Data Flow Models
for
Fault-Tolerant Computation

by

Gregory M. Papadopoulos
B.A., University of California, San Diego (1979)

Submitted in partial fulfillment of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology

June 1983

© Gregory M. Papadopoulos

Signature of Author . . . .
Department of Electrical Engineering and Computer Science
May 13, 1983

Certified by . . . .
Arvind, Associate Professor of Electrical Engineering

Certified by . . . .
Larry D. Brock, Charles Stark Draper Laboratory

Accepted by . . . .
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 2 3 1986
LIBRARIES

ARCHIVES

Acknowledgments

I thank Professor Arvind for his encouragement, confidence, and valuable contributions in this work; Dr. Larry Brock for his unending support and confidence, in me and in my research; Dr. Basil Smith and Dr. John Deyst for the many valuable discussions and suggestions that has kept the research topical; Most importantly, my wife, Elizabeth, for her unfailing support, encouragement, and unmatched editorial skills.

Special mention is due to the Charles Stark Draper Laboratory for providing the facilities, and support for this research. This work was performed under IR&D Project No. 18701. Publication of this thesis does not constitute approval by Draper Laboratory of the findings or conclusions contained herein.

I hereby assign my copyright of this thesis to the Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.

Permission is hereby granted by the Charles Stark Draper Laboratory, Inc. to the Massachusetts Institute of Technology to reproduce any or all of this thesis.

Data Flow Models for Fault-Tolerant Computation

by

Gregory M. Papadopoulos

Submitted to the
Department of Electrical Engineering and Computer Science
on May 13, 1983 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

Data flow concepts are used to generate a unified hardware/software model of dispersed physical systems which are prone to faults. Fault propagation is restricted to occur only through the data exchange mechanisms between independent modules. A powerful fault model, *token fragments*, is used to express vagaries of such exchanges in the presence of faults. The need for *source congruence* among inputs to similar fault-free modules is established, and procedures are developed for both the construction and analysis of *congruence schemata* in the presence of faults. The need for nondeterminate operators is justified for real-time systems, and a method for maintaining source congruence is presented for such systems. *Events* are defined as the receipt of tokens by operators, and a technique for minimizing the time-skew between similar events, *synchronization*, is developed. Applications are explored and bounds are given for the minimum number of independent hardware modules that are required for correct system operation in the presence of a specified number of faults.

Table of Contents

Chapter One: Introduction 7

1.1 The Data Flow Approach 8
1.2 Problem Development 9

Chapter Two: Simplex and Redundant Programs 11

2.1 Source Language, L_0 11
2.1.1 Syntax 12
2.1.2 Semantics 12
2.1.3 Input Robustness of L_0 Programs 15
2.2 A Language to Express Redundancy: R_0 16
2.2.1 R_0 Program Graphs 16
2.2.2 L_0 Program Instances 17
2.2.3 Voters 17
2.2.4 Restoring Buffers 18
2.2.5 External Inputs and Redundant Outputs 18
2.3 Abstracting the Implementation: Fault Models, Fault Sets, and Links 18
2.3.1 Fault Sets 19
2.3.2 Interconnection Links 20
2.3.3 Fault Models 21
2.3.4 Restoring Buffers Revisited 23
2.4 Summary 24

Chapter Three: Making Simplex Programs Fault-Tolerant 25

3.1 Overview of the L_0 to R_0 Transformation 25
3.2 Bit-for-Bit Agreement and Congruence 29
3.2.1 Congruent Sets 31
3.2.2 Congruence Tests 31
3.3 Testing R_0 Models for Correctness 32
3.3.1 Input Robustness and Correct Operation of R_0 Programs 32
3.3.2 Fail-Operate Specification 33
3.3.3 Aggregate System Reliability 33
3.4 Required Level of Redundancy 34
3.5 R_0 Schemata For Congruent Inputs 36
3.5.1 C_1^p Schemata 37
3.5.2 C_f^p Congruence Schemata 41
3.5.3 Remarks 44
3.6 Sequential Faults 46
3.7 Graph Coloring Algorithm 47

Chapter Four: Nondeterminism and Synchronization 48

4.1 Events 49
4.2 Nondeterministic Source Language, L_1 50
4.3 Nondeterminism in R_1 52
4.4 A Priori Orderings 53
4.4.1 Timing Races 54
4.4.2 Bounded-time Congruence Schemata 55
4.5 Simultaneous Events 59
4.6 Synchronization 61
4.6.1 Event Skews 62
4.6.2 Self-Synchronizing Schemata 62
4.6.3 Latency 65
4.7 Remarks 65

Chapter Five: Applications 66

5.1 Sensor Models 67
5.1.1 Demand Driven Sensors 67
5.1.2 Processor Hosted Sensors 68
5.1.3 Sensor Cross-Strapping 70
5.2 Physical Clocks 72
5.3 Critique of Current Systems 74
5.3.1 SIFT 75
5.4 FTMP and FTP 76
5.4.1 The Space Shuttle System 78
5.4.2 C.vmp 80

Table of Figures

Figure 2-1: Example Source Program Schema 13
Figure 2-2: An n-input p-output Function 14
Figure 2-3: 3-Input Voter 17
Figure 2-4: Restoring Buffer Schematic 18
Figure 2-5: Sample R_0 Program 21
Figure 2-6: Inconsistent Reception of a Token Fragment 23
Figure 3-1: Example Source Program 26
Figure 3-2: General Topology for Triplex P 27
Figure 3-3: Hazardous Distribution of External Inputs 28
Figure 3-4: A Correct Triplex Version of P 30
Figure 3-5: Schematic for a C_f^p Congruence Schema 37
Figure 3-6: Canonical C_1^p Schema 38
Figure 3-7: Minimal C_1^p Schema Assuming Independent Source 39
Figure 3-8: Failure of a C_1^3 Schema with Two Faults 41
Figure 3-9: Minimum Fault Set Implementation of C_f^p 43
Figure 3-10: Minimum Stage C_f^p 45
Figure 4-1: Two Nondeterministic Operators 51
Figure 4-2: Decomposition of Programs into L_0 Fragments 52
Figure 4-3: Prior Orderings from Dependencies and Timing Paths 53
Figure 4-4: Time-out Test as a Timing Race 55
Figure 4-5: Hazardous Redundant Time-out Test 56
Figure 4-6: Restoring Buffer with Time-out 57
Figure 4-7: A Canonical Bounded-Time τC_1^p Congruence Schema 58
Figure 4-8: Standard Representation of a Canonical τC_1^p Schema 59
Figure 4-9: Correct Implementation of Redundant Merge 60
Figure 4-10: Self-Synchronization of Redundant Edges 63
Figure 5-1: Simplex Model for a Demand Driven Sensor 68
Figure 5-2: R_1 Model for a Demand Driven Sensor 69
Figure 5-3: Fault Set Assignment in the C.S. Draper Triplex FTP 71
Figure 5-4: Correct Model for a Cross-Strapped Sensor 72
Figure 5-5: Absolute Clocking of a Program 73
Figure 5-6: Physical Clocks from a Self-Synchronizing Schema 74
Figure 5-7: Example SIFT Configuration for $f = 1$ 76
Figure 5-8: Simplified NASA/IBM Space Shuttle Computer System 79
Figure 5-9: Carnegie-Mellon C.vmp Configuration 81

Chapter One

Introduction

The direct replication of digital modules, known as modular redundancy, is a popular technique for improving the reliability of digital systems [1] [2] [3]. Most implementations are for real-time control systems, such as aircraft, spacecraft, and critical process control. In these systems, a loss of computational capability can directly lead to substantial financial and human costs. In applications of interest, the probability that the system will catastrophically fail, due to loss of control functionality, must be less than one part in 10^9 per operational hour.

Despite the diversity of applications and the frequency of attempts, there is a paucity of theory and practice for the analysis, design, and development of highly reliable digital systems. In fact, the vast majority of approaches are ad hoc and suffer from serious theoretical shortcomings.

Basic results associated with aspects of redundant computation and computer synchronization, however, have been developed in the excellent work of Lamport, Shostak, and Pease [4] [5], and Davies and Wakerly [6]. These results have been correctly exploited only by a minority of fault-tolerant system designs, notably SRI's SIFT computer system [7], and C.S. Draper Laboratory's FTMP [8]. A paper design of a packet switch communication architecture by Clement Leung of M.I.T. [9] contains some improvements and restrictions, but is rooted in substantially the same principles.

Unfortunately, no uniform theories for the systematic analysis and construction of general redundant systems has evolved from these efforts. An engineer interested in constructing a fault-tolerant system faces a bewildering array of ideas and choices [10] [3], most of them incorrect. Even the above implementations are burdened by excessive complexity or synchronization requirements (*e.g.,* FTMP), or severe inefficiencies (*e.g.,* SIFT). A simple yet provably correct engineering model for the design and analysis of highly reliable redundant digital systems is clearly called for.

1.1 The Data Flow Approach

The correct design of a fault-tolerant system must consider the interactions of algorithms with imperfect hardware. The problem is not separable. Algorithms must be sensitive to aspects of the hardware while hardware design must provide the resources for the correct execution of the algorithms. This report develops a unified framework for the synthesis and analysis of highly reliable redundant systems.

Unfortunately, commonly used models of computation do not mesh well with models of distributed physical resources and communication interconnection. While von Neumann models and associate imperative languages have, in general, been very successful, the rigorous extension of these models to distributed programs is often intractable. Data flow computation models [11] [12], however, provide a very well-suited framework on which to impose the necessary hardware abstractions of physical dispersion and interconnection. In addition to unifying the hardware and software under a single analytic paradigm, data flow inherently lacks any requirements for centralized and/or synchronous control.

The problem in fault-tolerance is to specify the interconnection and communication between computational elements. Data flow is the disciplined study of such interconnections. One expects that data flow languages and graphs can provide a rigorous foundation on which to formulate and implement fault-tolerant systems.

In the proposed hybrid model, data flow operators are grouped into sets called *fault sets* which correspond to the physical processing sites which fail independently of each other. Edges in the dataflow graph which connect fault sets are called *interconnection links*. Faults may propagate from one fault set to another only through corrupted data on the interconnection links. That is, a fault in set $\mathcal{A}$ can only cause erroneous operation in a set $\mathfrak{B}$ by giving the operators in $\mathfrak{B}$ 'bad' data, *not* by changing the functionality of $\mathfrak{B}$.

This model allows complete data flow analysis of fault-propagation. Outputs from fault sets which are assumed to have failed are replaced with an arbitrary stream of special tokens called *token fragments*. Similarly, the hardware designer can extract the driving design features of the distributed system: the number and nature of independent modules that need to be designed, and the nature and topology of the interconnection.

In this work, we restrict our ability to detect and isolate faults through the application of

redundancy. In other words, we test the validity of an output by comparing it for an exact match with a redundant output. We term any redundant system in which redundant outputs may be bit-for-bit compared to detect failures as *congruent*. This is to be contrasted with more semantically rich tests such as reasonableness tests which attempt to determine whether an output is "reasonable" given some knowledge of the function and inputs.

We assert that *congruent active redundancy* has the lowest semantic content and offers the highest reliability of any fault detection and isolation scheme for deterministic systems. In fact, we are able to transform any data flow program into a redundant model which can be implemented to yield a prescribed level of reliability.

This ability is not as trivial as it might seem at first glance. Informal reasoning about redundant systems has often proven disastrous. There are many systems which have been built and plan to be built [13] [14] [15], and yet do not meet the necessary conditions of this theory. As a result, the systems may possess single point failures which invalidate analysis of effectiveness.

1.2 Problem Development

Chapter Two lays the framework for the *simplex source* programs, programs which assume a perfect machine, that will be transformed into a redundant *model*, a data flow language with hardware abstractions The source program is given in the form of a directed graph. These graphs are interpreted as a recursive connection of monotone and continuous functions. The functions are the graph nodes and map histories of data on the input edges to histories of data at its outputs. A complete partial ordering is imposed which is a naturally extended prefix ordering of these histories. The notions of *fault sets* and *interconnection links* are also developed and a general fault model, *token fragments*, is introduced.

Chapter Three develops construction rules for correct fault-tolerant graphs. The need for *input congruence* is established. This is a statement that all fault-free redundant graphs must have identical inputs for the outputs to be identical. A canonical model schema, known as a *congruence schema*, is developed. This schema guarantees input congruence for a specified number of independent failures.

Chapter Four presents techniques for incorporating nondeterministic operators. Such operators are necessary for "time-out" tests, which are essential in real time systems. The concept of an *event*, the receipt of a token by an operator, is introduced. Correct implementation of nondeterministic functions is cast as a problem of generating a consistent time-total ordering of selected events at redundant sites. A completely distributed *synchronization* technique is developed which, without the need for a system of global clocks, interactively bounds the time-skew between similar events.

Chapter Five examines applications. This includes models for cross-strapped sensors which may be independently "read" by several processing sites, physical clocks and time scheduling. Several current high reliability systems are briefly critiqued using the theory developed in this report.

Chapter Two

Simplex and Redundant Programs

Our goal is to take an arbitrary (but well-formed) dataflow program, the *simplex source*, make it *redundant*, and impose the necessary restrictions on the physical implementation so that it is *fault-tolerant*. The source program is given in the deterministic language L_0, and the programmer assumes that it will operate on a perfect machine. To make it tolerate hardware faults, the source program is replicated, and the resulting redundant program is described by the language R_0. Finally, R_0 programs are augmented with the salient features of a physical implementation: the effects of a fault, the ability to physically isolated selected operators, and aspects of the data communication protocol between processing sites which fail independently of each other. The result is a *model* of the fault-tolerant system to be implemented.

This chapter presents the syntax and semantics of the languages L_0 and R_0, and presents abstractions of the physical implementation. Later chapters will give the transformation algorithms that take simplex source programs and translate them into a model of system that will continue correct operation in the presence of a specified number of faults.

2.1 Source Language, L_0

The following syntax and semantics for the source language, L_0, are largely due to Kahn [16] and Manna [17]

We are given the source as a *simplex* program.

> **Definition 2-1:** A *simplex source program* is a program that is written in the source language L_0, and which assumes it is operating on a perfect machine.

The whole problem is that physical machines are not perfect, they are prone to faults. Our techniques need not bother the source program writer, however. They can automatically transform a simplex source program into a redundant one which operates correctly in the

presence of a specified number of independent faults.

It is important to separate the problem of the correctly executing program from the correctly executing *system*. For instance, we can offer no assurances that the source program is correctly written and will yield meaningful results. Nor can we guarantee the correct operation (or failure detection) of an input since the generating sensors are usually stochastic and can not be directly compared. *Any sensor testing must be done by the program since the semantics are too rich to characterize here. We shall only generate a reliable vehicle on which to execute the program.*

2.1.1 Syntax

A *source program schema* is an oriented graph with labeled nodes and edges. Incoming edges which have no source node are called *inputs*. Outgoing edges with no destination node are called *outputs*. An example is shown in Figure 2-1.

2.1.2 Semantics

We shall treat the information flowing on the edges as tokens in the usual data flow sense. Suppose that each token (a real, integer, function, *etc.*) is an element of the set D. We let ω denote the empty token. The sequence of all tokens which are generated on an edge shall be called the *history* of the edge. All Finite sequences are naturally extended to denumerably infinite sequences by appending ω's. Let the set of all such denumerably infinite sequences be denoted by D^{ω}.

> **Definition 2-2:** A *complete partial ordering* on D^{ω} is defined as follows. Let a and b be any two distinct non-null elements of D. A partial order $\sqsubseteq$ of elements in D is given by
>
> 1. $a \sqsubseteq a$
> 2. $\omega \sqsubseteq a$
> 3. $a \not\sqsubseteq b$ and $b \not\sqsubseteq a$
>
> We now define the ordering on D^{ω}:
> Let $\mathbf{a} \in D^{\omega} = \langle a_1, a_2, \ldots \rangle$; $\mathbf{b} \in D^{\omega} = \langle b_1, b_2, \ldots \rangle$
>
> Then $\mathbf{a} \sqsubseteq \mathbf{b} \Leftrightarrow a_j \sqsubseteq b_j \quad \forall j$.

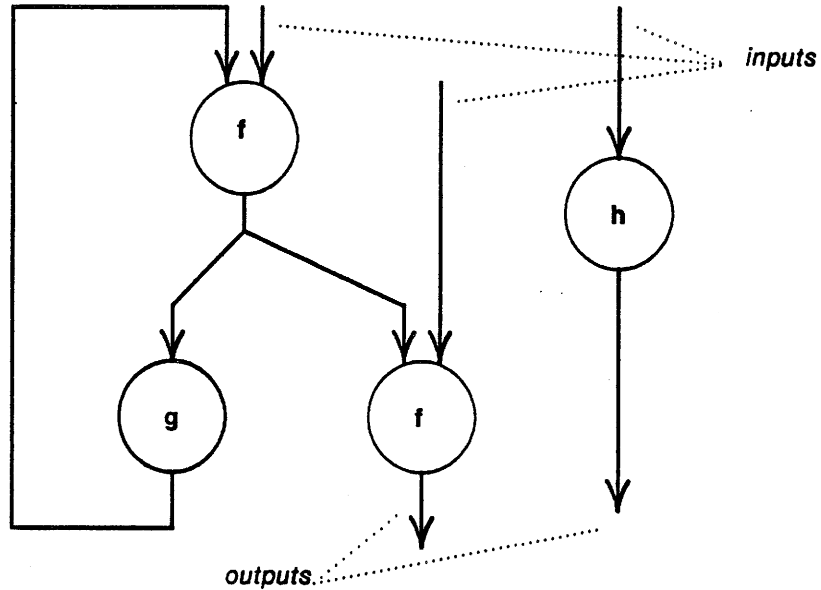


Figure 2-1: Example Source Program Schema

Definition 2-3: An *n-input, p-output function,* **f**, is a mapping from histories of the input edges to histories of output edges (Figure 2-2)

$f_1: D^{\omega}_1 \times D^{\omega}_2 \times \ldots \times D^{\omega}_n \rightarrow D^{\omega}$

$f_2: D^{\omega}_1 \times D^{\omega}_2 \times \ldots \times D^{\omega}_n \rightarrow D^{\omega}$

•
•
•

$f_p: D^{\omega}_1 \times D^{\omega}_2 \times \ldots \times D^{\omega}_n \rightarrow D^{\omega}$

Definition 2-4: An n-input, p-output function, **f**, is *monotone* if for any two sets of input sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$

$A \sqsubseteq B \Rightarrow f(A) \sqsubseteq f(B)$

where the partial order for an ordered set of sequences is defined as the conjunction of partial orders for each corresponding sequence.

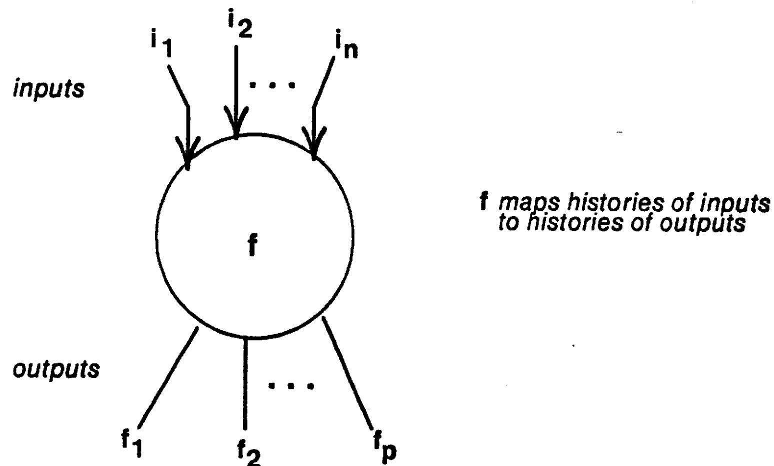


Figure 2-2: An n-input p-output Function

The function is *continuous* **iff** it is monotone and for any ordered sequence of histories $A_1 \sqsubseteq A_2 \sqsubseteq \ldots$

$$f[\mathbf{lub}(A_1 \sqsubseteq A_2 \sqsubseteq \ldots)] = \mathbf{lub}[f(A_1) \sqsubseteq f(A_2) \sqsubseteq \ldots].$$

where **lub** is the least upper bound of the sequence.

We restrict all source programs to be a recursive construction of monotone and continuous operators. In concrete terms:

1. Functions must be determinate, *i.e.*, independent of the relative times of arrival of tokens at the inputs.

2. Monotonicity restricts functions to monotonically adding information at its output, *i.e.*, a function may not revoke a token once it has been output.

3. Continuity prevents a function from sending output only after receiving an infinite amount of input.

It should be noted that we allow the more general class of functions which map *histories of tokens* rather than just tokens. This permits functions which have internal "state".

We have introduced all of this structure primarily to exploit the important property that *a source program schema consisting of recursive connections of monotone and continuous functions is itself monotone and continuous* [16].

2.1.3 Input Robustness of L_0 Programs

Even though the simplex source programmer is not concerned with failures of the machine which is executing the program, he must deal with the possibility that certain inputs to the program are erroneous. That is, it is the responsibility of the source program to produce meaningful results even when certain of the inputs have failed. Usually, this is done by employing redundant inputs to measure those quantities which are of vital importance to the correct operation of the system. Other times, the malfunctioning of the input may sacrifice system performance, but operation is still within acceptable bounds so we shall say the program is producing meaningful results. In any case, L_0 programs must have the ability to deal with certain combinations of erroneous or meaningless inputs.

All L_0 programs assume well-formed tokens at the inputs, whether these sequences are meaningful is extremely context sensitive. We attempt to characterize these semantics in a neutral way, the *valid input space* for L_0 programs.

> **Definition 2-5:** For any L_0 program with *n* inputs the *valid input space* is $\mathcal{V}_P \subset D^{\omega}{}_1 \times D^{\omega}{}_2 \times \ldots \times D^{\omega}{}_n$ such that if the histories on inputs to the program $\langle i_1, i_2, \ldots, i_n \rangle \in \mathcal{V}_P$ then the outputs of the program are meaningful.

The explicit determination of $\mathcal{V}_P$ is usually intractable. Instead, we are only interested in the structure of $\mathcal{V}_P$ for our analysis. We seek the ability of an L_0 program to produce meaningful results when certain inputs are replaced with an arbitrary history. That is, the correctness of the program is insensitive to certain combinations of arbitrary input failures. For example, suppose the program requires temperature data and employs a triple redundant sensor. The temperature used is the median value of the three readings. Thus if two sensors are functional, we can substitute arbitrary failures for the third sensor and the program should still produce meaningful results (if the temperature is differentiated a better filter is required. However, the results may not be *identical* to the unfailed case.

