**Modeling Reconfiguration Algorithms for Regular Architectures**

by

Linda Sumners DeBrunner

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

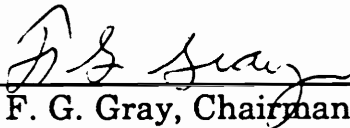
in partial fulfillment of the requirements for the degree of

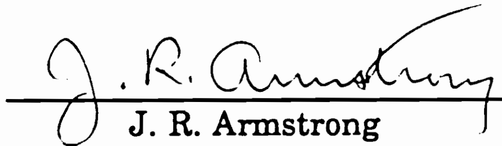
Doctor of Philosophy

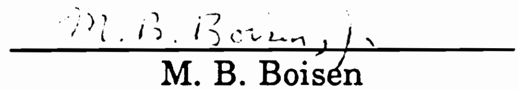
in

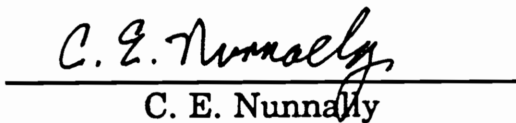
Electrical Engineering

APPROVED:


F. G. Gray, Chairman


J. R. Armstrong


M. B. Boisen


C. E. Nunnally

H. F. VanLandingham

July 1991

Blacksburg, Virginia

**Modeling Reconfiguration Algorithms for Regular Architectures**

by

Linda Sumners DeBrunner

F. G. Gray, Chairman

Electrical Engineering

(ABSTRACT)

Three models are proposed to evaluate and design distributed reconfigurable systems for fault tolerant, highly reliable applications. These models serve as valuable tools for developing fault tolerant systems. In each model, cells work together in parallel to change the global structure through a series of separate actions. In the Local Supervisor Model (LSM), selected cells guide the reconfiguration process. In the Tessellation Automata Model (TAM), each cell determines its next state based on its state and its neighbors' states, and communicates its state information to its neighbors. In the Interconnected Finite State Machine Model (IFSMM), each cell determines its next state and outputs based on its state and its inputs.

The hierarchical nature of the TAM and IFSMM provides advantages in evaluating, comparing, and designing systems. The use of each of these models in describing systems is demonstrated. The IFSMM is emphasized since it is the most versatile of the three models. The IFSMM is used to identify algorithm weaknesses and improvements, compare existing algorithms, and develop a novel design for a reconfigurable hypercube.

# Acknowledgements

I would like to thank Dr. F. G. Gray for his help and support, without which this work would have been much more difficult. He cultivated my sense of independence which was previously underdeveloped. I would like to thank Dr. J. R. Armstrong for kindling my interest in computer architecture. I would like to thank Dr. M. B. Boisen for showing me the beauty of algebra and for the insight which he provided into the graph theoretical aspects of this work. I would like to thank Dr. C. E. Nunnally for his consistent support in numerous ways throughout my graduate career. I would also like to thank Dr. H. F. VanLandingham for his interest in this work and for making me laugh through his classes. In addition, I would like to thank Dr. R. D. Riess for serving on my final examination committee. Finally, I would like to thank my husband, Victor, for his unceasing support during this work.

# Table of Contents

1.0 Introduction .... 1
1.1 Literature Review .... 2
1.1.1 Array Reconfiguration .... 2
1.1.1.1 Summary of Research at Virginia Tech .... 4
1.1.1.2 Survey of Research by Other Investigators .. 6
1.1.1.2.1 Configuration of Chips Around Fabrication Faults .... 6
1.1.1.2.2 Pattern Growth .... 9
1.1.1.2.3 Local Reconfiguration Around Faults .... 12
1.1.1.2.4 Other Related Research .... 16
1.1.2 Hypercube Reconfiguration .... 17
1.2 Summary .... 21
1.3 Structure of the Dissertation .... 21

2.0 Local Supervisor Model .... 24
2.1 Description of the Distributed Recovery Strategy .... 25
2.1.1 Basic Definitions .... 25
2.1.2 The Distributed Recovery Strategy Fault Model .... 39
2.1.3 The Distributed Recovery Strategy .... 46
2.2 Application of the Distributed Recovery Strategy to an Array Architecture .... 52
2.2.1 Description of the ADAS Model .... 55
2.2.2 Simulation Results .... 59
2.2.3 Related Proofs .... 71
2.2.4 Limitations of the Distributed Recovery Strategy ... 90
2.3 Description of the Local Supervisor Model .... 91
2.4 Use of the Local Supervisor Model .... 98
2.5 Summary .... 99

