 19. An Aerodynamically Unstable Wing Structure.

The most difficult steps to automate are those that require engineering knowledge and experience. The previous paragraph, which was an informal partial specification of the control problem, already contains a number of insights and judgements about how the system can be modeled. This type of knowledge is difficult to capture in heuristic form, but it might be elicited by a knowledge-based "expert system" through interactive inquiries to the user. Even a non-expert system can provide the capability to browse through previously stored design and specification descriptions in search of pertinent or useful partial solutions.

At this stage, the designer may decide that the problem is similar to that

of roll-control, and decide to use the program template previously used for that purpose. This could be very like the program template discussed in the section above. He then proceeds to supply information required by the various parts of the program. To fill out the appropriate spreadsheets he would be obliged to supply names and specifications for input and output devices used by the program. This might be done by giving the name y (which is actually wing deflection) to the signal value of a particular type of strain guage that appears in the spreadsheet along with manufacturers data on its characteristics. Similarly he may assign the name theta to the deflection angle produced by some already specified actuator device, and designate that as a program output.

Proceeding in this way, he must next describe the modes of the control system. This might take the following form

$$\begin{aligned} &supersonic = airspeed > 600 \\ &subsonic = (airspeed < 600) \quad \wedge \quad (airspeed > 120) \\ &landing = airspeed < 120 \end{aligned}$$

This gives names to three control modes which in the designer's judgement require different control laws and therefore different program treatment. At this point he realizes that airspeed is also an input to the program and returns to the input/output specification to supply the name airspeed to one of the aircraft devices that measure it.

After one or more iterations of this specification process, the designer is ready to consider the actual design details of the control system. Reasoning by analogy with the previous roll-control problem that is known to be similar he might choose to look at the control law descriptions that have previously been used in that application. He may decide to borrow one of those implementations or, since the dynamics are so simple for this problem, he might just write out a differential equation for the control law by summing moments about the wing root. Such a relation might look like

$$m \times CM \times ddy + k1 \times dy + k2 \times y = L1 \times P(t) + L2 \times F(t)$$

here dy and ddy are the first and second derivitives of the wing deflection y respectively, m is the wing mass, F(t) is the force due to the aileron, L1 and L2 have the dimension length, and finally, k1 and k2 are constants that depend partly on mechanical properties of the wing and partly on the assumed short-term gain of the control system. This equation introduces two new state-variables, ddy and dy, as names of program values, and the programming system should at this point inquire about their initial values. Also the functions P(t) and F(t) have not been defined, so the designer should supply some function definitions at this point. Say,

$$F(t) = A(mode) \times airspeed \times sin(theta)$$

$$P(t) = U(t)$$

where U(t) stands for the unit step function of perturbation and A introduces a new program variable that depends on flight mode.

At this point the programming system's symbolic manipulation facilities could be used to generate the transition matrix of the control law and to request values for the design parameters k1 and k2 and A. Either an automatic polynomial solver or a matrix-eigenvalue routine can then be invoked to give the characteristic roots of the system that determine its stability. The designer may use the latter tool in a "cut-and-try" iterative fashion to converge on system parameters that give what he regards as satisfactory dynamic behavior. He will repeat this exercise three times, one for each of the control modes previously specified. Finally, an automatic translator can be used to convert the now determined transition-matrix representations into executable code sequences that fit into the case-statement part of the program template.

Some details have been omitted in the foregoing scenario, either deliberately because they seemed inessential, or because we have overlooked important difficulties. In any event the above general style of interaction with a specialized programming invironment is suggestive that progress can be made in semi-automatic control generation.

4.5 Concerning Accuracy

One may inquire whether the replacement of analog-style devices with digital devices might introduce unforseen problems caused by the accumulation of error through round-off or truncation of values while the many arithmetic operations are carried out. It turns out that for any control system that was designed to be stable against small environmental perturbations (say noise), that system will also be stable against any small arithmetic errors. This fact can be proved quite generally for all linear systems, and is undoubtedly true as well for stable nonlinear ones. The reason is that a stable computation has the property that the current state is very weakly dependent on states in the remote past–that is, history gets attenuated. Consequently, although errors occur continuously they never can accumulate to a significant value.

4.6 Conclusions and Recommendations

Control programs constitute a rather limited sub-class of the complete spectrum of numerical software. They share a number of similarities in form, purpose, content and accuracy requirements that allow the construction of such programs to be partitioned into a set of well defined subtasks. Most of the subtasks (that are not related to human creativity in the design phase) are not too complex, and usually can be handled using one or a few standard methods.

For the above reasons, the prospects for semi-automatic program generation seem considerably better for control programs than for more general numerical software construction. One at least can see how semi-automatic methods could be applied to the different phases of control program generation. However, considerable research needs to be done in several areas before any substantial results will emerge.

Two critical areas are the organization of expert knowledge and the specification of control dynamics. Ways need to be found to organize the kind of reasoning that is involved in the choice of a control law for a particular problem, selection

of internal state- variables etc. With regard to specification of control dynamics, there are criteria such as

- Don't subject the passengers to unpleasantly large accelerations.

that are relatively easy to specify by stating limits on control function outputs. To meet these requirments by a semi-automated design process, however, seems to require an iterative design approach that would make use of simulation to verify the system properties. This is an area of research that needs to be investigated. There are other types of specification such as

- Optimize the control law with respect to fuel consumption

that we don't yet know how to deal with in semi-automatic fashion. There are some well developed mathematical techniques for treating such problems, see for example Sage [6], but their applicability in a programming environment seems difficult and needs to be explored.

References

[1] Goldberg, Jack, et al., Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer, NASA Contractor Report 172146.

[2] Tou, Julius T., Modern Control Theory, McGraw-Hill Electrical and Electronic Engineering Series.

[3] Robinson, L., The HDM Handbook, Prepared for: Naval Ocean Systems Center, SanDiego, California 95152.

[4] Stevenson, David, A Proposed Standard for Floating-Point Arithmetic, Computer, Vol. 14, No. 3, March 1981.

[5] Cody, W. J., Analysis of proposals for the Floating-Point Standard, Computer, Vol. 14, No. 3, March 1981.

[6] Sage, Andrew P., Optimum Systems Control, Prentice-Hall Electrical Engineering Series.

[7] Ashcroft, E. A., and Wage, W. W., Lucid – A Formal System for Writing and Proving Programs, SIAM Journal on Computing, Vol 5, No 3, September 1976.

[8] Ashcroft, E. A., and Wage, W. W., Lucid, a Nonprocedural Language with Iteration, Communications of the ACM, Vol. 20, No. 7, July 1977.

[9] Pingali, K., Efficient Demand-driven Evaluation, Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, June, 1983.

[10] Akers, S. B., On the Construction of (d,k) Graphs, IEEE Transactions on Electronic Computers, Vol EC-14, No. 6 (June 1965), pp488.

[11] Elspas, B., Topological constraints on interconnection limited logic, Switching Theory and Logical Design, Vol S-164, Oct 1964, pp133.