This structure is characterized by an abstraction of the valid input space.

Definition 2-6: The *robust input failure direction set*, V_P, for an L_0 program **P** with **n** inputs, describes all combinations of inputs which can be replaced with arbitrary histories and still be elements of Υ_P. That is,

$\langle l, k \rangle \in V_P$ if for any $\langle v_1, v_2, \ldots, v_l, \ldots, v_k, \ldots v_n \rangle$ that correspond to fault-free inputs,

$$\text{then } \langle v_1, v_2, \ldots, d_l, \ldots, d_k, \ldots v_n \rangle \in \Upsilon_P \qquad \forall d_l, d_k \in D^{\omega}.$$

This simply says that if $\langle l, k \rangle \in$, then the program will produce correct outputs when all inputs are fault-free except l and k, which may be replaced by arbitrary sequences.

2.2 A Language to Express Redundancy: R_0

The source language has no constructs for expressing redundancy or physical implementation. Therefore, we cannot meaningfully ask the question: *Is a given L_0 program fault-tolerant?*

Ultimately, we will take a source program, replicate it, disperse the replicated copies to independent processing sites, and provide the necessary "glue" to ensure that all fault-free copies will produce identical outputs. The first step is to express the process of replication and provide the operators required for connecting redundant programs together. We do this in a language called R_0. Our automatic procedures for making a program fault-tolerant are transformations from L_0 to R_0.

2.2.1 R_0 Program Graphs

We simply present R_0 here with a minimum of justification for the extra operators we have added. There are four elements in the language R_0:

1. *Program instances*, which are complex nodes corresponding to the simplex source program,

2. *Voters*, which output tokens that correspond to a majority of input tokens,

3. *Restoring Buffers*, which are simply identity operators but will have certain signal restoring properties given later, and

4. *Edges*, which connect together the program instances, voters, and restoring buffers.

Definition 2-7: A *R_0 program* is a directed graph of recursive connections of L_0 program instances, voters, and restoring buffers.

We note that R_0 programs have the same semantics of L_0 programs, *given no failures*.

2.2.2 L_0 Program Instances

We need a way to represent the replication of L_0 programs. Because L_0 programs are monotone and continuous functions of their inputs to outputs, we can collapse the entire source program, say P, into a single node and still maintain the semantics of L_0. The incoming edges to a node P are the inputs to the program P, while the outgoing edges correspond to the outputs. Each instance of P in an R_0 program is an *instance of P*. The *level of redundancy* in R_0 is simply the number of instances of P.

2.2.3 Voters

We will often have a need to reduce a set of redundant streams of tokens down to a single stream. The resolution of the redundant copies is done by a *majority operator*, or *voter*, which is a monotone and continuous R_0 function. An example is shown in Figure 2-3.

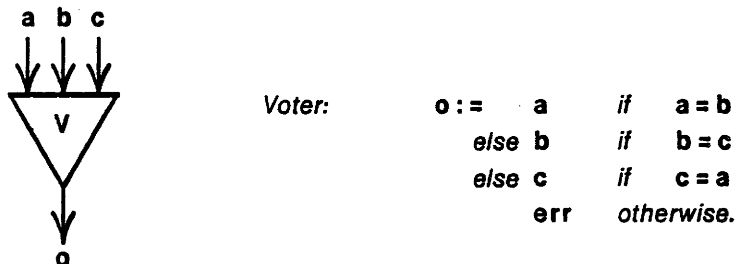


Figure 2-3: 3-Input Voter

Note that a p-input voter is defined as:

$$\textbf{output} := \begin{cases} \textbf{input}_i & \text{if } \textbf{input}_i \text{ is equal to at least } 1 + p \text{ div } 2 \text{ other inputs,} \\ \textbf{error} & \text{if there is no such input.} \end{cases}$$

That is, the output is the majority of the inputs, an error value if there is no majority, or ω if the decision is not possible with the available inputs. It is easy to verify that this function is monotone and continuous.

2.2.4 Restoring Buffers

A restoring buffer is simply the identity function for well-formed tokens. Later, we shall require restoring buffers to have particular implementation properties. Its schematic representation is shown in Figure 2-4.

Figure 2-4: Restoring Buffer Schematic

2.2.5 External Inputs and Redundant Outputs

We shall say that any edges in an R_0 program which have no source node are the *external inputs* to the program, while edges that have no destination node are *outputs*. Those outputs which correspond to similar edges from redundant program instances are called *redundant outputs*. In general, the external inputs to an R_0 program correspond to those inputs of a simplex source program if it were operating on a perfect machine.

2.3 Abstracting the Implementation: Fault Models, Fault Sets, and Links

Even though R_0 can represent the replication of L_0 source programs, we still cannot tell whether an R_0 program is fault-tolerant. Since our formulation of faults includes anomalous system behavior due to hardware failures (we assume the source program is correctly written), we can discuss the fault-tolerance of a program only after certain important characteristics of the implementation are known. When an R_0 program has been augmented with the necessary abstractions we call it a *model* of the system to be implemented.

Our assumptions of the underlying implementation are as follows:

1. Operators may be grouped into *fault sets* which correspond to dispersed physical sites which fail independently of each other.

2. Communication between fault sets is restricted to occur only through *interconnection links*, which have two important properties:

   a. The interconnection topology is fixed. A destination fault set always knows the source of incoming information.

   b. Information only flows in the direction of the edges. A faulty destination set cannot induce a source set to fail via link which is directed from source to destination.

3. Faults propagation between fault sets can only occur through the exchange of information on interconnection links, and the range of possible data that faulty set may produce is totally captured by the fault model of *token fragments*

Briefly, the semantics of a fault are characterized by the fault model of token fragment. Tokens and token fragments are the only method for communicating information, and thus control, between fault sets, and they travel only along interconnection links.

2.3.1 Fault Sets

Implementing an R_0 program requires the mapping of the function nodes and edges onto specific hardware. Any hardware is suceptable to *threats* [3]. These are stresses which produce anomalous (*i.e.*, non-functional) conditions. Threats may manifest in many forms: normal environmental threats such as component aging, abnormal environmental threats such as transients, or software/hardware design flaws. *Faults* are the anomalous conditions resulting from threats. A *fault model* is a functional description which captures important aspects of expected faults.

Any hardware design must assess the class of threats which the redundancy will be expected to handle. We then partition the system into independent *fault sets*, each set having independent failure statistics from the others. Formally,

Definition 2-8: A *fault set* $\mathcal{A}$ is a collection of nodes and edges and a failure rate λ_A such that,

1. A threat which effects the set can cause a malfunction of any and every node or edge in the set.

2. Given fault sets $\mathcal{A}$ and $\mathcal{B}$ with failure rates λ_A, λ_B respectively. The joint failure rate (*i.e.*, the probability that both $\mathcal{A}$ and $\mathcal{B}$ will fail within a unit time interval) is given by $\lambda_{AB} = \lambda_A \lambda_B$.

2.3.2 Interconnection Links

Fault sets abstract our intuitive notion of physical isolation. Generally speaking, functions implemented on one processor will be in a different fault set from functions implemented on another, if suitable isolation (*e.g.*, electrical and electro-optical isolation) is engineered between the processors. If these processors exchange information then the malfunction of one processor, by generating faulty results, might very well cause the other to compute irrational results. The second processor is still fully functional (*e.g.*, it can still add). The primary goal of fault-tolerant system design is to *prevent the malfunctioning of a limited number of units to cause other healthy units to obtain incorrect results.*

Clearly the way a malfunctioning unit may influence good units is through the exchange of data and the exercise of control. Data flow models have the virtue that there is no separate control mechanism. Thus all effects of a malfunctioning unit upon good ones can be expressed purely in terms of information exchanges. This leads to the second important feature of R_0 models:

> **Definition 2-9:** The edges which join nodes which are in different fault sets are called *interconnection links*. These links are abstractions of the real communication medium. We demand two implementation properties of these links, however:
>
> 1. The source of information (*i.e.*, which fault set) is always known by a good destination fault set. Specifically, we do not allow outputs from set $\mathcal{A}$ to be confused with outputs from set $\mathcal{B}$ when used as inputs to any good set $\mathcal{C}$.
>
> 2. Information only flows in the direction of the edge. If handshaking is used, we cannot permit a faulted *destination* set to interfere with a good source set's ability to communicate with other good sets.[1]

These two properties, although seemingly benign, are essential for correct system operation. The communication mechanisms must be carefully engineered to ensure that they are met.

---

[1] We can certainly include handshaking in our data flow but this will tend to obscure its generality. In any case, one will quickly determine that condition (2) has to be met by the handshaking protocol so we are better off assuming that it is met *a priori*.

Figure 2-5 shows how fault sets and interconnection links are designated on an R_0 program. For convenience, fault sets are labeled 1 thru *n*, where *n* is the total number of fault sets in the model. This is an important number to minimize since the engineering and implementation of fail-independent modules may be quite costly.

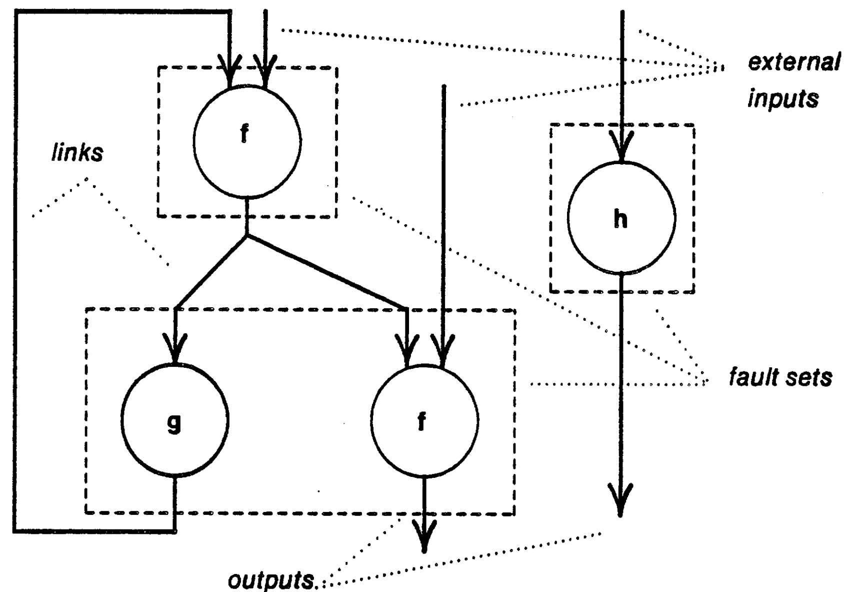


Figure 2-5: Sample R_0 Program

2.3.3 Fault Models

Since two fault sets may influence each other only through data transmitted on interconnection links, a fault model need only be concerned with characterizing the data produced by a faulted set. We must not only capture a faulted set's ability to put out erroneous or spurious tokens, or no tokens at all, but also the possibility of complete corruption of signal protocols.

We shall say that "tokens" transmitted in the latter case are *not well-formed*. Our fault model is as follows:

> **Definition 2-10:** If a fault set $\mathcal{A}$ is *faulted* then we replace the histories of each output edge with an arbitrary sequence of tokens called *token fragments*, $\mathrm{E} = \langle \varepsilon_1, \varepsilon_2, \varepsilon_3, \ldots \rangle$. A fragment is characterized by the rather insidious properties:
>
> 1. $\varepsilon_j \in \mathbf{D} \cup \{\omega\} \quad \forall j.$
>
> 2. $\varepsilon_{j1} \not\sqsubseteq \varepsilon_{j2}$ for any two receivers, 1 and 2.

Property (1) states that the outputs of a faulted source set can be any arbitrary sequence of well-formed tokens. Property (2) captures the characteristics of tokens which are not well-formed. That is, two independent observations (receptions) made of the same token ε_j may differ in their interpretation. This is an essential feature of digital communications which are made through noisy channels with insufficient noise margins. In this case two receivers may extract *different* values from the communicated data.

This ability for a failed transmitter to "lie" to unfailed receivers is the most overlooked yet potentially catastrophic failure mode of all digital systems. As such, further discussion is merited.

Disbelievers protest that "that will never happen. How can two units reading the *same* wire get *different* results?". This ignores some very common failure modes of all physical communication schemes.

Figure 2-6 shows the connection of three fault sets, $\mathcal{A}$, $\mathfrak{B}$, and $\mathfrak{C}$. The interconnection link contains two physical wires, a **data** line and a **clock** line. Assume that the **data** line is sampled whenever the **clock** line makes a low to high transition. If $\mathcal{A}$ fails in such at way that the signals on the **data** line become marginal (*i.e.*, somewhere between the logical 0 voltage and the logical 1 voltage) then $\mathfrak{B}$ and $\mathfrak{C}$ could obtain a different value for any bit. $\mathcal{A}$'s failure might be as simple as a bad solder joint on a pullup resistor or a failing output transistor. In any case, this event seems quite probable and *it must be considered in the analysis of any high reliability system.*

Even if heavy coding is used (*e.g.*, Cyclical Redundancy Checks) there is still the possibility that the receivers will get different results. $\mathfrak{B}$ might be able to recover the signal from the noise while $\mathfrak{C}$ might detect an unrecoverable error. The results are still different.

The need for *source congruence* is probably the leading driver of a correct fault-tolerant implementation. It is also the most overlooked by the engineering community.

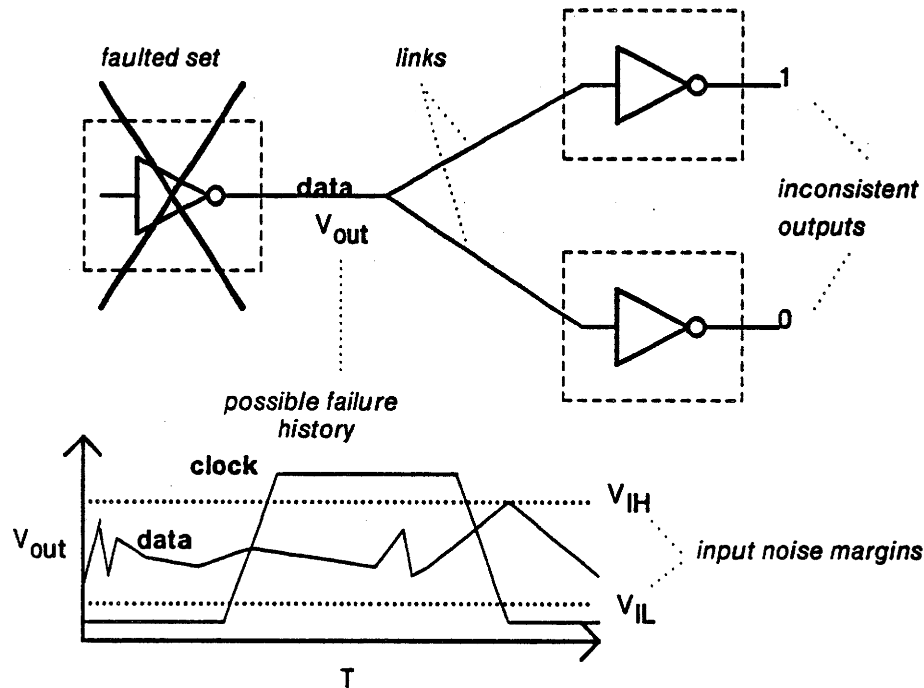


Figure 2-6: Inconsistent Reception of a Token Fragment

2.3.4 Restoring Buffers Revisited

Now that the concept of a token fragment has been defined, the implementation requirements of restoring buffers can be presented. We desire functions which can take a sequence of token fragments as inputs, and output an arbitrary sequence of well-formed tokens. That is, we desire functions that can map

$$D^{\omega} \cup \{\varepsilon\} \rightarrow D^{\omega}.$$

Any function which has this property is called *restoring*.

We remark that the design of a restoring function is totally dependent upon the signal protocols used in the system. It is, however, usually simple to create and often involves just clocking the inputs into latches.[2] Restoring buffers will be used liberally in our solutions.

2.4 Summary

We have presented a very powerful, albeit determinate, language in which to express *simplex source programs*, the programs we shall implement on our fault-tolerant hardware. The language is characterized by recursive connections of monotone and continuous functions which map *histories* of tokens at inputs to histories of tokens at outputs.

The language, R_0, is an extension of L_0, and can express redundant instances of L_0 programs. We then augment R_0 programs with abstraction of those hardware features which have an effect on the fault-tolerance of the system. An augmented R_0 program is called a *model* of the system.

In the absence of failures, the semantics of R_0 and L_0 are identical. We use *fault sets* to model failure-independent hardware modules, and *interconnection links* to model the interconnection of the modules.

A fundamental property of our approach is that fault propagation from bad modules to good is purely through the communication of data from the bad to the good. We argue that a totally general fault model for such data exchanges is characterized by arbitrary sequences of *token fragments*.

The following chapters will show how to construct correct fault-tolerant systems using R_0 models.

---

[2] We don't mean to treat this property lightly. If self-clocking asynchronous protocols are used, restoring functions require a great deal of care in design. It is often argued that such functions can never be fully realized, for even fault-free versions may contain metastable states. While we admit that this may be true, we assert that careful engineering can reduce the probability of such states to approximately the physical failure rate of the device.

Chapter Three

Making Simplex Programs Fault-Tolerant

This chapter presents procedures for transforming simplex L_0 source programs to R_0 redundant programs, and then imposes the necessary implementation restrictions.

The objective is to meet a given reliability specification for the target system. This specification can be given in two forms. The *fail operability* form is an integer specifying the number of independent fault set failures that can be tolerated while still maintaining correct system operation. The second form is the aggregate *system reliability*. This is a real number specifying probability of correct system operation over some interval of time.

We first precisely define what is meant by *correct system operation*. This definition relies on the concepts of *congruent histories* and *congruent sets* of arcs. It is shown that comparison monitoring in the most powerful test for congruence which is possible in R_0. Procedures are developed which transform L_0 programs into R_0 programs and meet a given fail-operability requirement. We show that the problem of *input congruence*, ensuring that all fault-free redundant graphs have identical inputs, is the driver for correct system design. R_0 schemata are presented which guarantee input congruence in the presence of faults. The properties of these schemata are explored in detail.

Finally, we give a graph coloring algorithm for determining the fail-operability, the aggregate system reliability, and for exposing critical failure paths.

3.1 Overview of the L_0 to R_0 Transformation

Our principal approach to improving system reliability is through modular redundancy. *Modules* correspond to fault sets that contain program instances.

Suppose the L_0 program, P, as shown in Figure 3-1, is given along with a fail-operability goal of 1. That is, the implementation should be able to tolerate any single fault and remain

operational. Thus P will have to be triplicated in R_0 where each instance of P is in an independent fault set.

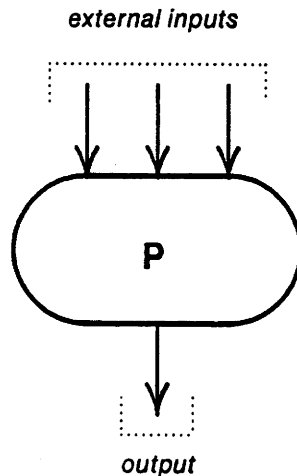


Figure 3-1: Example Source Program

The output devices are given the burden to resolve the redundant outputs from the three instances of P. This resolution will be through bit-for-bit comparisons of the three output lines so it is required that

(C1) Outputs from all fault free instances of P agree bit-for-bit.

Because all R_0 program fragments are continuous and monotone functions, a sufficient condition to satisfy C1 is

(C2) Similar inputs to all fault free instances of P agree bit-for-bit.

We call this the *input congruence* condition. It will become clear that ensuring input congruence is the leading driver in correct fault-tolerant system design. Unfailed outputs must also be *correct*, where correctness is relative to the original L_0 program. As shown in Figure 3-2, the external inputs must be delivered to each R_0 instance of P such that C2 is satisfied and

(C3) The inputs applied to any fault free instance of P must be valid inputs for P in L_0.

It is implicitly assumed that a full set of unfailed external inputs form a valid input set for P,

and that replacing certain inputs by arbitrary sequences of well-formed tokens also forms valid input sets. That is, P should contain algorithms for identifying or tolerating a faulty external input. This property is captured in P's robust input direction set V_P.

C2 requires all instances of P in R_0 to have the *same* inputs while C3 requires this input set to be *meaningful*. We satisfy both conditions in R_0, by requiring all inputs to all instances of P to be identical to the external inputs to the program. The external inputs themselves are subject to failure but, as previously noted, it is P's responsibility to operate correctly in the presence of external input failures. We only guarantee to deliver these inputs, *what ever they may be*, identically to all fault-free instances of P.

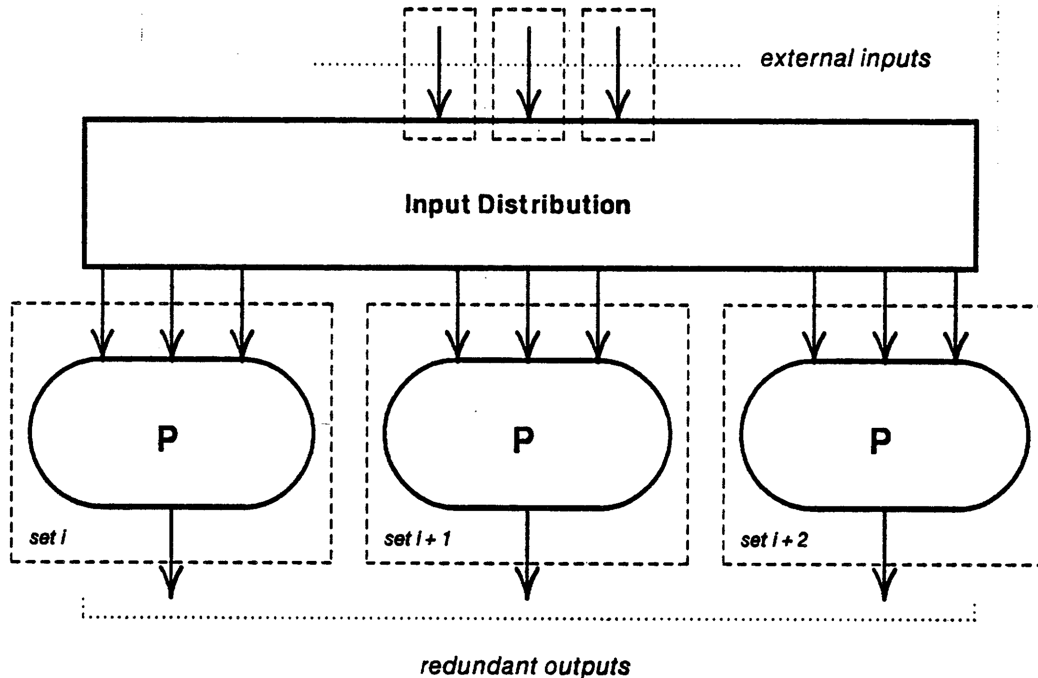


Figure 3-2: General Topology for Triplex P

Figure 3-3 illustrates an initial solution. Here the inputs are simply distributed to the independent instances of P. The problem is in the failure of an external input. In the simplex

assumptions of L_0, P could still produce correct[3] results even in the presence of selected input failures. Now when we fail an input each P will produce *correct* results but the results may not be the *same* since a failed external input, being modeled by an arbitrary history of token fragments may be interpreted differently by each instance of P.