3.0 Tessellation Automata Model .... 101

3.1 Description of the Tessellation Automata Model .......... 101
3.1.1 Tessellation Automata .......................... 103
3.1.2 Reasons for Choosing Tessellation Automata as a Basis for a Model .......................... 104
3.1.3 Description of the Tessellation Automata Model .... 105
3.2 Formal Definition of the Tessellation Automata Model ..... 107
3.3 Using the Tessellation Automata Model to Describe Algorithms .......................... 112
3.3.1 Using TAM to Model the Direct Reconfiguration Algorithm .......................... 112
3.3.2 Using the TAM to Model the Distributed Recovery Strategy .......................... 120
3.3.3 Using the TAM to Model the LSM Array Algorithm .......................... 124
3.4 Summary .......................... 128

4.0 Interconnected Finite State Machine Model .................. 131
4.1 Description of the Interconnected Finite State Machine ..... 132
4.2 Formal Definition of the Interconnected Finite State Machine Model .......................... 136
4.3 Using the Interconnected Finite State Machine Model to Describe Algorithms .......................... 164
4.3.1 Modeling the Direct Reconfiguration Algorithm using the IFSMM .......................... 164
4.3.2 Modeling the DIAG-1 Diagnosis Algorithm Using the Interconnected Finite State Machine Model ....... 175
4.3.3 Modeling the White-Gray Algorithm Using the Interconnected Finite State Machine Model ....... 182
4.3.4 Modeling the Distributed Recovery Strategy Using the Interconnected Finite State machine Model .... 192
4.3.5 Modeling of the LSM Array Algorithm Using the Interconnected finite State Machine Model ........ 195
4.3.6 Using the Interconnected Finite State machine Model to Describe the Tessallation Automata Model .......................... 198
4.4 Summary .......................... 201

5.0 Using the Interconnected Finite State Machine Model to Improve Distributed Algorithms .......................... 203
5.1 Improving the Local Supervisor Model ................ 203
5.1.1 The Use of Spares to Respond to Errors .......... 204
5.1.2 The Problem of Predicting an Algorithm's Coverage .......................... 218

5.1.3 The Handling of Multiple and Near-Coincident Faults . . . . . 219
5.1.4 The Implementation of a Local Supervisor Model Algorithm . . . . . 234
5.2 Improving the Direct Reconfiguration Algorithm . . . . . 235
5.3 Summary . . . . . 242

6.0 Using the Interconnected Finite State Machine Model to Compare Distributed Algorithms . . . . . 245
6.1 A Methodology for Comparing Distributed Systems . . . . . 245
6.1.1 Important Features that Differentiate Algorithms . . 246
6.1.2 Procedure for Comparing Systems . . . . . 248
6.2 Comparison of the White-Gray Algorithm and the Array Reconfiguration Algorithm . . . . . 250
6.2.1 Level 1 Comparison . . . . . 251
6.2.2 Level 2 Comparison . . . . . 255
6.2.3 Level 3 Comparison . . . . . 255
6.3 Summary . . . . . 260

7.0 Using the Interconnected Finite State Machine Model to Design a Distributed Algorithm for a Particular Application . . . . . 262
7.1 The Design Procedure . . . . . 262
7.2 The Design Procedure Applied to a Specific Problem . . . . . 265
7.2.1 Development of the Candidate Architecture List . . . . 266
7.2.2 Design Constraints and Goals for the Chosen Application . . . . . 267
7.2.3 The Choice of a Redundant Structure . . . . . 269
7.2.4 Design Reconfiguration Algorithm . . . . . 284
7.3 Summary . . . . . 291

8.0 Conclusions . . . . . 294

Appendix A. ADAS CSIM Routines for the Array Reconfiguration Algorithm . . . . . 297

Appendix B. Level 3 TAM for the Direct Reconfiguration Algorithm . . . 311

Appendix C. IFSMM Level 3 General Routines . . . . . 329

Appendix D. Direct Reconfiguration Algorithm . . . . . 342

Appendix E. DIAG-1 . . . . . 369

Appendix F. White-Gray Local Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 412

Appendix G. Distributed Recovery Strategy . . . . . . . . . . . . . . . . . . . . . 445

Appendix H. Array Reconfiguration Algorithm . . . . . . . . . . . . . . . . . . . 463