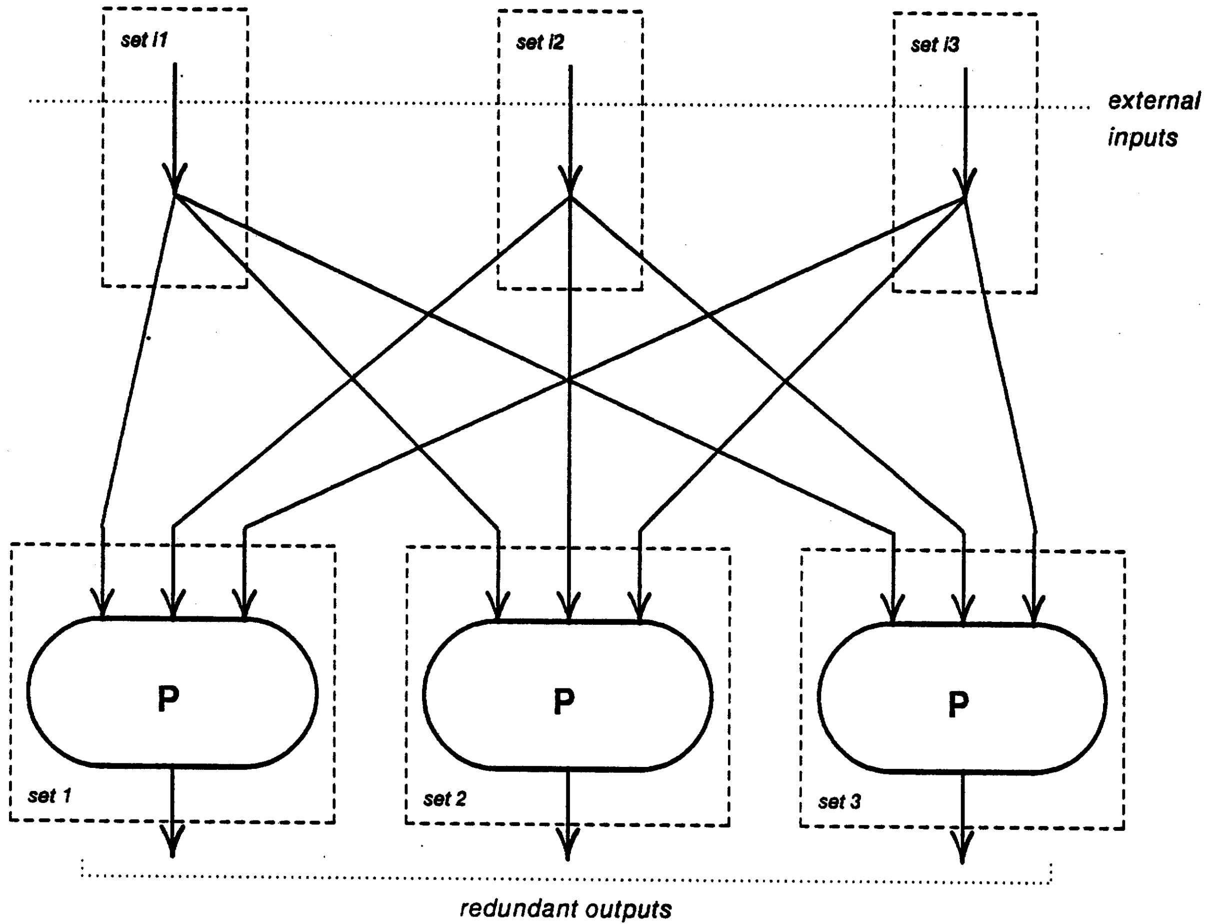


Figure 3-3: Hazardous Distribution of External Inputs

Clearly the problem with an external input failure is not that it can deliver an arbitrary

[3] This doesn't mean the results will be the *same* as if the input is operating correctly: the outputs of P are dependent upon the inputs. In the event of an input failure that P can tolerate, the outputs may change somewhat but they are still acceptable and so P operates correctly.

sequence of tokens, but that the value of these sequences is dependent upon the receiver. This happens when signal protocols are violated and the tokens are not well-formed, as motivated in Chapter 2. To solve this problem we need functions which, when fault-free, produce only well-formed tokens. Restoring buffers have this capability.

Our solution involves distributing each external input to three restoring buffers. The outputs of the buffers are exchanged and independently resolved at the input to each instance of **P**.

Figure 3-4 illustrates the use of restoring buffers and voters to consistently replicate an external input. The graph is similarly replicated for the remaining two external inputs. This triplex realization of P can tolerate any single failure, and still yield at least two correct and identical outputs.

The operators within the box in Figure 3-4 are an example of a *congruence schema*. The following sections will develop machinery for the creation and analysis of congruence schemata, as these are the distinguishing features of a correct fault-tolerant system design.

3.2 Bit-for-Bit Agreement and Congruence

Bit-for-bit agreement among redundant outputs is an imposing goal of modular redundant designs. Formulations of redundant systems where only approximate agreement among redundant outputs is required makes it substantially more difficult, if not impossible, for output devices to generate correct commands. In our case, any single bit disagreement between two redundant outputs will always imply a fault in one, or both, sources. Combinations of pairwise comparisons of outputs will be used to isolate and mask faults. *Isolating* a fault means to correctly determine which fault set has faulted. *Masking* a fault stops the bad data produced by a faulted set from propagating through the system. Selecting the median value of three signals, for example, will only mask a bad signal. Multiple pairwise comparisons, *voting*, of the same signals, if applicable, will both mask and isolate the bad signal.

This section formalizes the concepts of bit-for-bit agreement and pairwise comparison tests in terms of $\mathbf{R_0}$.

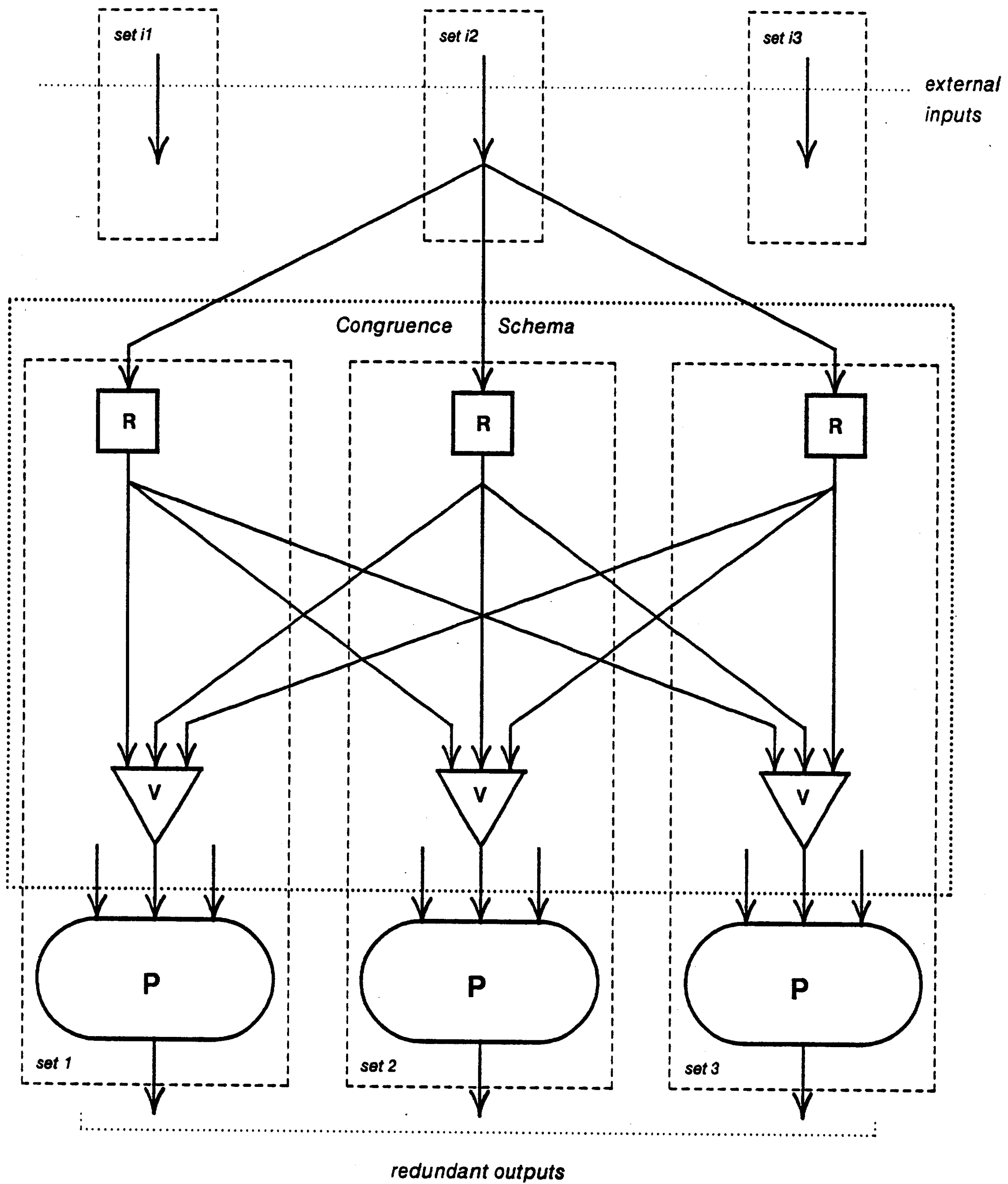


Figure 3-4: A Correct Triplex Version of **P**

3.2.1 Congruent Sets

Here we precisely define bit-for-bit agreement in the semantics of R_0. We shall say that two histories are *congruent* if they are the same point in D^ω. A set of edges form a *congruent set* if the corresponding histories are all congruent. Formally,

Definition 3-1: Histories $\mathbf{a},\mathbf{b} \in D^\omega$ are *congruent*, $\mathbf{a} \equiv \mathbf{b}$, **iff**

$\mathbf{a} \sqsubseteq \mathbf{b}$ **and** $\mathbf{b} \sqsubseteq \mathbf{a}$.

Definition 3-2: Edges $e_1, e_2, \ldots, e_n$ with histories $\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$ form a *congruent set* **iff**

$\mathbf{e_i} \equiv \mathbf{e_j} \qquad \forall i,j.$

We will sometimes talk of a *congruent set of histories* which we mean to read: *the edges corresponding to these histories form a congruent set.*

3.2.2 Congruence Tests

Since we shall use pairwise comparison to isolate and mask faults it is important to precisely understand the properties of this test. We wish to destroy the notion that such comparisons and associated majority logics are heuristic. Rather, they form a rigorous basis for provably correct fault-tolerant systems. The pairwise comparison test of two histories is defined as the weak equality function for R_0.

Definition 3-3: A *pairwise comparison test* of two histories $\mathbf{a} = (a_1, a_2, \ldots)$, $\mathbf{b} = (b_1, b_2, \ldots)$ is the weak equality function on each element: $a_i = b_i$ is

1. ω_i if a_i or b_i is ω
2. **true**$_i$ if a_i is b_i
3. **false**$_i$ otherwise.

We write $\mathbf{a} \neq \mathbf{b}$ if $(a_i = b_i)$ is **false**$_i$ for any i and $\mathbf{a} = \mathbf{b}$ otherwise.

We now prove that a pairwise comparison test is the strongest test for congruence that can be formulated in R_0.

Theorem 3-1: For any two histories **a** and **b**:

1. $\mathbf{a} \neq \mathbf{b} \quad \Rightarrow \quad \mathbf{a} \not\equiv \mathbf{b}$

2. There is no R_0 schema which decides $\mathbf{a} \equiv \mathbf{b}$.

Proof:

1. $a \neq b \Rightarrow \exists\, a_i, b_i \neq \omega \ni a_i \not\sqsubseteq b_i \Rightarrow a \not\equiv b$.

2. *By contradiction.* Suppose there exists a continuous and monotone function $F(a,b)$ which decides $a \equiv b$. That is, F generates $\mathbf{false}_i$ whenever $a_i \not\sqsubseteq b_i$ or $b_i \not\sqsubseteq a_i$ and $\mathbf{true}_i$ otherwise. Let $a_1 = (1, 1, \ldots)$, $a_\omega = (\omega, \omega, \ldots)$ then $\langle a_\omega, a_\omega \rangle \sqsubseteq \langle a_\omega, a_1 \rangle \Rightarrow F(a_\omega, a_\omega) \sqsubseteq F(a_\omega, a_1)$ since F is monotone.

However, $F(a_\omega, a_\omega) = \langle \mathbf{true}, \mathbf{true}, \ldots \rangle$ and $F(a_\omega, a_1) = \langle \mathbf{false}, \mathbf{false}, \ldots \rangle$. And $\langle \mathbf{true}, \mathbf{true}, \ldots \rangle \not\sqsubseteq \langle \mathbf{false}, \mathbf{false}, \ldots \rangle$.

This implies that F is not monotone. Therefore, no such function exists in R_0.
□

In some sense this exposes an inherent weakness in the seemingly powerful formulation of R_0. That is, R_0 programs lack the ability to detect missing tokens. Detection requires a nondeterministic merge, which is not a function in R_0. Such merges will be allowed in R_1, which will be presented in the next chapter. When nondeterministic operators are permitted, the problem of creating congruent sets is considerably more difficult, and a total ordering of system events, *synchronization*, is required.

3.3 Testing R_0 Models for Correctness

The ultimate measure of the systems which we produce is a probability of correct operation. *Correctness* of an R_0 program is defined in terms of its outputs, and whether the redundant outputs are congruent with respect to the same edges in a perfect L_0 simplex program.

3.3.1 Input Robustness and Correct Operation of R_0 Programs

When we transform an L_0 program to R_0 we group the inputs into fault sets which correspond to the implementation. Any input from a good fault set is assumed to lie in the valid input space of the L_0 program. Correct operation of an R_0 program is readily defined.

> **Definition 3-4:** An R_0 implementation of an L_0 program P is *correctly operating* if a majority of redundant outputs form a congruent set with the outputs of a perfect simplex version of P.

That is, all fault-free instances of P must be driven with inputs that are elements of Υ_P and

all fault-free redundant outputs must be identical. We can assure congruent outputs of all fault free instances of P by providing congruent inputs. We can't test for membership to $\mathcal{V}_P$ directly, but we can use the fact that if all but inputs *j, k, l,...* are congruent with fault free external inputs and $\langle j, k, l,..\rangle \in V_P$ then the inputs are in the valid input space $\mathcal{V}_P$.

3.3.2 Fail-Operate Specification

The fail-operability specification is now easy to precisely formulate given the definition of correct system operation.

> **Definition 3-5:** An R_0 implementation of the program P with n independent fault sets is *fail-op*^{f} if for any f fault sets with output edges replaced by arbitrary sequences of token fragments, the system continues correct operation.

In our models, fault sets can influence one another only through the exchange of data on output edges, and we suggest that arbitrary sequences of token fragments completely characterize the range of these failures. Our definition of fail-operability describes precisely the number of independent fault set failures that can be tolerated by a given implementation. Thus we believe that the above definition completely captures the meaning of "fault tolerance". Later, we shall give a graph coloring procedure for determining the fail-operability figure for any R_0 program.

3.3.3 Aggregate System Reliability

The primary reason that a system should tolerate faults is that individual component or module reliability is insufficient to meet a system reliability specification. It is of utmost importance, then, that we are able to extract an *aggregate system reliability* number for our implementation.

> **Theorem 3-2:** Given an R_0 program with n fault sets and failure rates $\lambda_1, \lambda_2,..., \lambda_n$. The aggregate system failure rate λ_{sys} can be determined by the following algorithm.
>
> Initialize $\lambda_{sys} := 0$.
>
> For all f: $0 \leq f \leq n$ test the system for correct operation against failures of each combination of f units. If the system does not correctly operate then update $\lambda_{sys} := \lambda_{sys} + \lambda_{j1}\lambda_{j2}...\lambda_{jf}(1-\lambda_{k1})(1-\lambda_{k2})...(1-\lambda_{k(n-f)})$ where the λ_js correspond to each of the f fault sets that are assumed failed and the λ_ks correspond to the remaining n-f fault free sets.
>
> **Proof:** Unique combinations of fault set failures form disjoint and exhaustive

sets. (There are 2^n of them.) Therefore the probability of any combination occurring in a set of combinations is the sum of the probability of occurance of each combination in the set. In our case the set of interest is defined by the attribute of correct system operation.

Since the underlying events are all independent, the probability of occurance of any combination is the product of the unreliability, λ, of each failed set times the product of the reliability, $1-\lambda$, of the remaining fault free sets.
□

Note that dominating terms in the sum reveal the critical failure paths of the system. The procedure, although correct, is much more exhaustive than it needs to be, given the domination of these terms. It is possible to extract these paths to obtain a good approximation of the unreliability of the system. Our intent here is to show that our model provides unambiguous information for extracting the aggregate system reliability.

It is possible to lend a probabilistic structure to V_P which describes the *coverage* of input failures by P. This would include the probability of *misdetection* and *false alarms*. Such additional information is easily incorporated into our model.

3.4 Required Level of Redundancy

It is important to determine the number of independent instances of a program[4] which are needed to meet a given fail-operability goal. Each instance is "expensive" in that it represents real computation and physically separate pieces of hardware. Thus we would also like to know what price we are paying for the modular redundancy approach. Are there any other approaches that yield analytically correct results but with less hardware?

By the definition of system correctness, a majority of redundant outputs must be congruent and correct. An arbitrary fault can cause an output to not be congruent and, more importantly, faulty outputs can "collude" to form congruent sets of their own. So if input congruence and correctness are satisfied for each fault-free unit, to tolerate f independent faults requires, in our case,

$$p \geq 2f + 1,$$

[4]Instances are independent only if they are implemented in separate fault sets.

where p is the number of independent instances of the L_0 source program **P**. Clearly, even in the presence of f faults, a majority of fault-free units are producing congruent outputs. Thus application of pairwise comparison tests to all pairs of outputs will always resolve a correct output.

This also implies that n, the total number of fault sets is bounded by

$$n \geq p \geq 2f + 1$$

If $n > p$ then the remaining $n - p$ fault sets are not directly associated with the computation of **P**, but with some other function or with the external inputs. The p sets associated with the computation of **P** are the "processors". Thus we conclude that we need a minimum of $p = 2f + 1$ independent sites computing **P**.

This constraint may seem excessive. However, the following result from testing theory [18] shows that this is indeed a minimum for the correct isolation of f failures *using any test*.

> **Theorem 3-3:** [Preparta *et al.*, 1967] Any system S composed of n independent units can correctly isolate f simultaneous faulty units by mutual testing only when $n \geq 2f + 1$.

Mutual testing is where unit A can apply a test to unit B to diagnose faults in B. If A is itself faulty then the result of the test is arbitrary. In our case, we say that a unit is faulty when its output ceases to be congruent with other good units. Thus our test, the weak equality test of two histories **a** = **b**, is a reflexive test. Unit A tests unit B through the same test as B tests A. In any case, this theorem establishes the bound on p to be at least $2f + 1$. Since we can meet this bound strictly, majority logics are in fact optimal in terms of the number of cotesting units required.

Unfortunately, the number of independent fault sets, n, required to maintain congruent inputs to all fault-free instances of **P** is strictly greater than p. This is not a result of testing theory, however. One quickly finds that the hardware, *e.g.*, testing theory approach to fault-tolerant system pays little attention to the types of algorithms that will be implemented. Similarly, the algorithmic approach often misrepresents or ignores important aspects of hardware design.

Maintaining congruent input sets falls in the "algorithmic" side of reliable system research, and is often called the *Byzantine Generals Problem*. Results here [4] [5] show that the number of independent processors, p, required to maintain congruent inputs to all fault-free processors is $p \geq 3f + 1$.

Theorem 3-4: [Pease *et al.*, 1980] A system S composed of n independent units can ensure congruent edges in all fault-free units only when $n \geq 3f + 1$.

The two results seem to be contradictory, however, both are correct within their assumptions. The elements of the theories which apply to our model reduce to

$$p \geq 2f + 1$$

$$n \geq 3f + 1$$

Thus p independent instances of P are required to correctly isolate any f faults in the computation of P. The remaining $n - p$ fault sets are required to maintain congruent inputs to p processing sites. We note that most fault-tolerant systems which employ congruent processing have $2f + 1$ processing sites but rarely have the $3f + 1$ fault sets required for correct distribution of external inputs. Therefore, these systems contain failure modes whereby the occurance of f simultaneous faults can cause a complete system failure.

3.5 R_0 Schemata For Congruent Inputs

In this section R_0 schemata are developed which guarantee congruent distribution of a simplex input in the presence of a given number of faults. As previously illustrated, the congruent distribution of a single *simplex* source of data is the driving requirement for correct fault-tolerant system operation. In particular, when the simplex source is itself faulted, inconsistent views of the value of the source are possible unless some congruence schema is employed.

Definition 3-6: C_f^p is a p-input, p-output R_0 *congruence schema* if distribution of a simplex source to all p inputs guarantees a congruent representation of the input at all outputs in the presence of f independent faults, including the simplex source.

Figure 3-5 gives a schematic representation of a C_f^p congruence schema.

For example, the solution presented in the overview, Figure 3-4, constructed a C_1^3 congruence schema which successfully delivers an external input to the three instances of P in the presence of any single failure.

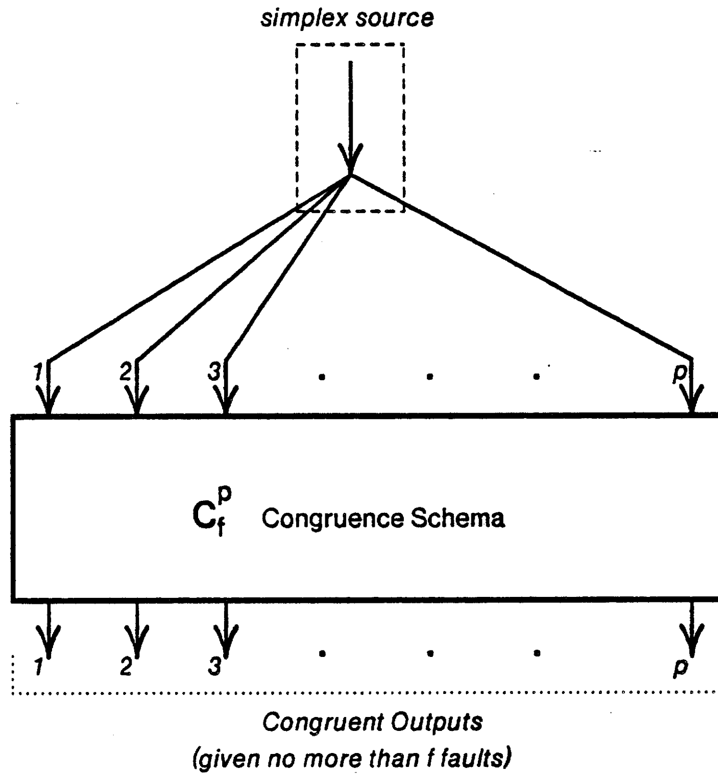


Figure 3-5: Schematic for a C_f^p Congruence Schema

3.5.1 C_1^p Schemata

A C_1^p schema can tolerate a simplex source failure or a set of internal, *i.e.*, inside C_1^p, failures and still deliver congruent results to the p outputs. The properties and construction of such schemata are important in that, in this section, we develop C_f^p schemata, those which deliver p congruent outputs in the presence of f faults, from recursive constructions of C_1^p schemata.

Figure 3-6 gives the canonical implementation of a C_1^p schema.

The numbers by the input and output arcs designate the fault sets to which the associated operator, a restoring buffer or voter, belongs.

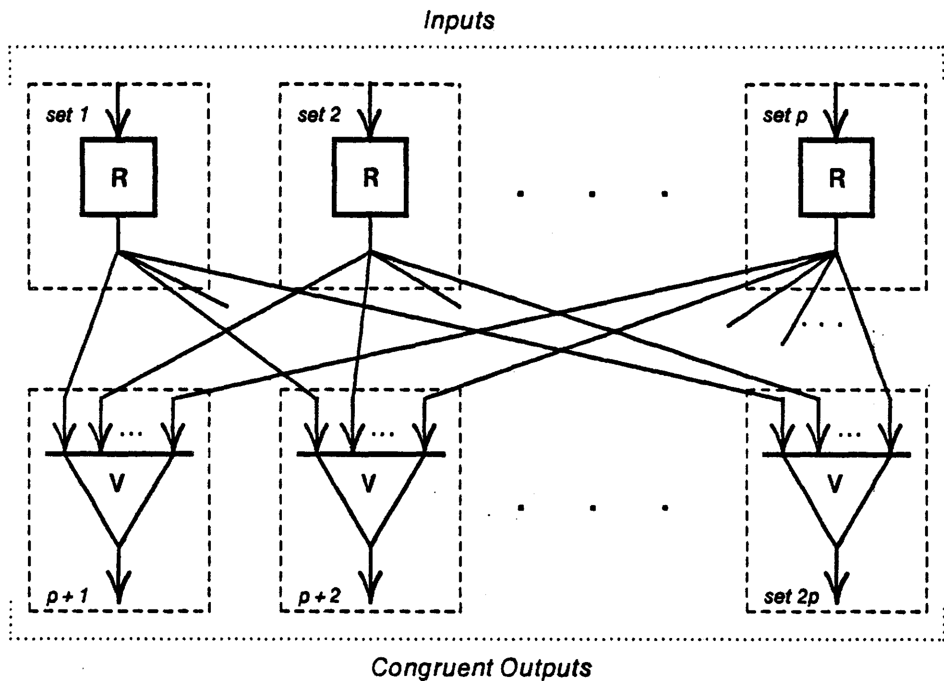


Figure 3-6: Canonical C_{γ}^{ρ} Schema

Because each output is used to drive an instance of P, the output fault sets, *i.e.,* the fault sets corresponding to the voters, are usually the same as the instance of P to which they are connected. When the source is originating from a *different* set than any of the output sets, fault sets of input restoring buffers may be paired with fault sets of the output buffers. We call this, see Figure 3-7, a *minimal* realization of the schema, because the total number of fault sets required is a minimum with respect to theorems 3-3 and 3-4. We now give several important properties of C_{γ}^{ρ} schemata, either canonical or minimal.

Property 3-5: All C_{γ}^{ρ} schemata are monotone and continuous.

This trivially follows from the fact that both restoring nodes and voters are monotone and continuous functions. Direct consequences of 3-5 are

Property 3-6: A C_{γ}^{ρ} schema always produces outputs when no input is ω.

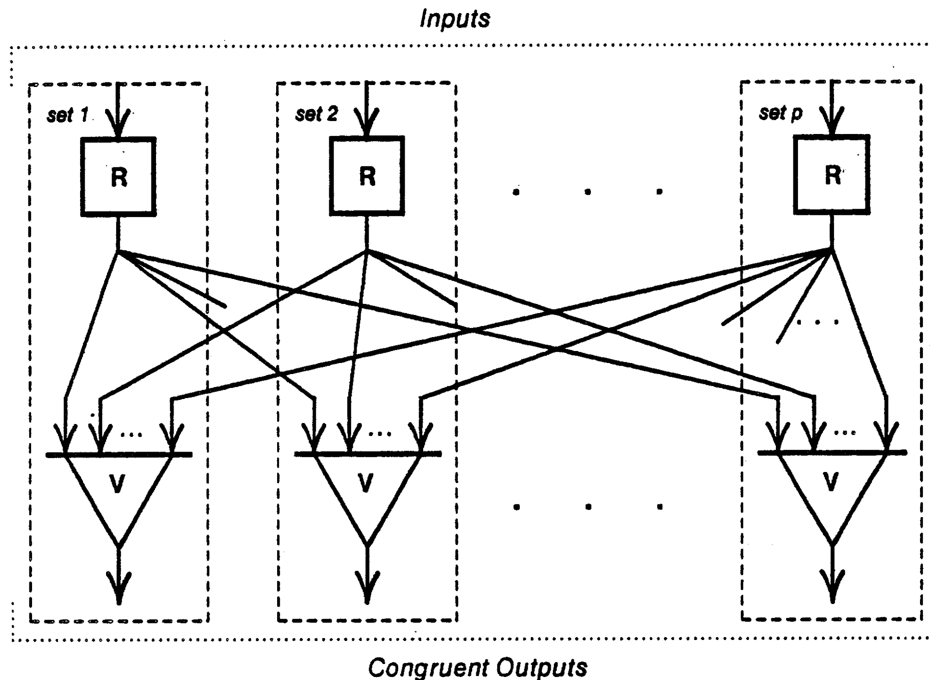


Figure 3-7: Minimal C_f^p Schema Assuming Independent Source

Property 3-7: If a majority of the p inputs are ω_i for some i then all fault free outputs are ω_i.

These properties will be exploited in the next chapter but we note that property 3-7 alludes to the most serious limitation of R_0. That is, it is impossible to detect the divergence of a simplex source. Clearly, the divergence of a source is quite a common failure mode and almost all failure detection and isolation algorithms for external inputs require a "time-out" test. This test is not available in R_0 because such tests are nondeterministic and therefore not functions of histories of inputs to histories of outputs. However, the structures developed here will be extensively exploited in the construction of congruence algorithms for nondeterministic systems.

The following properties are extremely useful in constructing general C_f^p schemata. They apply to both canonical and minimal implementations. The positive ones are:

Property 3-8: A C_T^p schema will always produce congruent histories at fault-free outputs whenever there are no more than (p div 2) - 1 failed restoring buffers and the inputs to no less than (p div 2) + 1 of the fault free buffers form a congruent set.

In other words, this property guarantees congruent outputs whenever the inputs are congruent and a majority of the restoring buffers are fault-free. The other interesting positive property is when token fragments are applied to all inputs, as in the case of a failed simplex source.

Property 3-9: A C_T^p schema will always produce congruent histories at fault-free outputs whenever all restoring buffers are fault-free. This is true for *any* set of inputs including arbitrary streams of token fragments.

The central negative property of C_T^p schemata is:

Property 3-10: A C_T^p schema may fail to produce congruent histories at fault-free outputs whenever a majority of the inputs do not form a congruent set and any single restoring buffer is failed.

This the negative dual of property 3-9. That is, in the presence of a simplex source failure, congruent outputs are assured *if and only if* all restoring buffers are fault-free. Figure 3-8 gives an example of property 3-10 for a C_T^3 schema. In this example the simplex source is failed along with one of the restoring buffers. Since the outputs of any bad fault set can be replaced with an arbitrary sequence of tokens, we give such sequences which can cause complete disagreement at all fault-free outputs.

We therefore conclude the following important design rule.

Lemma 3-11: The fault set of a simplex source driving a C_T^p schema can never be the same as any restoring buffer.

The important design consequence is when external inputs are brought directly to one of the p processing sites. This would correspond to the standard sort of input device hooked up to a computer. In this case the computer which reads the device and sources the information to the rest of the system *cannot* participate in the congruence schema.[5] Whenever this technique is used, as is often the case, a failure of the source computer will imply a failure of the external input. If the processor failure rate is much higher than that of the source, then the source is often routed to multiple processors. This technique is called *cross-strapping* and requires that each strap be treated as a different, but related, external input. Such

[5]Of course, it will want one of the outputs from the schema and thus the associated voter will most likely be implemented by the sourcing processor.

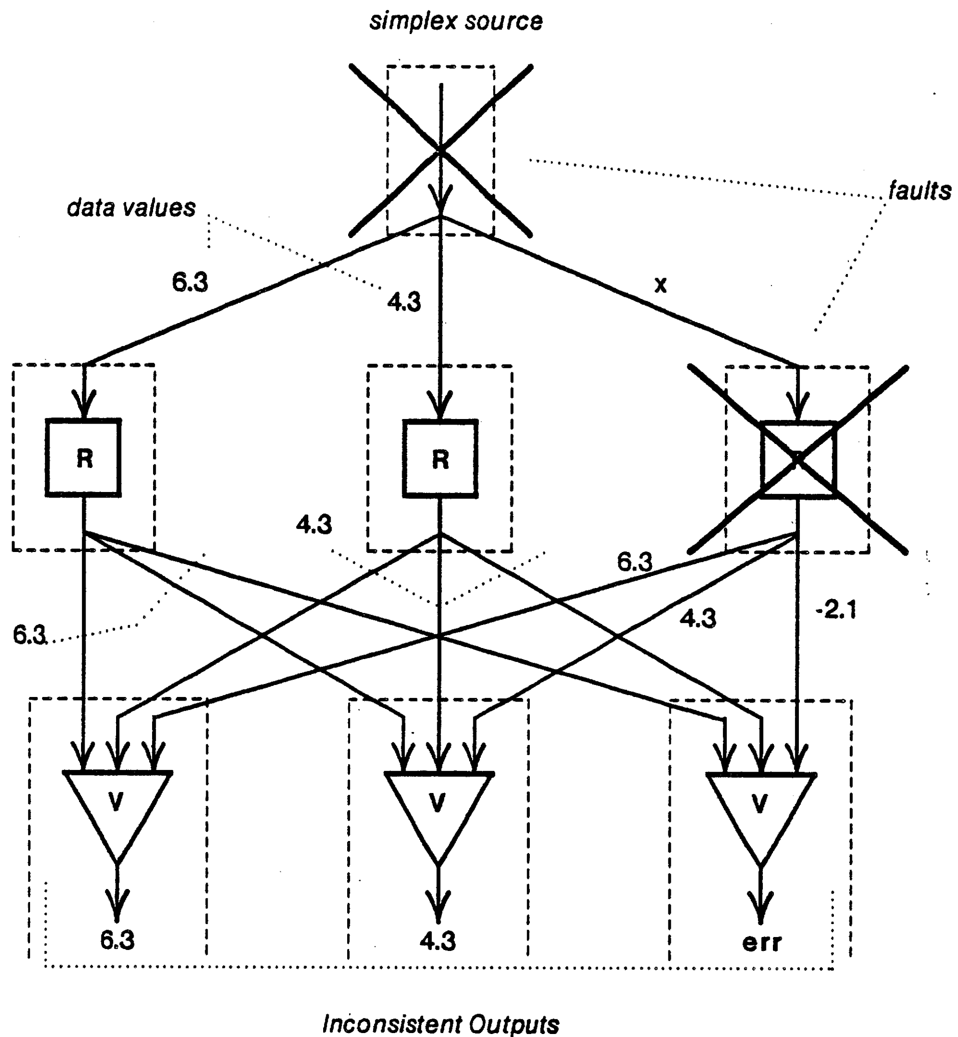


Figure 3-8: Failure of a C_1^3 Schema with Two Faults

models for cross-strapped sensors will be given in Chapter 5.

3.5.2 C_f^p Congruence Schemata

Here we present recursive algorithms for generating C_f^p schemata from C_{f-1}^p schemata. We expose an inherent tradeoff between the number of independent fault sets required to implement a schema versus the number of exchanges of information which must occur.

Minimum Fault Sets. In this recursion, the number of fault sets are kept at the theoretical

minimum. To keep the results as clear as possible we assume that p is a minimum, $p = 2f + 1$. Thus the total fault sets will be $n = 3f + 1$. Consider the simplex source to be in a unique fault set, although possibly the same as one of the output fault sets. The following algorithm constructs a correct congruence schema with $3f$ fault sets.

Theorem 3-12: A C_f^p congruence schema can be constructed from C_{f-1}^p schemata with a minimum number of independent fault sets as follows (see Figure 3-9).

Let each C_{f-1}^p schema be minimal and denote the $3f - 3$ fault sets associated with the ith schema as $i_0, i_1, \ldots, i_{(3f-4)}$. Cascade $3f$ C_{f-1}^p schemata where the outputs of the schema i are fed to the inputs of schema $i + 1$. The inputs to schema 0 are the inputs to C_f^p while the outputs of schema $3f - 1$ are the outputs of C_f^p. Assign the $3f$ fault sets so schema i uses sets i to $(i + 3f - 4) \bmod 3f$.

Proof: *By Induction.* Our basis step is property 3-8 and property 3-9. This defines C_1^p.

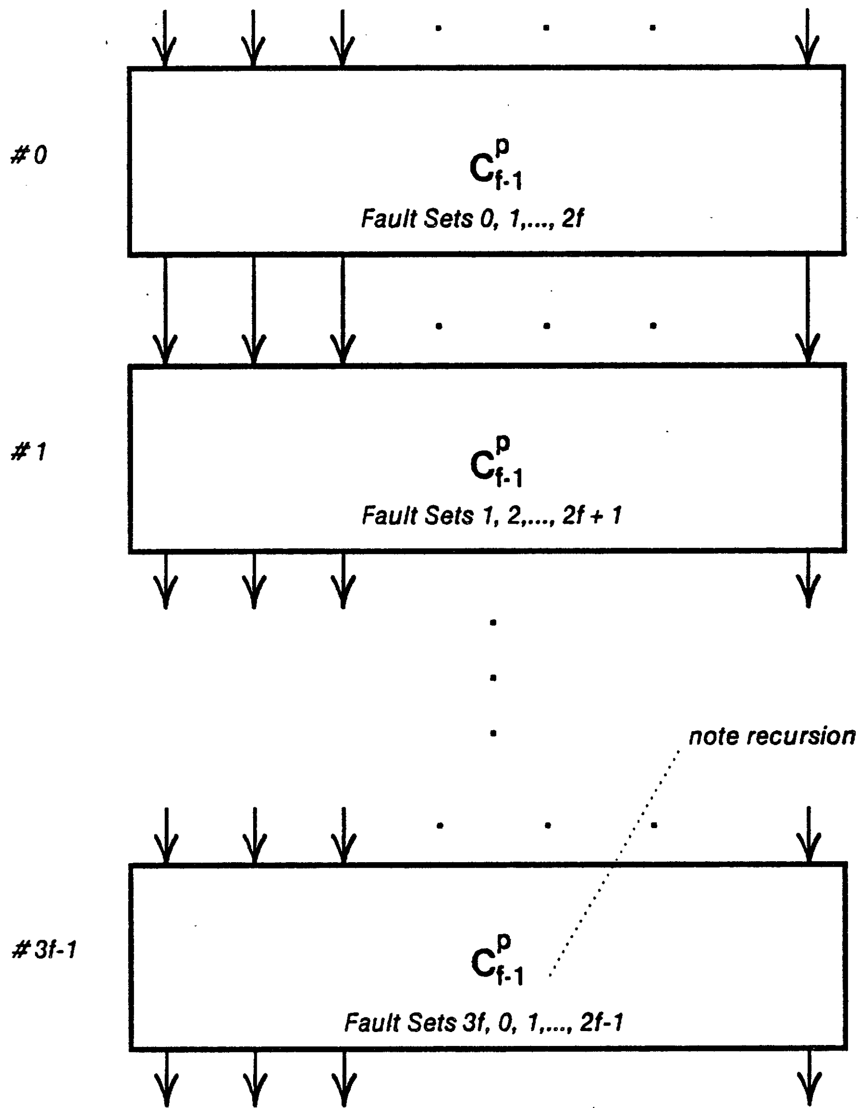
Suppose a C_{f-1}^p schema has the properties; (1) Congruent inputs yield congruent outputs with p *div* $2 - 1$, or fewer, internal faults; and (2) Fragmentary inputs yield congruent outputs with $f - 2$, or fewer, internal faults. Then our theorem holds for C_f^p, $p \geq 2f + 1$, if

(1) *Congruent inputs yield congruent outputs with p div 2 - 1, or fewer internal faults.* This is easy to show and corresponds to the case where the simplex source is fault free. Because p *div* $2 - 1 \geq f$ then C_{f-1}^p stages will produce congruent outputs with congruent inputs even with all f faults concentrated in one stage. Therefore C_f^p will produce congruent outputs from congruent inputs in the presence of any f internal faults.

(2) *Fragmentary inputs yield congruent outputs with f - 1, or fewer internal faults.* If all $f - 1$ faults are concentrated in a C_{f-1}^p stage where the inputs are not congruent then outputs may also not be congruent since a C_{f-1}^p can guarantee congruent outputs from incongruent inputs with at most $f - 2$ internal faults. However, if the inputs are congruent then the $f - 1$ faults can be tolerated and the outputs will be congruent. It is clear from the permutation strategy that, a stage with $f - 1$ internal faults will always be either preceded or followed by a stage with not more than $f - 2$ internal faults. Thus, congruent outputs are assured.

□

Although this construction requires a minimum number of independent fault sets, the amount of information exchanged between fault sets grows exponentially. Lamport [5]gives an equivalent minimum fault set construction with slightly less communication since he allows p to decrease at every stage. A bound for the amount of information to exchange for a minimum realization, however, has not been established. In any case the number of exchanges (*i.e.*, number of C_1 operations grows something like $p!/(p - f)!$.



$3f + 1$ Total fault sets
$3^f f!$ C_1 exchanges

Figure 3-9: Minimum Fault Set Implementation of C^p_f

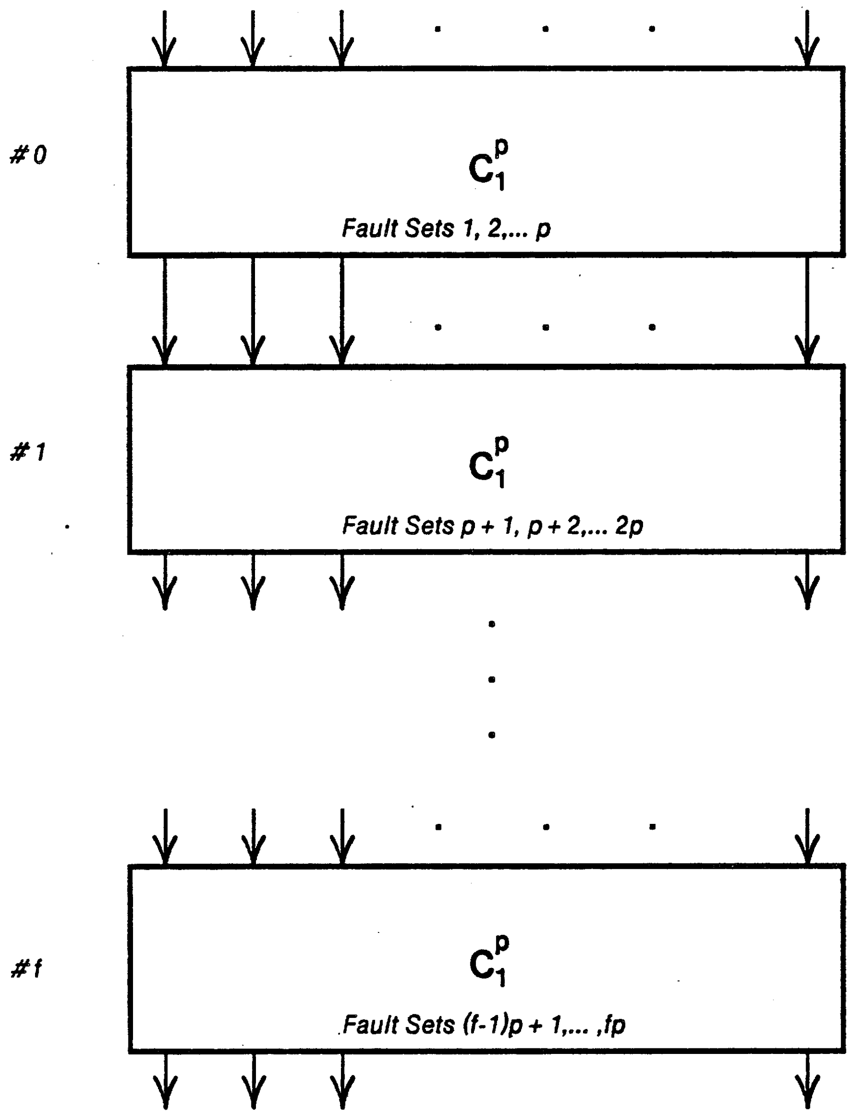
Minimum Stages. The number of C_1 exchanges (stages) is extremely high in the minimum fault set case. Researchers in this area have been driven by the minimum fault set requirement under the assumption that each set represented a "computer", in the sense of being able to support P. This is not necessary in our case for we have shown that the total number of processors required is only $2f + 1$ whereas a minimum fault set approach views each fault set as a processor and requires $3f + 1$ processors. We still need $3f + 1$ fault sets for congruence maintenance but the remaining f sets need only be restoring buffers, so the implementation cost is not nearly as high. If we are willing to add more fault sets than the minimum required, the system takes on much more regularity and the amount information exchanged is substantially reduced.