Appendix I. Modifications to the Local Supervisor Model . . . . . . . . . . . 474
I.3 LSM Modification 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 474
I.2 LSM Modification 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 486
I.3 LSM Modification 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 492
I.4 LSM Modification 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 495
I.5 Simulation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 498

Appendix J. Modifications to the Direct Reconfiguration Algorithm . . . 505

Appendix K. White-Gray Local Algorithm/Array Reconfiguration Algorithm Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 526

Appendix L. Hypercube Reconfiguration Algorithm . . . . . . . . . . . . . . . 539

List of References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 551

Vita . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 564

# List of Illustrations

Figure 1. Handling Faults in Reconfigurable Arrays . . . . . . . . . . . . . . . 3

Figure 2. A Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

Figure 3. A Directed Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

Figure 4. A Subgraph of *G1* . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

Figure 5. An Induced Subgraph . . . . . . . . . . . . . . . . . . . . . . . . . . 31

Figure 6. Isomorphic Graphs . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

Figure 7. A Labeled Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

Figure 8. Two Graphs which are not L-Isomorphic . . . . . . . . . . . . . . . . 36

Figure 9. Facility graph for a Computing System . . . . . . . . . . . . . . . . . 37

Figure 10. Facility Graph for an Algorithm . . . . . . . . . . . . . . . . . . . . 38

Figure 11. Facility Graph for Faulty Computing System with Node a Faulty . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

Figure 12. 6-Computer Basic Fault-Tolerant System [Yan86] . . . . . . . . 41

Figure 13. Graph models for the BFS [Yan86] . . . . . . . . . . . . . . . . . . 42

Figure 14. Example of the Reconfiguration Problem . . . . . . . . . . . . . . . 45

Figure 15. The Distributed Recovery Strategy . . . . . . . . . . . . . . . . . . 47

Figure 16. Assumptions Underlying the Distributed Recovery Model [Yan86] . . . . 49

Figure 17. An Example showing the Distributed Recovery Strategy . . . 51

Figure 18. The Basic Graph . . . . 53

Figure 19. The Redundant Graph . . . . 54

Figure 20. The ADAS Hardware Graph corresponding to the Redundant Graph [Gra88] . . . . 57

Figure 21. The ADAS Hardware Graph Port Assignments [Gra88] . . . . 58

Figure 22. Error Response Assignment for Yanney-Hayes Algorithm [Gra88] . . . . 60

Figure 23. Reconfiguration for a Double Sequential Fault in Cells 9 and 6 [Gra88] . . . . 62

Figure 24. Summary of Coverage from ADAS Simulations [Gra88] . . . . 64

Figure 25. Time Anaylsus of Single Faults from ADAS Simulations [Gra88] . . . . 66

Figure 26. Time Analysis of Double Sequential Faults from ADAS Simulations (Part 1) [Gra88] . . . . 67

Figure 27. Time Analysis of Double Sequential Faults from ADAS Simulations (Part 2) [Gra88] . . . . 68

Figure 28. Time Analysis of Double Sequential Faults from ADAS Simulation (Part 3) [Gra88] . . . . 69

Figure 29. State Name Transformation . . . . 72

Figure 30. Error Response Set . . . . 73

Figure 31. Error Response Set Rules . . . . 74

Figure 32. The Neighborhood of the Redundant Graph [Whi88] . . . . 75

Figure 33. Case Division for Theorem 1 . . . . 79

Figure 34. Case Division for Theorem 2 . . . . . . . . . . . . . . . . . . . . . . . . . 84

Figure 35. Error Response Digraph . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

Figure 36. Constraints on Error Response Set Choice . . . . . . . . . . . . . . . 95

Figure 37. Guidelines for Error Response Set Choice . . . . . . . . . . . . . . . 96

Figure 38. Control Circuit for the Direct Reconfiguration Structure [Sam86] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114

Figure 39. Example of Direct Reconfiguration [Sam86] . . . . . . . . . . . . . 115

Figure 40. Neighborhood of TAM of Direct Reconfiguration . . . . . . . . . . 117

Figure 41. Facility Graph for the Array Reconfiguration Algorithm . . . 125

Figure 42. Graph for TAM of Array Reconfiguration Algorithm . . . . . . . 126

Figure 43. The Relationship between IFSMM cell and the Moore Machine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133

Figure 44. A Redundant Linear Array Block Diagram . . . . . . . . . . . . . 138

Figure 45. Facility Graph for the Redundant Linear Array . . . . . . . . . . 140

Figure 46. Redundant graph with Boundary Cells . . . . . . . . . . . . . . . . 141

Figure 47. Alternative Way to Add Boundary Cells . . . . . . . . . . . . . . . . 142

Figure 48. Direct Reconfiguration Algorithm Circuit [Sam86] . . . . . . . . 166

Figure 49. Direct Reconfiguration Algorithm Example [Sam86] . . . . . . 167

Figure 50. Procedure for Constructing a Redundant Linear Array [Hos89] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173

Figure 51. Redundant Linear Array with 3 Clusters of 3 Processors in which Clusters are 2-Connected [Hos89] . . . . . . . . . . . . . . . . . . . 174

Figure 52. DIAG-1 Diagnosis Algorithm [Hos89] . . . . . . . . . . . . . . . . . . . 176

Figure 53. The Redundant System for DIAG-1 . . . . . . . . . . . . . . . . . . . . 178

Figure 54. The IFSMM Graph for DIAG-1 . . . . . . . . . . . . . . . . . . . . . . 179

Figure 55. The Redundant System for the White-Gray Local Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 185

Figure 56. The IFSMM Graph for the White-Gray Local Algorithm . . . 186

Figure 57. Time Requirements of the Array Reconfiguration Algorithm for Single Faults . . . . . . . . . . . . . . . . . . . . . . . . . . . . 199

Figure 58. Time Requirements of the Array Reconfiguration Algorithm for Double Sequential Faults . . . . . . . . . . . . . . . . . . . . . . 200

Figure 59. Linear Array Example Illustrating the Use of Spares to Respond to Errors . . . . . . . . . . . . . . . . . . . . . . . . . . . . 206

Figure 60. E(2)E(3) for Linear Array . . . . . . . . . . . . . . . . . . . . . . . . 207

Figure 61. Error Response Digraph for Linear Array Example . . . . . . . 209

Figure 62. Graphs for 1-FT Tree System [Yan86] . . . . . . . . . . . . . . . . . 212

Figure 63. Error Response Digraph for 1-FT Tree System . . . . . . . . . . . 213

Figure 64. Graphs for 2-FT Tree System . . . . . . . . . . . . . . . . . . . . . . . . 214

Figure 65. Error Response Digraph for 2-FT Tree System . . . . . . . . . . . 215

Figure 66. Recovery Process for 2-FT Tree System to Node b Faulty (E(1) exists) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 217

Figure 67. Array Reconfiguration Algorithm for E(3,9) . . . . . . . . . . . . . 220

Figure 68. Array Reconfiguration Algorithm for E(1,4)--FAILS! . . . . . . 222

Figure 69. Reconfiguration for E(1,4) Showing Previous State Information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 224

Figure 70. Example showing Problem where More than One Cell has the Same Normal Computation State . . . . . . . . . . . . . . . . . . . . . . . 226

Figure 71. Example Showing Reconfiguration for E(1, 4) Using Second Modified Technique . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 231

Figure 72. Double Fault Coverage for Original and Modified Array Reconfiguration Algorithms . . . . . 233

Figure 73. The Facility Graph for the Improved Direct Reconfiguration Algorithm . . . . . 238

Figure 74. The Error Response Digraph for the Array Reconfiguration Algorithm which Mimics the White-Gray Local Algorithm . . . . . 257

Figure 75. Coverage and Time Requirements for the Array Reconfiguration Algorithm Modified to Mimic the White-Gray Local Algorithm . . . . . 259

Figure 76. Basic Graph -- $2^4$ Cube . . . . . 274

Figure 77. Added Connections . . . . . 277

Figure 78. Added Connections Between Subcubes . . . . . 278

Figure 79. Added Connections for Idea B . . . . . 281

Figure 80. Added Connections for 3-Space . . . . . 283

Figure 81. Hypercube with E(1) . . . . . 287

Figure 82. Final State after Reconfiguration around E(1) . . . . . 288

Figure 83. System with Fault Combination E(1,2) . . . . . 289

Figure 84. Final State after E(1,2) Reconfiguration . . . . . 290

Figure 85. System with Fault Combination E(1,4) --Reconfiguration Fails! . . . . . 292

Figure 86. The Example with Initial State[Sam86] . . . . . 355

Figure 87. DIAG-1 Example . . . . . 394

Figure 88. White-Gray Local Algorithm Example . . . . . 416

Figure 89. The Example System after Reconfiguration . . . . . 444

Figure 90. Cyclic Example for Distributed Recovery Strategy . . . . . 447

Figure 91. Reconfiguration Example for the Array Reconfiguration Algorithm ........................................ 466

# 1.0 Introduction

Fault tolerance is an important aspect of computer system design. The high levels of reliability needed for many applications require the use of reconfiguration techniques. To avoid hard core components that must be present for the system to operate correctly, distributed systems are required. This work addresses the problem of distributed reconfiguration.