> **Theorem 3-13:** A C_f^p schema can be constructed by cascading f independent C_1^p schemata (see Figure 3-10). Independence means that no two restoring buffers share a fault set.
>
> **Proof:** Each C_1^p schema can tolerate f restoring buffer faults when their inputs are congruent. If the simplex source is failed then fragmentary inputs are only made congruent by a stage with no faulty restoring buffers. We are guaranteed at least one C_1^p with no restoring buffer faults since $f - 1$ faults are distributed over f independent stages. All stages following the fault-free one will give congruent outputs even with up to f restoring buffer failures.

Note that the number of extra fault sets grows like $pf = 2f_2 + f$ while the number of exchanges only grows like f. Compare this to the minimum fault set implementation that requires a total of $3f + 1$ fault sets, but performs $3^f f!$ exchanges. Also note that the additional sets for the minimum stage construction are quite simple, *e.g.*, single chip, collections of restoring buffers and voters. There is also the advantage that the stages form a physical pipeline and may integrate very well with packet routing techniques for interconnecting processing sites.

3.5.3 Remarks

It should be clear that congruence maintenance is an extremely demanding function for a fault-tolerant system, especially one that can tolerate more than a single failure without reconfiguration. Since *all simplex sources must be distributed by congruence schemata* the system design will be heavily influenced by these burdensome requirements. Practical solutions for real-time systems will need special hardware support in the communications system to support such transactions. It is reasonable to expect that our fault-tolerant design



$2f^f + f$	*Total fault sets*
f	*C_1 exchanges*

Figure 3-10: Minimum Stage C_f^p

methodology, being founded in data flow analysis, will implement well on systems which are operated on data flow principles. Especially since we share the mutual problem of efficiently routing large quantities of information to physically dispersed and independent processing sites.

3.6 Sequential Faults

We have implicitly assumed that all faults, up to f, occur simultaneously. While this gives the most robust solution, it is also a reason why the number of exchanges and fault set requirements grow so rapidly with f.

Sometimes it is possible to tolerate *sequential* faults by using one level of exchange and *reconfiguring* after a fault is detected. This reconfiguration consists, quite simply, of ignoring faulty units. Examination of the $f = 2$ and higher algorithms reveals that their complexity is due to not knowing the placement of the faults, and worse case conditions must be accounted. Thus, sequential systems isolate each fault as it occurs, and then mask the bad unit. If the isolation and reconfiguration can occur fast enough, the system need only be able to tolerate a single unknown fault at a time. In this case p grows like $2 + f$ and $n \geq p + 1$.

There is a penalty, of course. If a fault is not located before a second fault occurs then the whole system is likely to fail. The difficulty is uncovering *latent* faults in the system. These are faults which have occurred but have not been detected since the failure mode has not yet been excited by the normal operation of the system. Intermittent faults are also extremely difficult to both detect and correctly isolate. Thus, sequential systems, although very attractive engineering solutions require special consideration and do not yield the level of confidence of systems designed on simultaneous fault models.

Sequential models can be analyzed under our theory by simply considering the system to be a sequence of $f = 1$ models. The probability of correct transition to the next model after a fault can represent the coverage of fault isolation tests. The case where the transition is not made means just keeping the current model and determining the effects, usually fatal, of a second error. Sequential machines must still provide a majority of congruent and correct outputs.

3.7 Graph Coloring Algorithm

This section provides some very simple tools for coloring the interconnection links of an R_0 graph to determine the effects of a given combination of failures.

The interconnection links are given two colors. All sequences of well-formed tokens are assigned a shade of blue, one shade corresponding to each point in D^{ω}, and all sequences of token fragments, *e.g.*, those links sourced from failed sets, are colored red. Red sequences are irreproducible and dependent upon the receiver so we only need one shade. Note the following properties:

> **Property 3-14:** A fault-free restoring buffer transforms a blue input edge to a blue output edge of the same shade. A red input edge is transformed to a blue output edge of arbitrary shade.
>
> **Property 3-15:** A fault-free voter yields a blue output edge of the same shade as a majority of same shaded input edges. If no majority exists, the output is colored dark blue corresponding to the **error** token, unless *any* input is red, in which case we color the output red.

The voter property is somewhat conservative, but the exceptions detract from the simplicity of the algorithm. The following algorithm always rejects an incorrect graph under a given combination of assumed faults:

> Color all links sourced from faulted sets red. Color all fault-free external inputs unique shades of blue.
>
> Continue to color the graph according to the properties above. Make sure to assign the outputs of each restoring buffer driven by a red edge a *unique* shade of blue.
>
> For each instance of P if any input is red, color the output red. If the inputs colored dark blue do not form an entry in V_P, color its output red. For all other instances assign unique values of blue to the output edges, except that two fault-free instances with identical input shading should have identical output shading.
>
> If a majority of redundant outputs have the same shade of blue, then the system is guaranteed to operate correctly with respect to this output.

The correctness of the algorithm follows directly from the coloring properties above and the fact that R_0 schemata are pure functions which map histories of inputs to histories of outputs and therefore may be independently colored. The most valuable application of this tool is establishing the effects of failed simplex sources, since these conditions are clearly the litmus test for any fault-tolerant design.

Chapter Four

Nondeterminism and Synchronization

The fundamental shortcoming of L_0 is the inability to detect the divergence of a simplex source. One often wants to place a "reasonable" amount of time to wait for a token from a source. If this time is exceeded then the source is considered *timed-out* and possibly faulty.

This test, a *time-out test*, is clearly nondeterministic; the same test made at different space or time points could yield different results. For instance, a particularly malicious failure of the source could deliver a token arbitrarily close to its time-out limit, and thus induce certain sites to correctly receive the token while others would declare it timed-out.

The correct generation of a time-out test has three features:

1. Fault free sources will never time-out.

2. The results of the test must be generated in a bounded amount of time.

3. The results of the test must be consistent at all redundant sites.

If such a time-out test can be correctly performed on a system then the implementation of general nondeterministic functions is straight-forward. A more powerful source language that includes nondeterministic functions, L_1, can then be supported.

In this chapter we formulate the problem of nondeterminism in terms of time orderings of *events*, which are the receipt of tokens by operators. The outputs of nondeterministic programs, in general, are dependent upon a perceived time-total ordering of events. Thus outputs of redundant programs can be congruent only if the time-total orderings generated by each independent processing site are consistent. For a system to correctly support nondeterministic functions it must be able to generate a consistent total ordering of selected events at all redundant sites within a bounded amount of time.

The amount of time required to perform this ordering is directly related to the relative *skew* between similar events at redundant operators. In general, skews among similar events may

grow excessively large or even unbounded. Thus mechanisms must be provided to keep these skews within predetermined bounds. If the skew among similar events is guaranteed never to exceed such a bound, the events are said to be *synchronized*.

We begin by formalizing the concepts of events and their time orderings. The source language is extended to include nondeterminism. We then develop congruence schemata which not only deliver congruent outputs but do so in a bounded amount of time, even if the source has diverged. These schemata support the generation of correct time-out tests. We then show, using these schemata, how to support general nondeterministic functions where no prior orderings of inputs are known for the fault-free case.

We conclude by developing synchronization techniques to interactively minimize the skew between similar events in redundant processing sites, and thus improve the efficiency of time-out tests while also supporting the needs for real-time systems. A self-clocking technique is presented that can remove the skew from congruent edges in a completely decentralized manner, without the need of a central system of synchronized clocks.

4.1 Events

An *event* is simply the receipt of a token by an operator. By "receipt" we mean detection of the token at the input. The absolute time of the event is not important and, in a distributed system, is a meaningless measure. Instead, our models for nondeterminism and theories of synchronization rest on partial and total time-orderings of events [19].

Definition 4-1: For any input **a** to operator **f**, the *event* $e(a_j)$ occurs when the a_j token is received by **f**.

The important relationship between events and correct system operation is in the ordering of events. With respect to a single operator, which we consider a point, a time-partial ordering of events is unambiguously obtained.

Definition 4-2: For any two inputs **a** and **b** to an operator **f**, the *time-partial ordering* $<_f$ is given by

$a_j <_f b_j$ if $e(a_j)$ occurs before $e(b_j)$

$a_j \not<_f a_j$

We write **a** $<_f$ **b** if $a_j <_f b_j$ is true for all j.

We actually need a slightly stronger formulation of event ordering to meaningfully resolve system synchronization issues. Data dependencies and timing analysis will help extend a time ordering with respect to a single operator to time orderings with respect to a group of operators.

Clearly, there will be events which occur with respect to different operators and cannot be related by such dependencies or *a priori* timing analysis. These events are called *simultaneous*. It is important that the redundant program instances be able to generate a *consistent* ordering of simultaneous events in order to ensure congruent outputs.

Definition 4-3: A *time-total ordering* of events $e(a_1)$, $e(a_2)$,..., $e(a_n)$ is $a_1 < a_2 < \ldots < a_n$ if whenever $e(a_j)$ occurs before $e(a_j)$ then $a_j < a_k$

There are obvious ambiguities in the ordering of simultaneous events. The generation of orders within this degree of ambiguity is usually done through some sort of priority scheme but this does not concern us here. The important attribute of our system is the ability to generate a consistent total ordering of events at redundant processing sites within a bounded amount of time.

4.2 Nondeterministic Source Language, L_1

By *nondeterminism* we mean that the histories of the outputs of an operator is not just a function of the histories of the inputs, but also dependent upon the *relative order of arrival of the individual tokens at the inputs.* The extension of L_0 to include nondeterminism is simple. However, many of the nice properties of recursive interconnection of such operators are destroyed. For instance, we can no longer obtain congruent outputs from redundant instances of general L_1 programs by simply providing congruent inputs. A stronger time ordering is required at the inputs to *each* nondeterminsitic operator, not just the inputs to the program. Maintaining such conditions is "expensive" in terms of information exchanges between redundant graphs. Thus, nondeterminsitic operators should be sparingly employed.

Definition 4-4: An *n-input, p-output function*, f, is a mapping from time-ordered histories of the input edges to histories of output edges.

f_1: $D^{\omega}{}_1 \times D^{\omega}{}_2 \times ... \times D^{\omega}{}_n \times O \rightarrow D^{\omega}$

f_2: $D^{\omega}{}_1 \times D^{\omega}{}_2 \times ... \times D^{\omega}{}_n \times O \rightarrow D^{\omega}$

•
•
•

f_p: $D^{\omega}{}_1 \times D^{\omega}{}_2 \times ... \times D^{\omega}{}_n \times O \rightarrow D^{\omega}$

Where O is the set of all time-partial orderings of the receipt of tokens at the input of f.

There are two nondeterministic operators of particular interest. Figure 4-1 shows the *nondeterministic merge* places tokens at the output in the same order in which they were received at the inputs without transformation. A *consuming merge* is a nondeterministic filter which places the first arriving token at the output while consuming the token of the same sequence index on the other arc.[6]

Figure 4-1: Two Nondeterministic Operators

It will often be convenient to decompose L_1 programs into L_0 fragments connected by simple edges and nondeterminsitic merges; see Figure 4-2. We will then provide algorithms for ensuring congruent outputs of redundant L_1 programs by ensuring congruent inputs to all L_0 fragments and the same time-partial ordering to all redundant merges.

[6] A consuming merge can be implemented using a nondeterministic merge and a counter. We include it as a separate primitive for clarity.

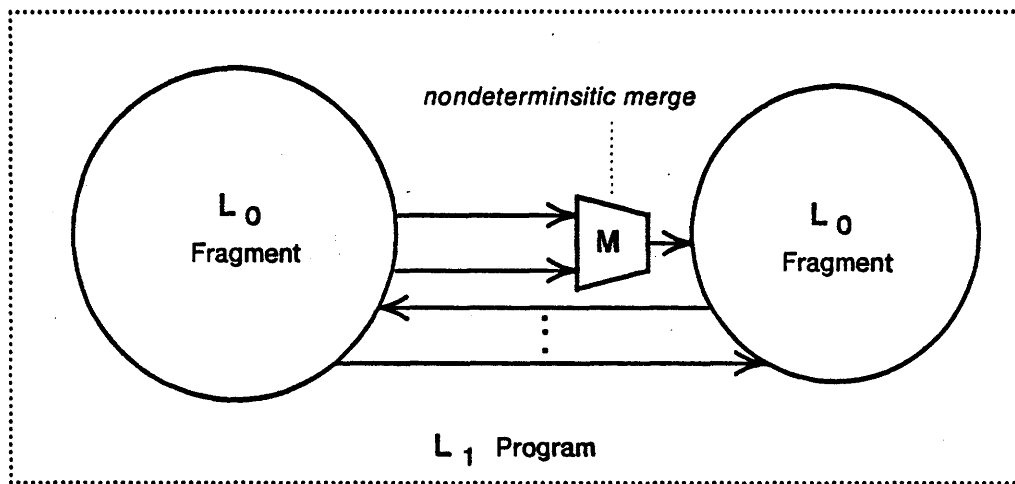


Figure 4-2: Decomposition of Programs into L_0 Fragments

4.3 Nondeterminism in R_1

R_1 is extended in the same way as L_1. That is, we assume L_1 to be a subset of R_1 in the same way as L_0 is a subset of R_0. In addition to the fault sets and interconnect links we further require, in certain cases, timing information for particular operators and interconnect links. These will be absolute bounds on the computation and transmission delays of the operators and links.

> **Definition 4-5:** For any operator or interconnect link the *minimum and maximum latency* δt_{min} and δt_{max} will be the minimum and maximum input to output delays, respectively, during fault-free operation.

This information need not be provided for all operators, but will be essential for the correct generation of time-out tests of divergent simplex sources. In some cases we will only use the

skew or relative difference between redundant instances of operators. In these cases only the weaker conditions of worst case skews need be given.

4.4 A Priori Orderings

In establishing a consistent total ordering of events we have distinguished between two classes: those events that can be ordered *a priori* in the fault-free case, and those events that are *simultaneous*. Events for which we have prior orderings are special in that a departure from the ordering during actual execution can only occur through a fault, somewhere, in the system. The prior ordering is derived either from data dependencies or by timing analysis. Figure 4-3 contrasts the two derivations.

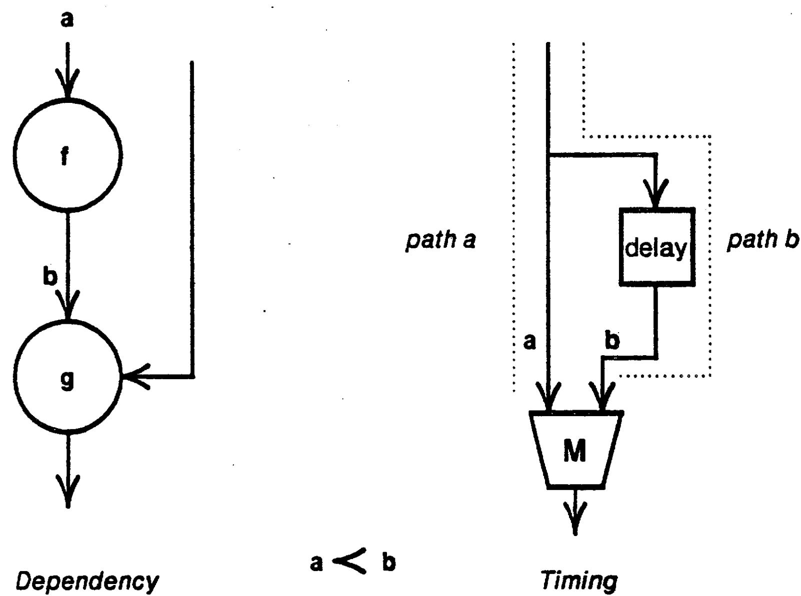


Figure 4-3: Prior Orderings from Dependencies and Timing Paths

The data dependency ordering is a static property of the topology and operators. In this case we can write **a** < **b** because b_i is dependent upon a_i for all i. In the timing relation we

can write a < b only when

$$\Sigma\delta t_{max,path\ a} < \Sigma\delta t_{min,path\ b}. \qquad (4\text{-}1)$$

Violation of a data dependency is a departure from the static functionality of the graph, and is completely captured in the fault model of the last chapter. Event orderings from timing analysis are much more implementation sensitive, but are essential for the correct construction of time-out tests. We shall show that the ability to establish *a priori* bounds on the time to perform a congruent distribution of a simplex source is the primitive property which distinguishes synchronous systems. By this we mean that the congruence schema must produce *some* (congruent outputs by a given time. That is, congruence schemata cannot have divergent outputs even in the presence of divergent inputs.

The creation and support of other constructs, such as clocks and non-deterministic merges, follow naturally from the bounded distribution-time property.

4.4.1 Timing Races

The generation of a time-out test reduces to a fundamental *timing race*, as shown in Figure 4-4. This Figure represents the primitive time-out test for the source S. Imagine a trigger token, represented on dashed arcs,[7] is dropped into the top of the graph. A timing race ensues between the two sides. If we know that $\delta t_{max,S}$ is strictly less than $\delta t_{min,delay}$ than we should always get (in the fault-free cases's at the output. If S should fail such that its output diverges, then we will get τ at the output. Clearly, in the presence of a failure of S, *we are guaranteed a token at the output within* $\delta t_{max,delay} + \delta t_{max,C}$ *seconds.* The design problems are to

1. Determine $\delta t_{min,delay}$ and employ a suitable number of redundant tests such that a fault-free source is never timed out;
2. Consistently distribute the results of the test to all fault-free sites using the test;
3. Generate the appropriate trigger tokens in a fault-tolerant manner and within acceptable skews to meet 1.

The generation of independent trigger tokens with acceptable skews is a problem of

---

[7]There could have been two different sources, *e.g.* clocks, for these triggers. This requires the skews between the arrival of the triggers to be known, a problem dealt with in a subsequent section.

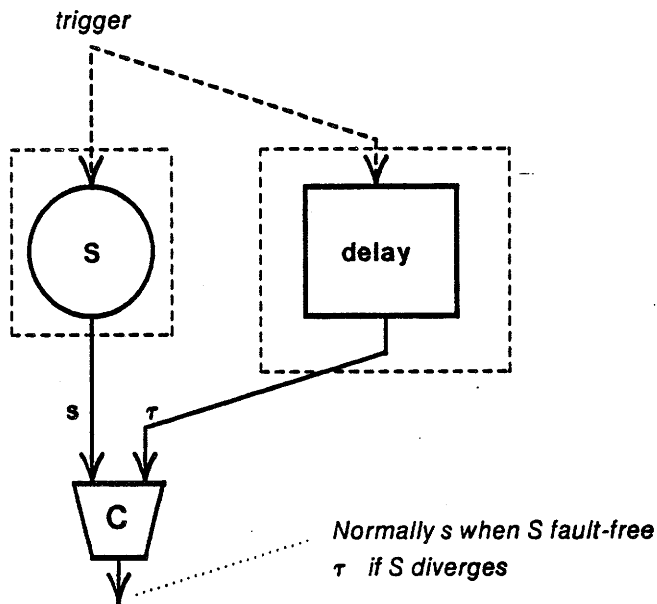


Figure 4-4: Time-out Test as a Timing Race

synchronization. The determination of the value of the delay is readily obtained if neither path has cycles and the worst case skews between the triggers are known. The following section solves the problem of generating consistent time-out tests among redundant processing sites.

4.4.2 Bounded-time Congruence Schemata

This subsection gives construction rules for congruence schemata which not only guarantee congruent outputs in the presence of a specified number of faults, but also guarantee outputs *within a bounded amount of time*. That is, these schemata are capable of the correct generation of time-out tests for simplex sources.

Figure 4-5 shows an initial attempt. Here the source is distributed by a C_f^P schema, so that we are assured congruent versions of the source at all fault-free outputs of the schema, given no more than *f* faults. Time-out tests are then performed independently at each processing site. Thus, each program will have some value for the source within a bounded amount of time.

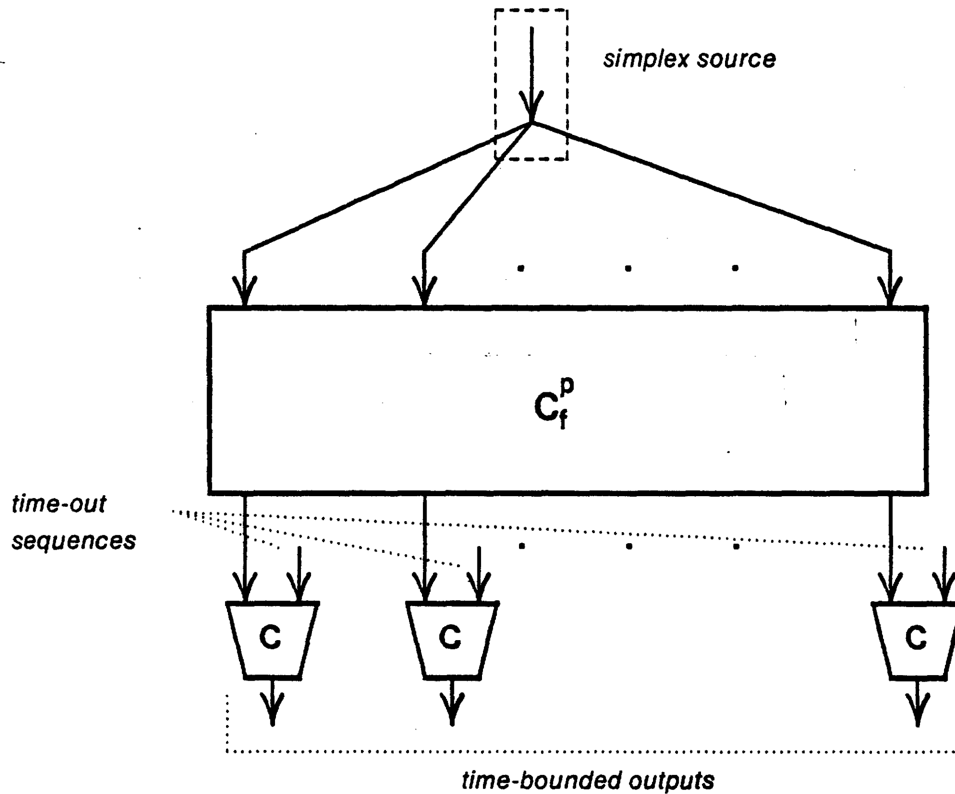


Figure 4-5: Hazardous Redundant Time-out Test

Although all the histories of the redundant edges feeding the consuming merge are congruent, there is no guarantee that the individual tokens arrive in the same order. Suppose that the source fails in such a way to output a token very close to the time-out limit. In this scenario, different tests may give different results. The source may barely arrive in time at some sites while at others the time-out tokens preceded it. *The results of the test must be consistent at all processing sites.*

A refinement of the approach in Figure 4-5 might be to exchange the results of each test and then arbitrate among them. The resolution of the different tests must be done in such a way that a good source is never timed-out. Therefore, we can declare a source timed-out only when a majority of the tests are timed-out. In the ambiguous case where the outcome of the

test is arbitrary (we assume this can only happen when the source is faulty), it can be argued that, the tests being a binary decision, we must have a majority of one result or another. This works only when the source is faulty *or* a minority of the tests are faulty. If the source is faulty and a single test diverges, then all voters may diverge, and we are back where we started: time-out tests must be made on the time-out tests!