To evaluate existing reconfigurable systems, an efficient, methodical approach is needed. A model for describing distributed reconfigurable systems would provide such an approach. The model should also be suitable for comparing systems. In addition, a procedure for designing distributed, reconfigurable systems is needed. A model that would support both analysis and design would be a valuable tool. This work proposes three models of this type.

## 1.1 Literature Review

To illustrate the use of the models, two principle architectures are used: an array and a hypercube. This chapter describes existing reconfiguration techniques for arrays and hypercubes.

### 1.1.1 Array Reconfiguration

Techniques have been developed for handling both fabrication faults and run-time faults. The steps involved in handling faults in reconfigurable arrays are summarized in Figure 1. In the beginning, an architecture is designed. The design process consists of simulation and redesign. This process results in a design to be fabricated which may or may not have design flaws. Many of the chips fabricated will be faulty. The chips can be tested to hopefully determine which chips are faulty. The bad chips can then be eliminated. In addition, techniques have been proposed which allow reconfiguration around the faulty processing elements so a chip does not need to have one hundred percent good processors to be usable.

After an initially good chip is identified, a computation graph can be embedded in the array. The pattern growth algorithm may or may not grow around faulty elements. After the computation graph is embedded into the processing array, the architecture can be used for processing.

If a fault occurs while a computation is taking place, two things can occur. The array could continue without detecting the fault--errors may result. Or, the fault can be detected and appropriate action taken. If the fault is