The proper place to perform the time-out tests is in the congruence schemata itself. To do so, we introduce a modification to the restoring buffer as shown in Figure 4-6.

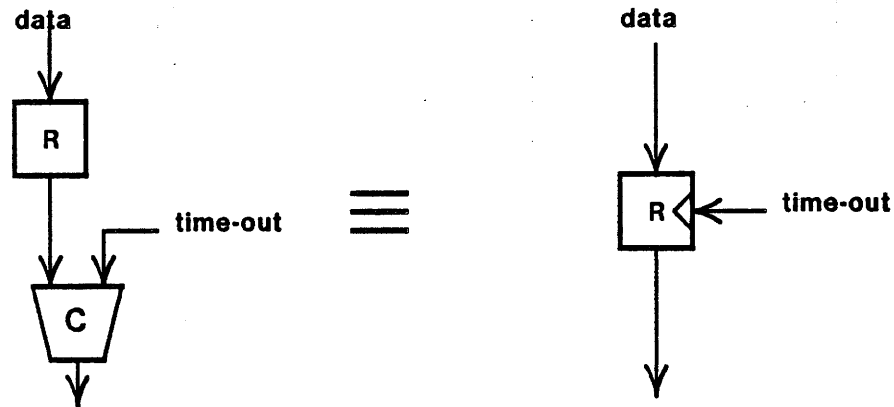


Figure 4-6: Restoring Buffer with Time-out

The buffer restores the input data and merges it through a consuming merge with time-out tokens on the trigger input. We will assume that, under fault-free conditions, the data tokens always arrive before the time-out tokens, **data < time-out.** Thus the output is normally just the stream of data tokens. This new restoring buffer not only restores the signal protocols but, in some sense, restores the timing relationships between individual tokens. We note that the time-out concept is easily extended to a *time-window* that imposes a restriction on the *minimum* time between source tokens.

Figure 4-7 shows the use of these new buffers to generate a canonical τC_1^0 schema. It is easy to see that fragments placed at the inputs still yield arbitrary, but well-formed and

congruent, sequences of tokens at the outputs. Importantly, we are guaranteed outputs within a bounded amount of time under the exact same conditions that C_f^p schemata (see chapter 3) produce congruent outputs. Figure 4-8 shows the standard representation of a τC_f^p schema.

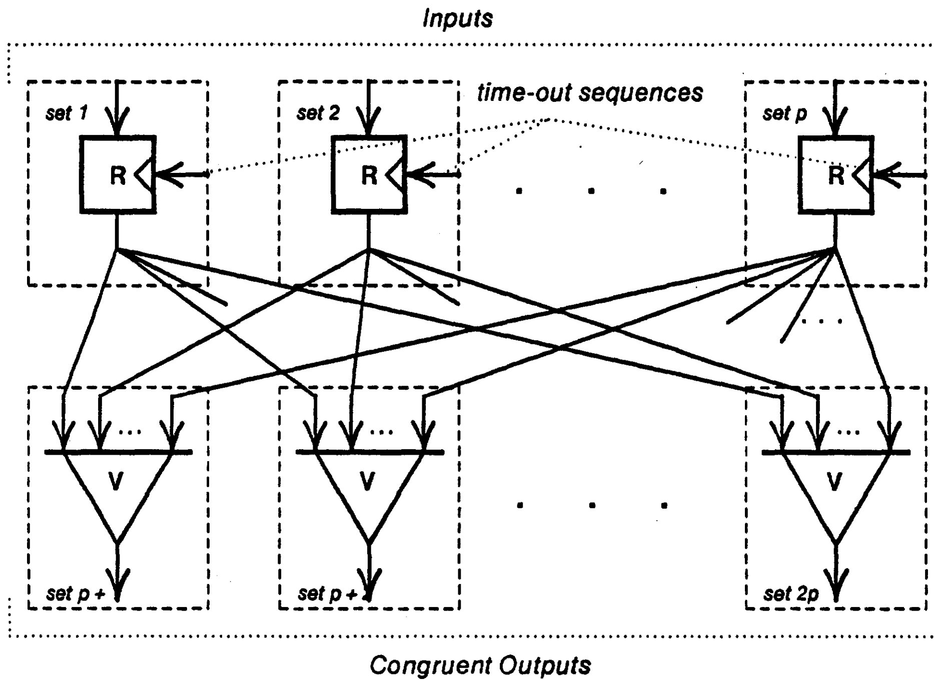


Figure 4-7: A Canonical Bounded-Time τC_f^p Congruence Schema

The construction bounded-time congruence schemata that work correctly in the presence of f faults follows the same recursions employed in section 3.5.1. The proof of correctness follows by induction on the following τC_f^p properties (compare to properties 3-8 and 3-9).

Property 4-1: A τC_f^p schema will always produce congruent histories at all fault-free outputs in a bounded amount of time, whenever there are no more than (p div 2) - 1 failed restoring buffers or associated time-out sequences, and the inputs to the (p div 2) + 1 remaining fault-free buffers form a congruent set.

Property 4-2: A τC_f^p schema will always produce congruent histories at fault-free outputs in a bounded amount of time whenever all restoring buffers and associated time-out sequences are fault-free. This is true for *any* set of inputs including arbitrary streams of token fragments or any combination of divergent inputs.

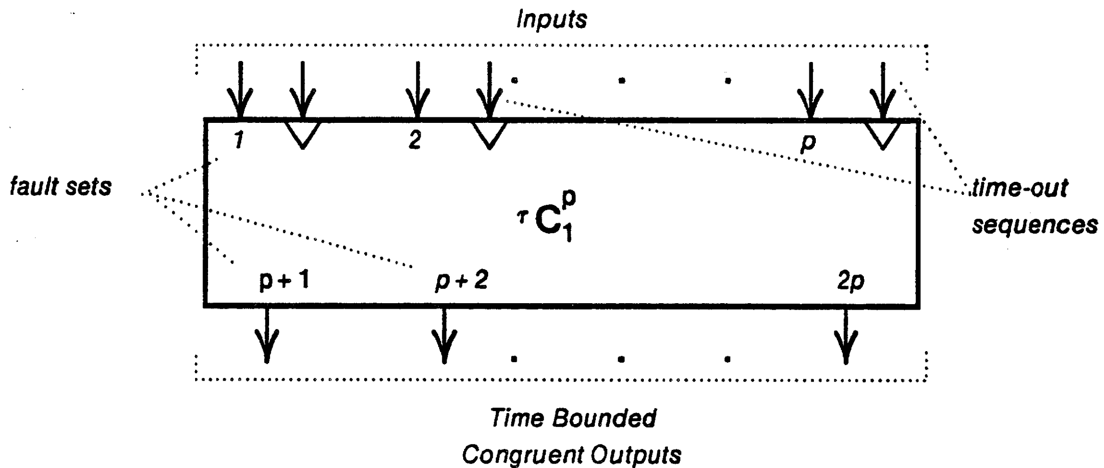


Figure 4-8: Standard Representation of a Canonical $_\tau C_1^p$ Schema

It has been tacitly assumed that we are able to correctly generate the time-out sequences at all fault-free restoring buffers, a nontrivial assumption. We address this problem in the following section on synchronization. Before presenting the synchronization problem, however, we first introduce the prerequisite machinery for the consistent ordering of simultaneous events.

4.5 Simultaneous Events

Suppose we are given a nondeterministic merge in an L_1 program with no prior information on the ordering of inputs. We say that the associated events are *simultaneous*. To maintain congruent outputs of the merge we must require

(M1) All similar inputs to fault-free merges must form a congruent set;

(M2) All similar fault-free merges have the same time-partial orderings on the inputs.

It is actually easier to first consider the problem of constructing a congruent consuming

merge. Remember that a consuming merge forwards the first-arriving token on one edge and subsequently consumes the corresponding token on the other edge if and when it arrives.

Figure 4-9 shows our approach to maintaining congruent outputs of redundant consuming merges.

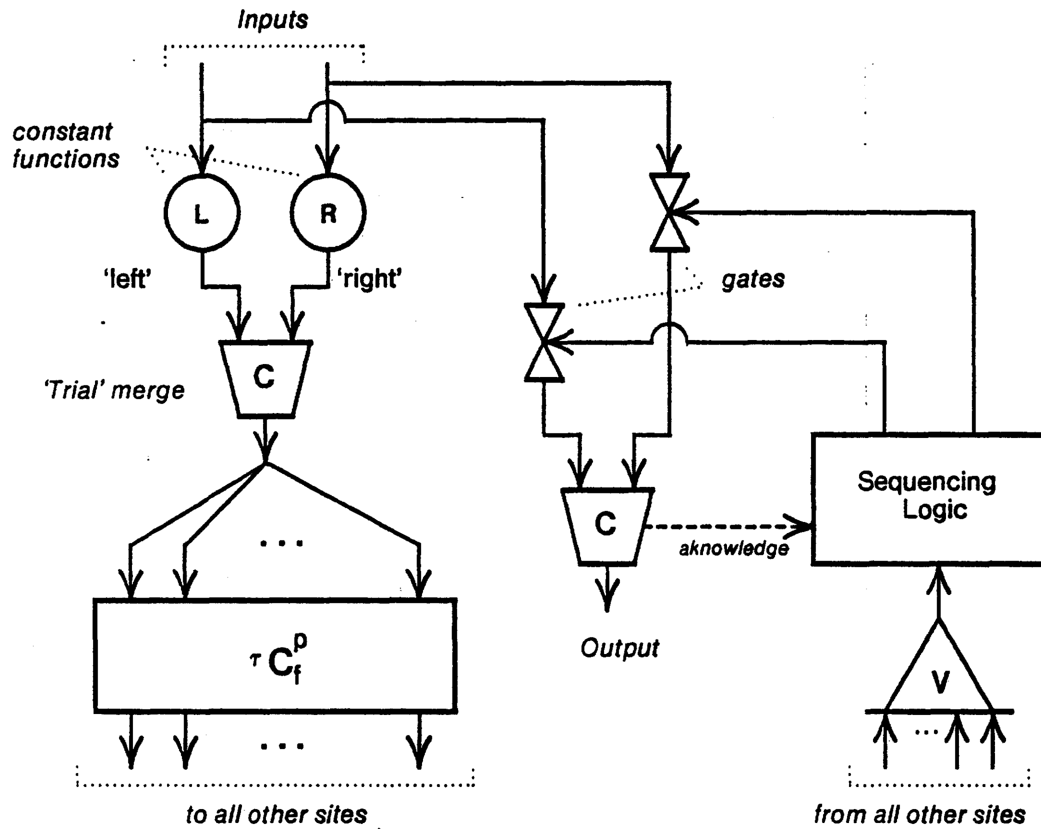


Figure 4-9: Correct Implementation of Redundant Merge

The two incoming edges are routed to a trial merge and the result of this trial merge is exchanged through a τC_f^p schema to all other redundant merges. Similar trial merges are performed at each redundant site and their results are exchanged. All trial outputs are voted, and the input in the majority is selected for output. Thus, each site has consistent information of the results of each trial merge. In the event of a lack of a majority, *i.e.*, the events are indistinguishably close, the left or right input is arbitrarily selected. We guaranteed that all voters have exactly the same input arguments within a bounded amount of time since the

inputs were distributed by $_{\tau}C_f^p$ congruence schemata. Also, when an input is selected we are guaranteed that all fault-free merges actually have a token on the corresponding arc.

One can view each trial merge as an independent external input or "sensor". The output of the sensor is either 'left' or 'right'. If there is no ambiguity, (*e.g.*, there is only a 'left' input), then all fault-free tests will agree, and we will select the correct edge for input. If there is ambiguity, (*i.e.*, there are tokens on both inputs) then we randomly[8] pick a side and we are still assured of a token at the selected input.

General nondeterministic R_1 functions can be implemented in a similar fashion. Thus an L_1 program can be decomposed into L_0 fragments connected by nondeterministic functions as shown previously in Figure 4-2. We obtain congruent outputs of the corresponding redundant R_0 fragments by providing congruent inputs. Congruent inputs, in turn, are guaranteed by the replacement of the nondeterministic functions with their congruent counterparts, and by the distribution of external inputs with $_{\tau}C_f^p$ schemata.

We note that our approach to resolving nondeterminsitic merges is similar to the *synchronizing merge* developed by Leung [9]. In our solution, however, only $2f + 1$ merges are employed, whereas Leung requires $3f + 1$.

4.6 Synchronization

Up to now we have assumed that the time-out sequences have been correctly generated. The efficiency of an R_1 system is intimately related to how close (in time) similar events among redundant processing sites occur. The greater the potential skew, the greater the uncertainty of the relationship between similar events, and thus the larger the delay required for correct time-out tests. We need some way of keeping these events aligned. If similar events are guaranteed to occur within a predetermined amount of time, then these events are *synchronized*. The closer the alignment the greater the *degree of synchronization* of the system.

[8] Some scheme other than random may be employed, such as priority encoding, to resolve the ambiguous case. This is application dependent, but any algorithm which consistently selects one or the other inputs is valid.

4.6.1 Event Skews

One must take care to define the time relation of events independent of some global notion of time. Otherwise, a view of synchronization is developed which is *stronger* than we need and perhaps unrealizable in a distributed system. The following definition of skew yields valid results in Newtonian frames of reference.

> **Definition 4-6:** The skew, δ, among a congruent set of similar events is the maximum time between the first occurring event and the last occurring event that would be measured by any fault-free observer.

We will sometimes speak of *skews between a set of edges or tokens*. Since events are defined only by the receipt of tokens by operators, this usage is imprecise. In all cases, we take this to mean the events which would occur if we connected an operator to the specified edges.

> **Definition 4-7:** A set of similar events are *synchronized* just in case there exists a finite prior bound, δ_{ff}, for the skew of those events associated with fault-free sets

4.6.2 Self-Synchronizing Schemata

In this section we expose properties of C_f^p schemata which can aid in the synchronization of selected redundant edges. Significant features of this technique are

1. The resulting skew after a self-synchronizing exchange is independent of the skew prior to the exchange.

2. The exchanges are supported by C_f^p rather than τC_f^p schemata.

3. Any selected set of congruent edges may be synchronized in a totally decentralized fashion without the need of physical clocks.

The technique is similar to a more specific *mutual feedback* clocking algorithm first developed by Davies and Wakerly [6]. Figure 4-10 shows the implementation of our technique to align similar edges in a general fail-op *f* graph. Edges to be synchronized are exchanged and voted and then delayed according to the maximum skew expected between redundant sites. The resulting edges are realigned to the *maximum skew in the congruence schema rather than the skew in the incoming edges.*

The correctness of this implementation exploits the following interesting timing properties of C_f^p congruence schemata. Note that these are not bounded-time schemata.

> **Definition 4-8:** The *induced skew*, σ, of a fault-free C_f^p schema is the maximum

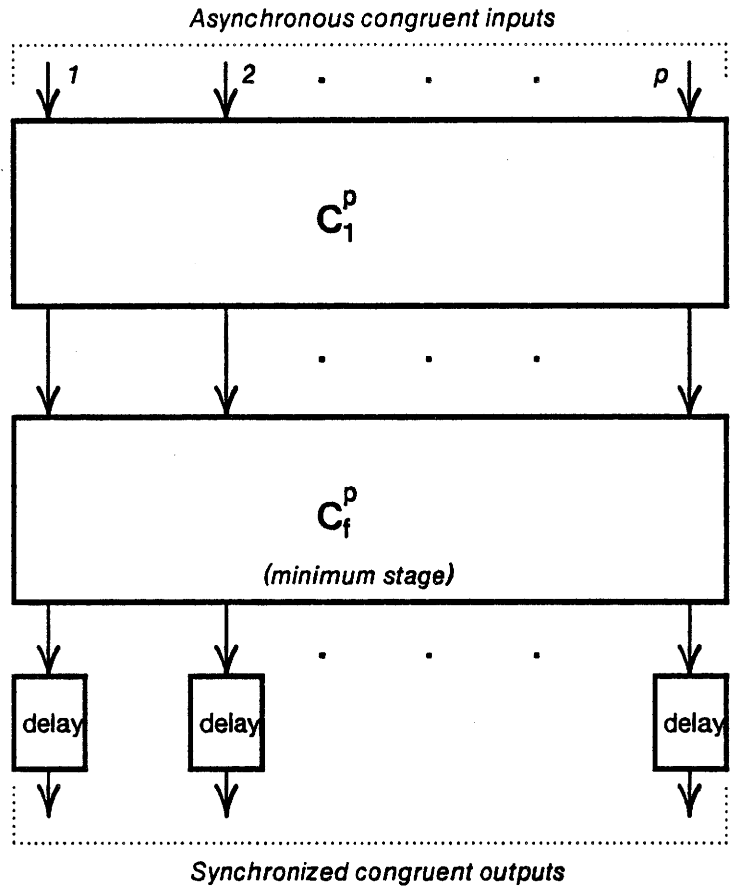


Figure 4-10: Self-Synchronization of Redundant Edges

time-skew among the output tokens when the input tokens have zero input skew.

Property 4-3: A fault-free C_f^p schema with induced skew σ will, given a majority of the inputs form a congruent set, produce output tokens of skew σ, *independent of the input skew.*

Clearly, if all restoring buffers are fault-free then all voters obtain the same tokens within skews that are determined solely by the C_f^p schema itself. Thus, all voters fire at roughly the same time and any additional skew is induced by the differences between the individual voters. Therefore, the output skew is simply the induced skew of schema, independent of the input skew. If some of the restoring buffers are faulty then we can bound the worst-case skew

at the outputs of all unfailed voters as follows.

Property 4-4: A C_f^{ρ} congruence schema with induced skew, σ, given any majority of fault-free restoring buffers driven by congruent inputs with skew δ_{ff}, will produce congruent outputs at all unfailed voters with skew no greater than $\sigma + \delta_{ff}$.

In other words, a C_f^{ρ} schema with a minority of faulty restoring buffers cannot increase the original skew of the good input data by any more than the schema's induced skew.

The proof of this property is straight-forward. All voters will receive the data originating from fault-free restoring buffers with approximately the same skew. The voters cannot fire, however, until a majority of inputs are present. Thus, they cannot fire until at least one token from a fault-free buffer is received, and may have to wait until the tokens from all fault-free buffers are received. Therefore, the worst possible scenario of colluding and lying faulty restoring buffers can induce selected voters to fire on the receipt of the first fault-free restoring buffer token, while others are forced to wait until the last token is received. Because the maximum skew among inputs to *any* majority of fault-free restoring buffers is bounded by δ_{ff}, by assumption, the maximum skew among fault-free outputs is bounded by $\sigma + \delta_{ff}$.

Theorem 4-5: Suppose the self-synchronizing schema shown in Figure 4-10 is implemented with a minimum stage C_f^{ρ} schema. If each constituent C_f^{ρ} schema has an induced skew of σ, then the outputs will be synchronized within skew of $(f+1)\sigma$, given no more than f internal faults.

Proof: There are a total of $f+1$ independent exchanges, each with induced skew σ. Thus, in the presence of no more than f faults we are guaranteed at least one fault-free exchange. By property 4-3 the outputs of this exchange will have skew σ, independent of the input skew. By property 4-4 the subsequent faulty exchanges can only increase this skew by σ at each stage. Since there may be as many as f faulty subsequent stages, the total accumulated skew must be less than $\sigma + f\sigma = (f+1)\sigma$.
□.

The delays introduced after the exchange are selected to allow all fault-free redundant graphs to have produced an input to the schema before the computation proceeds. This prevents any timing races that could be generated by a cyclic graph from "leaving behind" the slower computation sites, since the schema only needs a majority of congruent inputs to fire. The delays are simply equal to the value of the worst case input skew δ_{ff} that can be expected among all fault-free inputs.

The determination of suitable delays is easy for acyclic subgraphs. In the case of a cyclic subgraph, the cycle should be broken and then reformed so that the looping variables are

realigned through a self-synchronizing exchange. We assert that it is possible to generate automatic procedures for the insertion of self-synchronizing schemata, given just the simplex source program and the difference $\delta t_{max} - \delta t_{min}$ for each operator.

4.6.3 Latency

The amount of time it takes to perform a correct time-out test or to synchronize a set of edges is directly proportional to the sum of the worst case skew, δ_{ff}, and the total delay of a the congruence schema.

This total delay, the *latency*, relates to the efficiency of the system, and defines, to a certain extent, the finest grain computation cycle which can be supported by the system. For high performance control systems such as aircraft flight control, the latency should be under one millisecond to prevent an excessive domination of redundancy overhead. Similarly, latencies lower than approximately 100 microseconds are less than what is required. Clearly, the degree of system synchronization and the implementation technology of the congruence schemata (hardware versus software) will have the largest impact on the system latency.

4.7 Remarks

The ability to consistently order events and the ability to bound the maximum skew among similar events are the key attributes of a system which can successfully support nondeterminism. It has been shown that the ability to generate a congruent set of edges with congruence schemata is an essential feature of correct systems, and can be the dominating factors in the system's efficiency.

This theory allows two fundamental questions to be answered about a fault-tolerant system design. Is it correct? That is, are the minimum number of fault sets present, are congruence schemata implemented, is the synchronization technique hazard free? If the system passes these tests, is it efficient? What is the maximum skew and the total latency? Many engineering decisions can be simplified if the fault-tolerant designer keeps these questions in mind.

Chapter Five

Applications

A primary application of the theory developed in this research is the provably correct implementation of highly reliable real-time control systems. In these systems, a loss of computational capability, even for a fraction of a second, can directly lead to substantial financial and human costs. This chapter presents some specializations of our theory to the needs of such systems.

There are two distinguishing features of real-time control systems that have a direct effect on the formulation of our system models: the extensive use of stochastic input devices (*e.g.*, analog sensors), and the requirement that certain computations must occur at predetermined intervals of time. This chapter develops *sensor models* for those classes of input devices which are frequently employed, and investigates the construction of *physical clocks* which time-regulate the flow of computation. We conclude with a critical discussion of current fault-tolerant control systems which both meet and, fall short of, the theoretical requirements for correct system operation.

The author has directly applied this theory to his design and analysis of the communication and synchronization subsystems of two recent fault-tolerant control systems targeted for flight control of high performance aircraft. The C.S. Draper Laboratory FTP is quadraplex fail-op^2 (sequential) that employs four tightly synchronized M68000-based microprocessors to give the user a transparent simplex programming environment [10]. The Honeywell M^{2}FCS, in contrast, is a distributed, data synchronized multiprocessor system which exploits self-checking processor pairs as the fundamental fault-isolation unit [20] [21]. The models developed in this report proved valuable in clarifying subtle engineering design decisions that might otherwise compromise the high levels of reliability desired.

5.1 Sensor Models

This section specializes the theoretical concept of an external input to represent the kinds of input devices most often found in real-time control systems. We show how sensors that are "read" are easily represented by demand driven models. Cross-strapping techniques which provide redundant interconnect links are also modeled. Finally, we discuss the architectural implications of a sensor which is directly connected to processing site, (*e.g.*, through an I/O card), before being distributed through a congruence schema.

5.1.1 Demand Driven Sensors

In our abstraction of external inputs, they seem to spontaneously produce tokens for the consumption of the program. Most inputs in control systems are treated in a more imperative fashion. When a value is required it is "read" from the device. This operation, in the case of analog sensors, is the reading of a conversion latch from an analog-to-digital converter. An acceptable way to model this operation in our source language is through the use of *demand driven sensors*. As shown in Figure 5-1, the external input is modeled in the source language L_1 by a stochastic function (the sensor) which produces data whenever *any* type of token, the demand or trigger, is placed at its input. Note that the sensor is not a pure function in the sense of our languages since it does not determinately map time-partial ordered sequences of inputs to sequences of outputs.[9] Instead, the value of the output is a function of the absolute time (relative to the sensor) at which the trigger token was detected.

Figure 5-2 shows how this model is translated into R_1. The voter resolves redundant demand streams from the p processing sites to a single demand stream. The resulting output is distributed to the redundant processing sites through a τC_f^p schema. The value of delay required is equal to

$$\delta t_{min,delay} = \delta t_{max,V} + \delta t_{max,sensor} + \delta t_{max,links} + \delta_{ff}.$$

Where δ_{ff} is the maximum skew expected among all fault-free demand streams. It often happens that δ_{ff} completely dominates this sum. In tightly synchronized systems such as the FTP, the maximum expected skew is less than one microsecond and the total exchange latency is approximately 5 microseconds. In contrast, the SRI SIFT system has considerably

[9] If it did, it would be a most uninteresting sensor

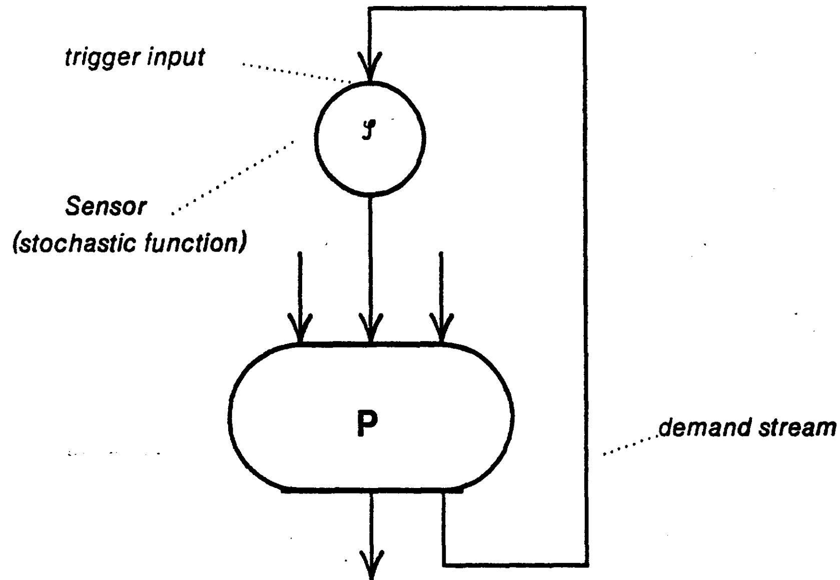


Figure 5-1: Simplex Model for a Demand Driven Sensor

more skew and an (unacceptable) latency of well over a millisecond.

5.1.2 Processor Hosted Sensors

In the above sensor model it is assumed that the sensor fault set contains the necessary hardware for the conversion of the analog sensor output to a well-formed token. Sensors are usually not this sophisticated. Instead, the analog-to-token conversion is usually hosted at one of the redundant processing sites. That is, the analog signals are typically routed to analog-to-digital converters on input/output boards on the backplanes of digital processors. This brings up an interesting architectural issue of how to correctly distribute this class of inputs. The data from the sensor must be correctly distributed for input to all other processing sites. Thus, as shown in Figure 5-2, the data must be routed through a congruence schema. Note that the output edge of the sensor is actually a member of the fault set corresponding to one of the instances of P (the one that had the conversion hardware). Recalling design lemma 3-11, we cannot let this fault set participate in the restoring

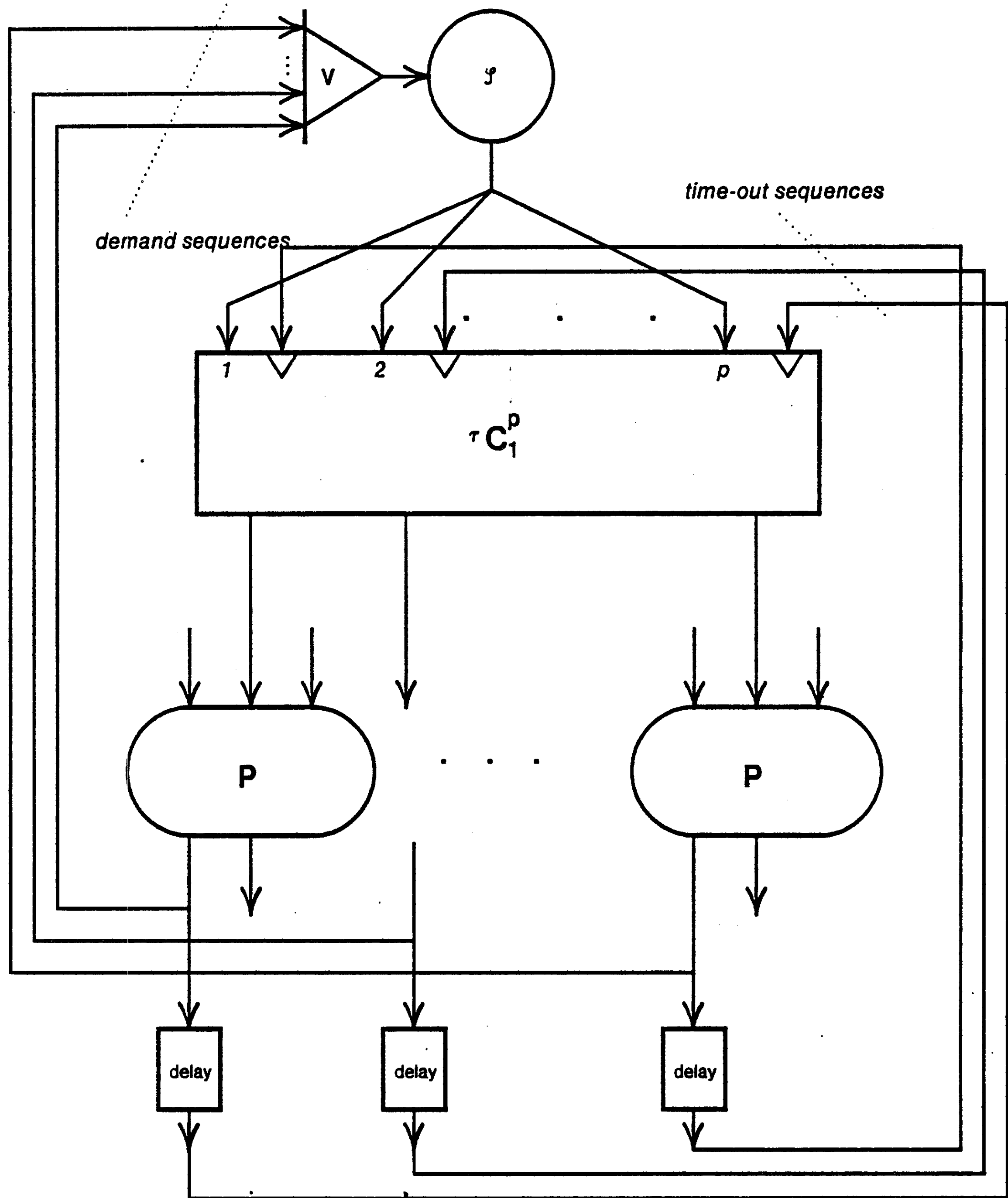


Figure 5-2: R_1 Model for a Demand Driven Sensor

operations in the congruence schema. Up to now, in designing a minimum fault set congruence schema, we allowed all target sets to participate in the exchange. We can still allow all the other processing sites to perform restoring functions except the host of the sensor.

We might be tempted to introduce a new fault set which assumes the remaining restoring function. Typically, all processing sites will host sensors, so this additional set must somehow be multiplexed to support the restoring functions for any host processor which must be excluded. This is certainly possible to do. However, we gain much more regularity by completely divorcing the restoring function from the processing sites. The FTP system does precisely this and the restoring functions, with time-outs, are called *interstages*. Topology and fault set assignment are shown in Figure 5-3. Here, a departure from a $3f + 1$ fault set minimum has yielded a much higher performance and regular system.

5.1.3 Sensor Cross-Strapping

It often happens that the inherent reliability of the sensor is much higher than the processor which is hosting it. If the sensor is feed only through this processor then its failure rate is approximately that of the host processor. To recover the original sensor reliability, the outputs are often routed to several processing sites in a technique called *cross-strapping*. In this case, if a single analog-to-digital conversion or the associated processor fails, the sensor data is still available to the system. To correctly model such a connection we must conceptually view the sensor as several highly correlated sensors sharing the same fault set (see Figure 5-4). The multiple sensors will tend to converge to the same value as the skew among the demand streams decrease. In general, even with zero skew, these results will never be precisely the same due to broad spectrum white noise induced on the analog lines. Thus a blending algorithm must be supported within P to resolve a best estimate of the sensed value from the several read. Of course, each independent value must be distributed to all users through a congruence exchange. We note that even the hosting processor cannot use the value directly; it must always use the exchanged value to ensure consistent information throughout the system. Due to the heavy input/output demands of most real-time control systems, the throughput of the system may become heavily dependent upon the efficiency of congruence exchanges.

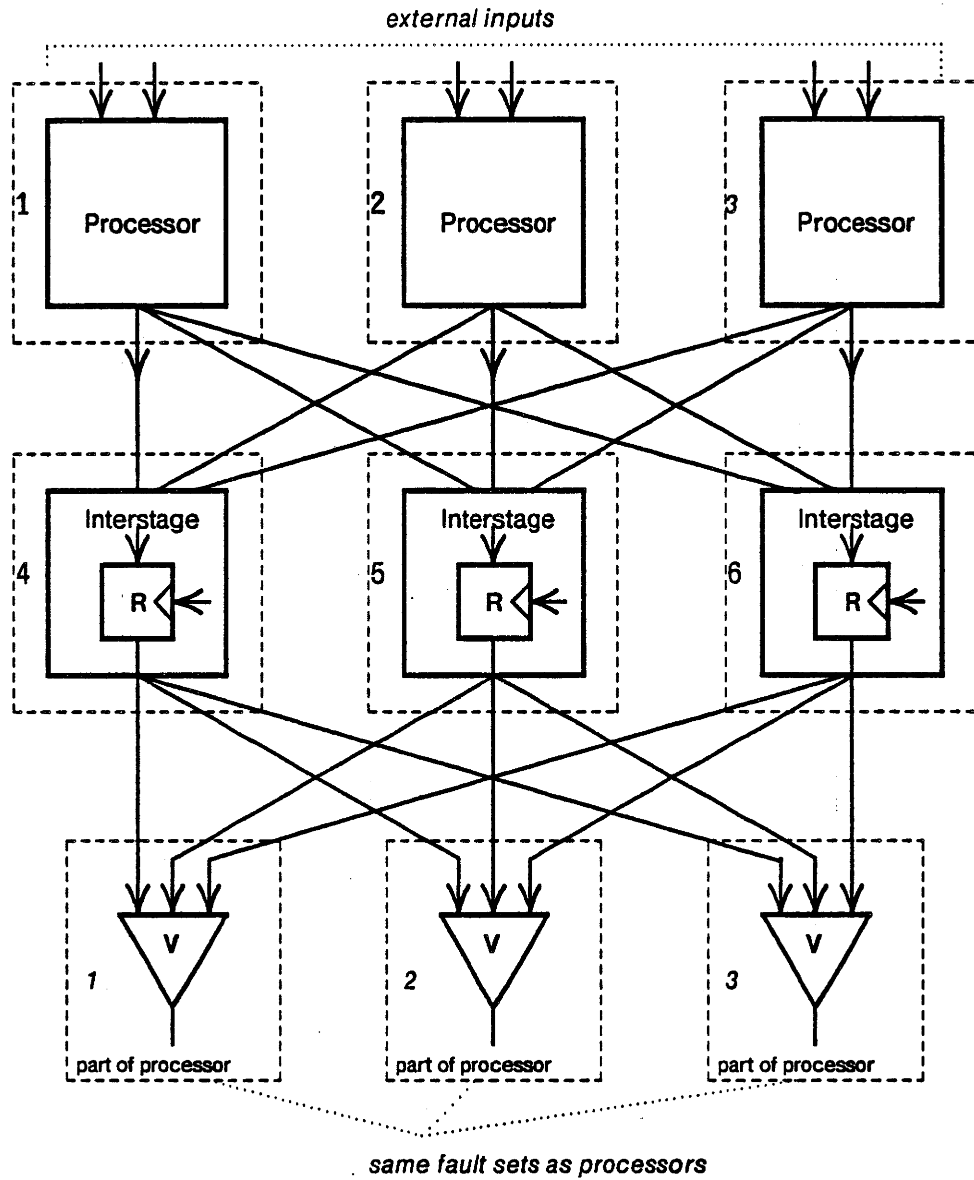


Figure 5-3: Fault Set Assignment in the C.S. Draper Triplex FTP

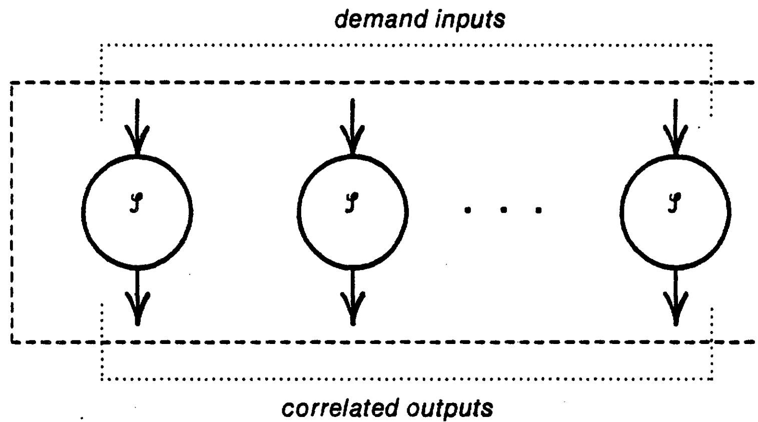


Figure 5-4: Correct Model for a Cross-Strapped Sensor

5.2 Physical Clocks

Physical clocks and corresponding *schedulers* are essential for real-time control systems where certain operations, such as sampling sensors, must occur at specified times and at regular intervals. Figure 5-5 shows a sample L_1 program where clocks have been introduced to regulate the system behavior. A timebase puts out regular ticks and feeds a finite state machine sequencer, the scheduler. The scheduler sends trigger tokens to gates which have been installed on various edges. These new dependencies are the sequencing constraints.

When translating this structure to an R_1 model we must ensure that tokens are at the inputs to all gates before the corresponding triggers arrive. This analysis can be carried out for acyclic subgraphs, but is difficult, if not Turing-unsolvable, to bound the computation time for an arbitrary collection of operators. Fortunately, in real-time systems the programs which require tight sequencing have execution times which in fact are independent of the inputs.

Once these times have been established and the sequencer designed, the redundant

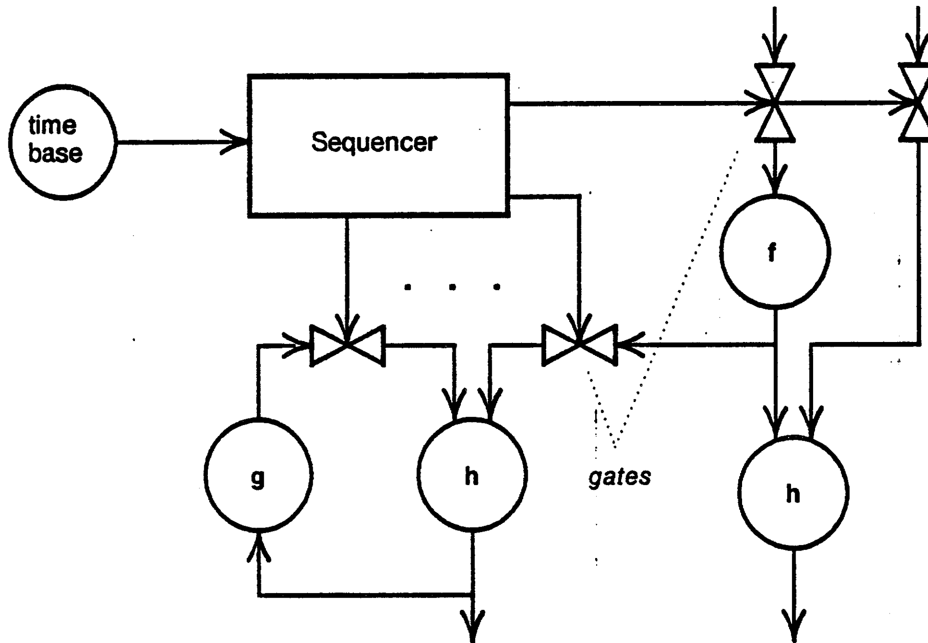


Figure 5-5: Absolute Clocking of a Program

sequencers must be kept aligned to a prescribed timing skew. That is, we must somehow keep the timebases aligned.

This problem is very easy to solve using self-synchronizing schemata. As shown in Figure 5-6, the outputs of the schema is feed through a delay which is approximately equal to the period of the desired clocks. The actual period is found by summing this new delay with the latency of the schema. Many other approaches are possible. Daly, Hopkins and McKenna [22], for example, show how to keep independently running oscillators aligned through a fault-tolerant phase-locked loop. We note that the analysis of phase-locking techniques are greatly simplified if one can assume that all oscillators have the same relative phase information as all other oscillators. This, in turn, is easily assured through the use of the correct congruence exchange mechanisms for the distribution of phase information.

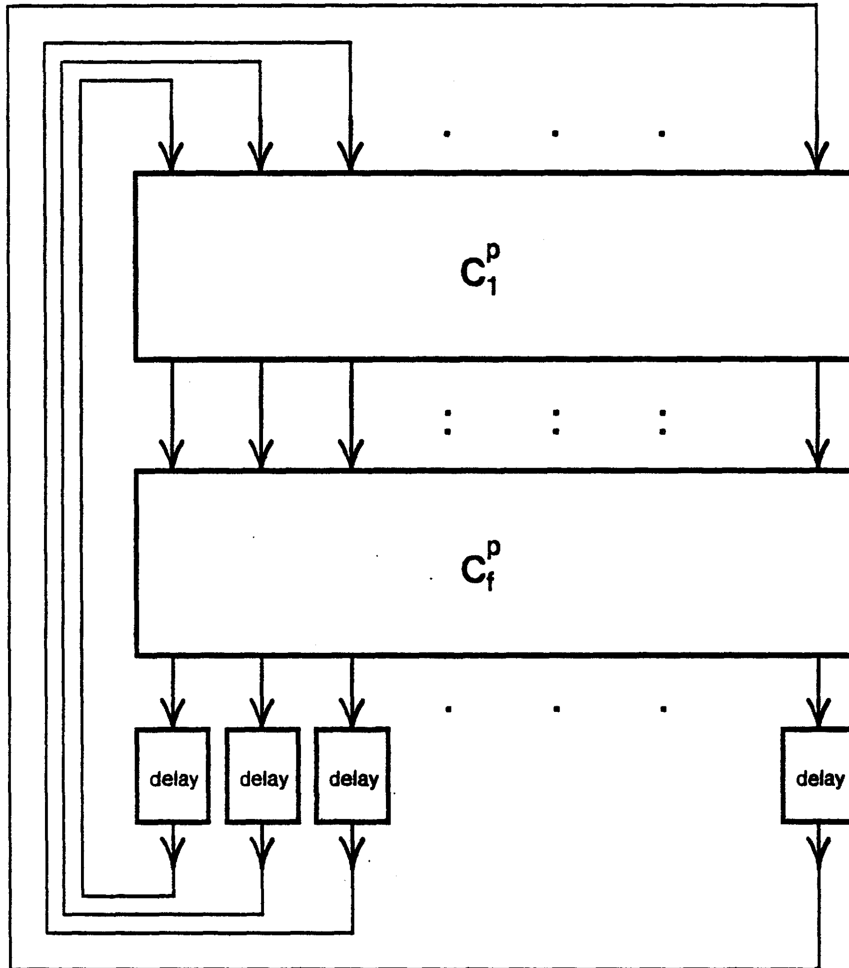


Figure 5-6: Physical Clocks from a Self-Synchronizing Schema

5.3 Critique of Current Systems

Many systems which claim to be "fault-tolerant" are not so in the very strong sense in which we use the word. Most systems do not employ active redundancy or comparison monitoring. Instead, they rely on a spectrum of self-test, encoding, re-try, and reversion

modes, to detect and isolate hardware faults. Overall, an aggressive application of these techniques can expect to cover (correctly isolate and reconfigure against) about 95-98% of the failures that occur. The actual reliability of a given computation is not significantly enhanced, but the total "up-time" of the system can be substantially improved. We term such systems *high availability* rather than *high reliability*. Systems like the Tandem Non-Stop, Stratus, and CMU's C.mmp and C.m* fall into this category, and all contain single-point hardware failure modes.

Systems which employ active congruent redundancy, on the other hand, attempt to improve the short-term reliability of the system rather than its long-term availability, although high availability may be a consequence. We briefly examine four such systems. Two of them, SRI's SIFT and Draper Laboratory's FTMP, meet the conditions of our theory, but suffer from excessive inefficiencies and complexity, respectively. The other two systems, the NASA/IBM Space Shuttle System, and CMU's C.vmp are theoretically deficient, and we point out a few of the single-point failure modes.

5.3.1 SIFT

The Stanford Research Institute's SIFT (for Software Implemented Fault Tolerance) computer system is a theoretically aggressive approach to construct a provably correct fault-tolerant computer system [7]. A fully interconnected (cross-linked) set of 4 or more fairly conventional processors are configured into a fault-tolerant system using software to implement the various aspects of the redundancy management (see Figure 5-7).

This project is largely responsible for recognizing the hazards of *source inconsistency*, (SIFT's term for the problem of distributing a simplex source) and for generating the algorithms which correspond to our minimum set congruence schemata [4]. Although theoretically correct, the system suffers from severe inefficiencies.[10] Examination reveals that the latency of the congruence schemata is excessively high (approximately 3 milliseconds for a useful computation cycle). In large part this is a consequence of the unnecessary software complexity of the processor exchanges, and the fact that all restoring functions are performed by the processors themselves. The system is therefore compelled to

---

[10] Real-time throughput is about an order of magnitude lower then that required for safe flight control.

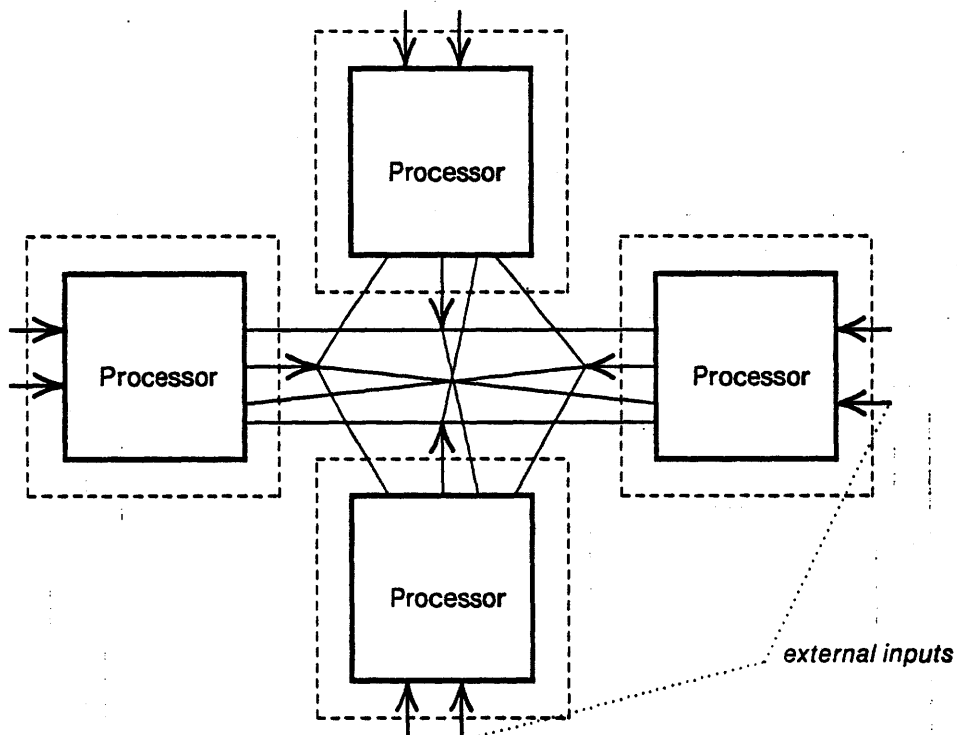


Figure 5-7: Example SIFT Configuration for $f = 1$

support $3f + 1$ independent processors, rather than the $2f + 1$ in our formulation. The project failed to recognize the costs of maintaining congruent edges among redundant processing sites.

We assert that a data flow approach to the generation of the congruence schemata and the synchronization algorithms, along with a little extra hardware (for restoring buffers) would dramatically improve the performance of this system.

5.4 FTMP and FTP

The C.S. Draper Laboratory FTMP [8] was designed as direct competitor to the SIFT system. A very different hardware-intensive approach was taken. The design is based upon triads of independent processor-cache memory modules and common memory modules

which communicate via redundant serial busses.

In sharp contrast to the SIFT system, the FTMP requires highly specialized and custom engineered processors and memories. This is primarily due to the extremely high degree of synchronization of the modules: all processor microcycles are slaved off of a redundant set of phase-locked oscillators. Thus, all redundant data transfers are in synchronism, so a triad looks like it is a single machine. The *Bus Guardians* prevent a single module from containing complete control of the communication system. (The direct links in the SIFT system avoid this problem, but are less flexible.)

Although the throughput of the system is adequate for its intended aircraft control application, the system is very complex; an undesirable attribute for a system which must be provably correct. This complexity was recognized by the designers, and a second generation machine, the FTP, was developed. The FTP consists of a single triad (or quadruple) of processor/memory units, as shown in Figure 5-3. The FTP also continues the philosophy of microcycle synchronization of the redundant processors. The claim is that this gives a transparent (*i.e.,* simplex von Neumann) programming model for the machine. The high degree of synchronization is required to maintain this illusion, owing to the strong, implicit time-ordered control structure of von Neumann processors.

The problem with the high degree of synchronization is not necessarily the difficulty of maintaining a high frequency fault-tolerant clock. Instead, excessive burdens are placed on the design of the independent processing modules. That is, each module must take *exactly* the same number of microcycles when performing any instruction. This requires that all operations in the processor are deterministically derived from the processor clock. This is counter to strong trends in processor design. For instance, refresh mechanisms for dynamic memories usually work on an asynchronous cycle-steal basis. Dynamic memory error correction or internal bus retry strategies cannot be employed due to their inherent nondeterminism.

The most serious problem occurs with interfacing the processors to I/O devices. Here a designer is faced with two relatively unattractive alternatives: (1) extend the synchronization to the I/O device, which requires custom design of each device, or (2) treat the device as an asynchronous peripheral, in which case interrupts (ugh!!) or software polling algorithms must be employed. For example, connecting a disk controller illustrates this problem.

Synchronizing the disk to the processor microclock is hopeless. This would require synchronizing both the platter rotation and the access arm, or by always waiting a worst case amount of time. Data could not be directly DMAed into memory since this would knock the processor out of synchronization. Polling the disk buffer for the next word would be exceptionally inefficient. The only tractable solution is to build a fault-tolerant DMA network that must not only steal the same microcycles from each processor, but also distribute the DMA data in a congruent manner. Similar problems exist in trying to expand the number of processors for parallel computation, and the system quickly assumes the complexity of the FTMP.

FTMP and SIFT both suffer from the problem of supporting parallel computation (of which redundant computation is a special case) on von Neumann machines. An implementation will most likely be either inefficient or require excessive synchronization and associate complexity of design.

5.4.1 The Space Shuttle System

The Space Shuttle computer system [13] consists of five central processors, 24 I/O and interprocessor communication buses, and several remote terminal data multiplexing and display modules. In full configuration, four of the processors, are in tight synchronization, executing approximately the same instruction at approximately the same time. The fifth processor is an independently programmed back-up which may be switched in by the crew in the event of a total failure of the primary four. The system contains theoretical deficiencies and thus possesses potential single-point failure modes. Figure 5-8 [23] is a simplified block diagram of the system. The *Multiplexing DeMultiplexing Units* (MDMs) collect sensor data from around the ship. A device is read by the processors by writing a channel I/O command to the appropriate MDM. The MDM actually only responds to one command even though all processors send out the command. Thus, only one processor actually causes the device to be read, the processors are synchronized enough so that each processor thinks that it issued the command. The data from the MDM is sent directly to each processor, there is no congruence exchange.

Thus, there exists the possibility that a faulty MDM, putting out marginal signals, could issue inconsistent data to each processor. This problem was recognized, but unfortunately too late

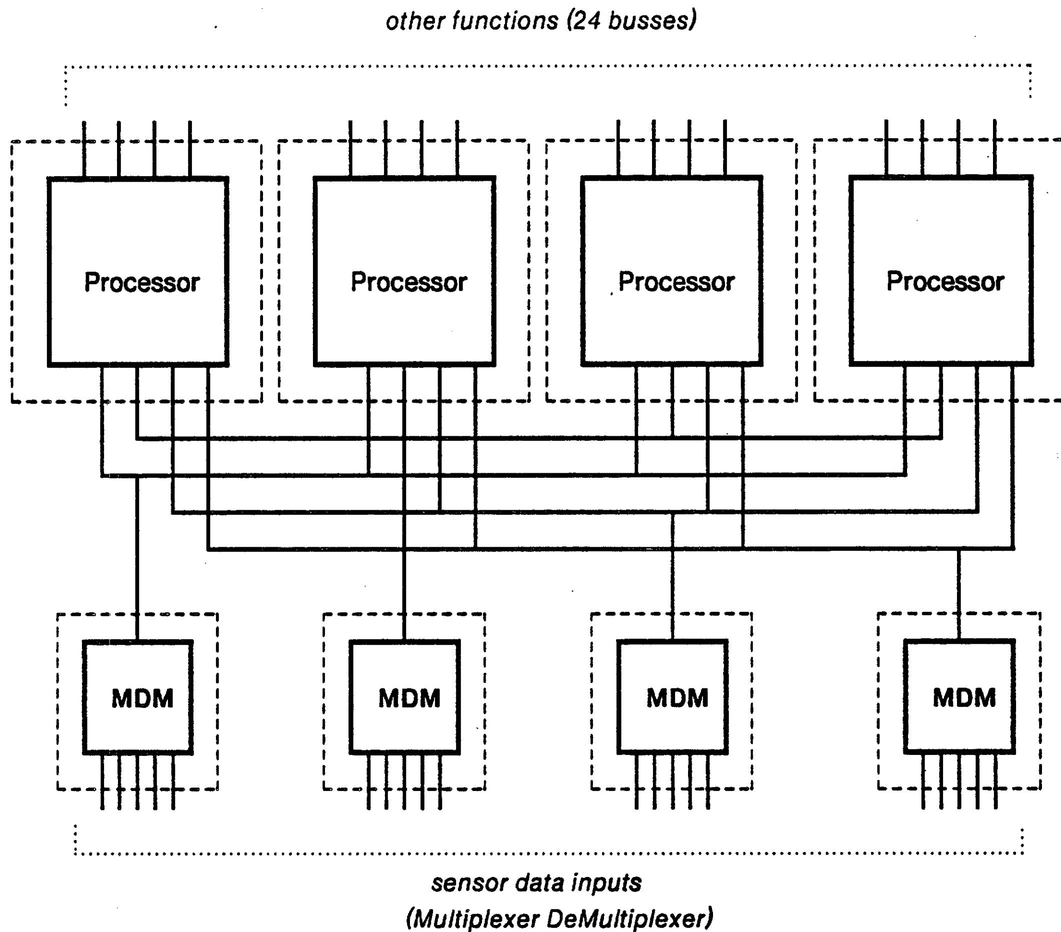


Figure 5-8: Simplified NASA/IBM Space Shuttle Computer System

to provide the necessary extra processor bandwidth to perform the congruence exchange. A "fix" was given whereby the incoming data are checked for parity, etc., then this parity data is exchanged before a block of data is used. If *any* processor claims that the parity is bad then they *all* ignore the data. If the same single processor claims the data is bad for some number of successive transfers, then this processor is considered failed. *This still does not work. (1) The parity data is not passed through a congruence exchange, (2) A bad MDM could force two processors to receive bad data, thus resulting in two processors being thrown out.* We have informally learned that (2) actually occurred during a ground test where a technician neglected to install the proper termination on an MDM bus. In this case, two of the processors

were situated at the correct position on the bus so that the resulting reflections were canceled out, two others were not so fortunate and were dropped by the system. All one can do is hope that this does not occur in flight.

We also note that the Shuttle System faces the same expansion problems as the FTP, and the system is already extremely overloaded.

5.4.2 C.vmp

The C.vmp computer system was developed at Carnegie-Mellon University in connection with C.mmp and Cm* computer systems. The trial configuration consists of of three DEC LSI-11 processors which have a voter inserted within the Q-bus, as shown in Figure 5-9. The voter can work in a "feed-through" mode which allows the processors to work essentially uncoupled, or in a "voting" mode which votes Q-bus data both directions (memory-to-processor and processor-to-memory transfers). The voter offers a single-point failure for the system, which the authors, of course, recognize [14]. However, their solution to the problem is questionable.

> The nonredundant portion of the voter does not represent a reliability "bottleneck",... [T]he voter may be totally triplicated if desired. With voter triplication even the voter can have either a transient or a hard failure and the computer will remain operational.

The authors underestimate the implications of their assertion. The single voter would unambiguously restore a bad signal protocol on a single bus. But in the triplicated case, if we attempt a simplex I/O transfer from a faulty device, there exists the possibility that each voter will see the transfer differently and the resulting data will be incongruent, ultimately leading to a complete crash. Moreover, it may be impossible to construct a congruence schema given the fault set assignments in the system. It is also extremely doubtful that the complex Q-Bus asynchronous protocols could be independently implemented by each of the three voters. Some form of tight synchronization would be required.

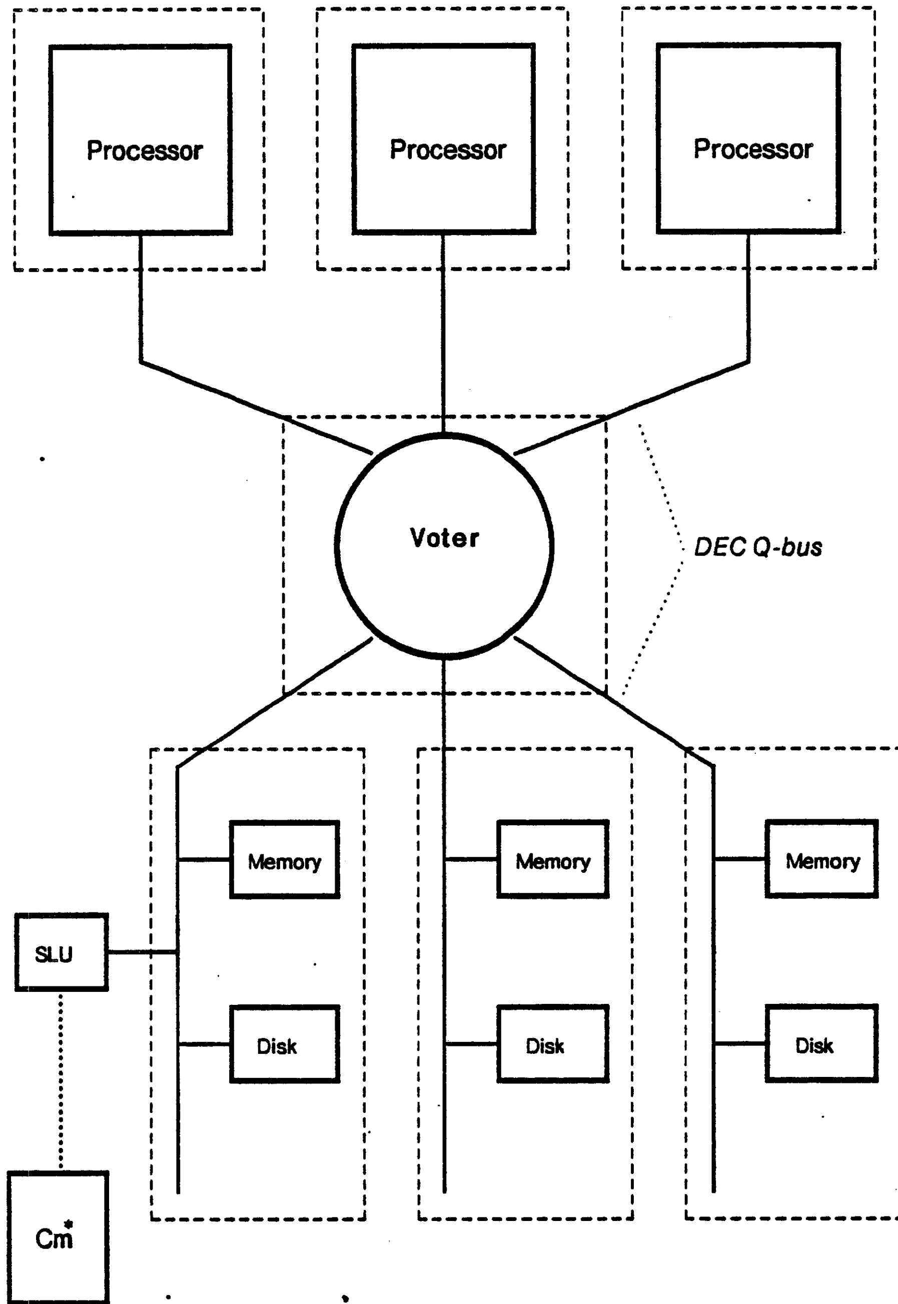


Figure 5-9: Carnegie-Mellon C.vmp Configuration

References

[1] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata Studies,* Vol. Annals of Math. Studies No. 34, Princeton University Press, 1956, pp. 43-98.

[2] A. Avizienis, "Fault Tolerance: The Survival Attribute of Digital Systems," *Proceedings of the IEEE,* Vol. Vol. 66, No. 10, October 1978, pp. 1109-1125.

[3] A.L. Hopkins, "Fault-Tolerant Systems Design: Broad Brush and Fine Print," *IEEE Computer,* March 1980, pp. 39-45.

[4] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM,* Vol. Vol. 27, No. 2, April 1980, pp. 228-234.

[5] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems,* Vol. Vol. 4, No. 3, July 1982, pp. 382-401.

[6] D. Davies and J.F. Wakerly, "Synchronization and Matching in Redundant Systems," *IEEE Transactions on Computers,* Vol. Vol. C-27, No. 6, June 1978, pp. 531-539.

[7] J.H. Wensley, *et al.*, "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft," *Proceedings of the IEEE,* Vol. Vol. 66, No. 10, October 1978, pp. 1240-1254.

[8] A.L. Hopkins,Jr., T.B. Smith,III, and J.H. Lala, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings of the IEEE,* Vol. Vol. 66, No. 10, October 1978, pp. 1221-1239.

[9] C. Leung, "Fault Tolerance in Packet Communication Computer Architectures," Ph.D. Thesis TR-250, MIT Laboratory for Computer Science, September 1980.

[10] T.B. Smith, "Synchronous Fault-Tolerant Flight Control Systems," C.S. Draper Laboratory, Cambridge MA. P-1404,.

[11] J.B. Dennis, "First Version of a Data Flow Procedure Language," *Lecture Notes in Computer Science, Volume 19: Programmin Symposium, Callojue sur la Programmation, B. Robinet, Ed.,,* Springer-Verlag, , pp. 362-376.

[12] Arvind, K.P. Gostelow, W. Plouffe, "An Asynchronous Programming Language and Machine," Technical Report TR114a, Department of Information and Computer Science, University of California - Irvine, Irvine California, 1978.

[13] , "OV-102 Space Shuttle Systems Handbook, Vol. II," Tech. report JSC Document #11174 Rev. A, NASA Johnson Space Center, Houston, Tx., 1979.

[14] D.P. Siewiorek, *et al.*, "A Case Study of C.mmp, Cm*, and C.vmp: Part I," *Proceedings of the IEEE,* Vol. Vol. 66, No. 10, October 1978, pp. 1178-1199.

[15] H. Ihara, F. Fukuoka, Y. Kubo, and S. Yokota, "Fault-Tolerant Computer System with Three Symmetric Computers," *Proceedings of the IEEE*, Vol. Vol. 66, No. 10, October 1978, pp. 1160-1177.

[16] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proceedings of the IFIP Congress 74*, Information Processing 74:, June 1974, pp. 613-622.

[17] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill Publishing Company, New York, 1974.

[18] A.D. Friedman, P.R. Menon, *Fault Detection in Digital Circuits*, Prentice-Hall., Englewood Cliffs, New Jersey, 1971.

[19] L. Lamport, "Time, Clocks, and Ordering of Events in a Distibuted System," *Communications of the ACM*, Vol. Vol. 21 No. 7, July 1978, pp. 558-565.

[20] R.E. Pope, *et al.*, "A Multi Microprocessor Flight Control System for Aircraft - Design Principles," , AIAA Guidance and Control Conference, Boulder, CO, August 1979, pp..

[21] J.A. White, *et al.*, "A Multi Microprocessor Flight Control System - Architectural Tradeoffs," , Second Conference on Computers in Aerospace, Los Angeles, CA, November 1979, pp..

[22] W.M. Daly, A.L. Hopkins, and J.F. McKenna, "A Fault-Tolerant Digital Clocking System," *Dig. 3rd Int Symp. Fault-Tolerant Computing*, Vol. IEEE Publ 73CH0772-4C, June 1973, pp. 17-22.

[23] , "OFT Functional Design Specification," Tech. report NAS 9-14444, IBM Space Shuttle Orbiter Avionics Software, February 1977.