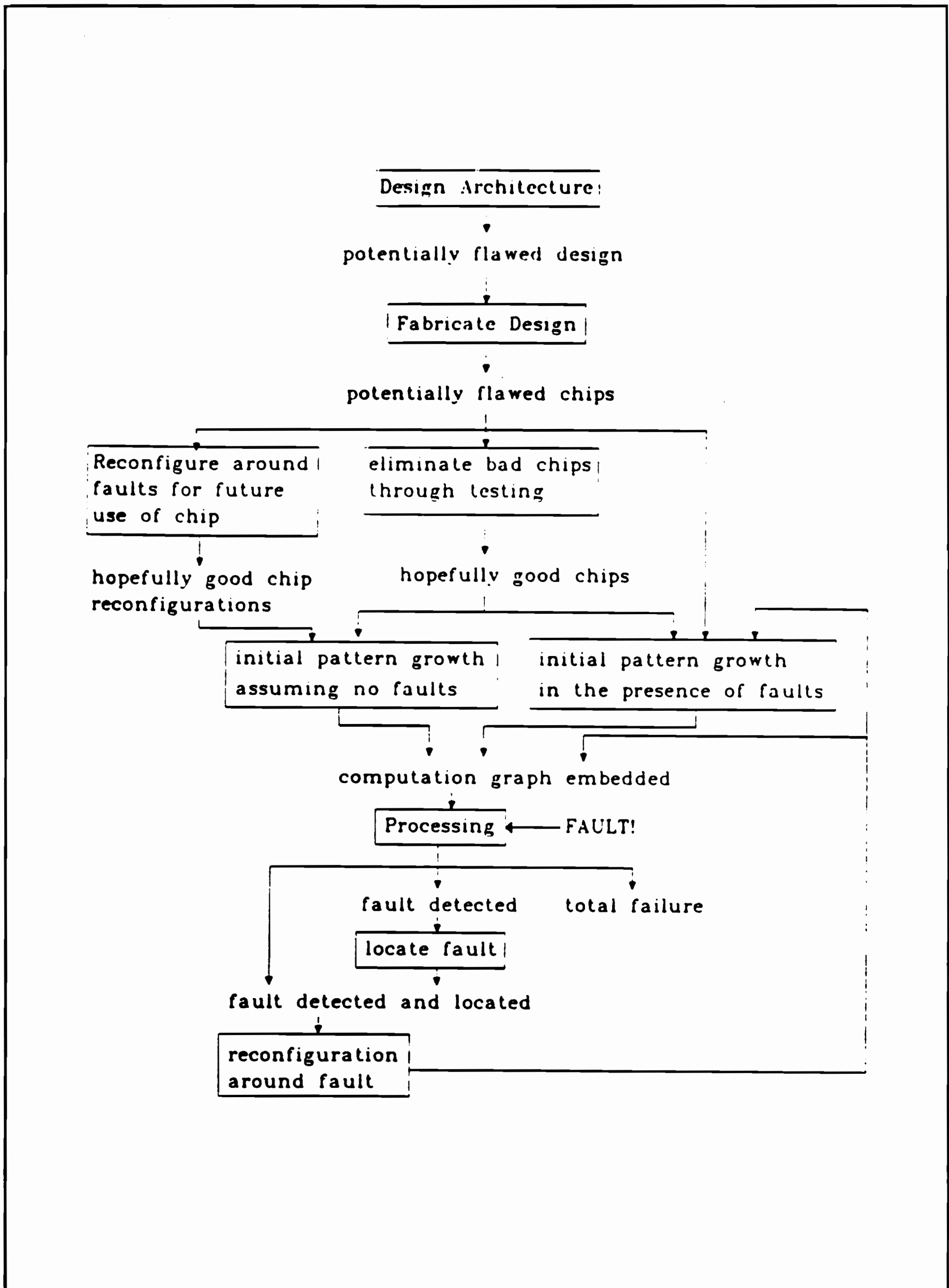



**Figure 1.** Handling Faults in Reconfigurable Arrays

detected the entire array could be replaced or the fault can be located and some kind of reconfiguration can occur.

When a fault is detected, the processing could halt and the computation graph could be embedded in a fault-free portion of the array. Another approach is to locally reconfigure around the fault and continue the processing. The local reconfiguration approach is typically less complex and requires less time. Chean [Che90] presents a taxonomy for classifying reconfigurable arrays.

Researchers at Virginia Tech have been addressing the problem of handling faults in arrays of processors for many years. First, research at Virginia Tech related to the reconfiguration of processors will be discussed, then research by other researchers.

#### 1.1.1.1 Summary of Research at Virginia Tech

For many years, researchers at Virginia Tech have been working on reconfigurable arrays for fault-tolerance. Martin [Mar80] addresses the problem of automatic reconfiguration with large neighborhood size. Tessellation automata theory is used to describe the problem. The properties of tessellation, or cellular, automata are discussed in [Yam70] [Yam71] [Amo72a] [Amo72b] [Amo74] [Tof87] [Pre84] [Cod68] [Bur70].

Walters, et al. [Wal81] show that any finite state machine can be implemented in a self-diagnosing cellular space. Rather than implementing the function and then making the implementation fault-tolerant, the approach

presented combines the implementation of the function and implementation of fault-tolerance into the same step. Each cell in the automaton is divided conceptually into two components: a computational element and a control element.

Then, Gray [Gra82] presents an architecture that can reconfigure itself in response to user requests and fault detection. The algorithms presented are distributed and dynamic in nature. An approach to pattern growth is given, and a formal model of the control structure is presented using tessellation automata.

Kumar and Gray [Kum84] integrate previous test/quarantine, pattern-growing and reconfiguration algorithms into one single theory. A unidirectional growth algorithm and an alternate reconfiguration algorithm are also presented. Small neighborhood size is used to reduce the interconnection complexity so the system is realizable.

Gollakota and Gray [Gol84] address the problem of reconfiguring an array into any arbitrary interconnection pattern using distributed control. A distributed mechanism is used to clear the array, and a cancerous growth mechanism, which is also distributed in nature, is used to grow the desired configuration. Brighton [Bri85] extends and simplifies the algorithms of Kumar.

Several other related areas have also been developed. Connell [Con85] presents an algorithm to grow a path from the computation array to the

terminals for the design of Kumar. Zaidi [Zai88] proposes an algorithm to grow a path between modules for the design of Kumar. Gray and White [Gra89] specify a method for local reconfiguration around faults to be used with Kumar's design and the related algorithms.

This research at Virginia Tech has prepared a foundation for development of a more theoretical local reconfiguration theory.

#### 1.1.1.2 Survey of Research by Other Investigators

In this section, research involving the reconfiguration of arrays of processors will be described. Systolic arrays, a subset of arrays of processors, are described in [Kun82]. Reconfiguration of arrays of processors can be divided into three principal areas: configuration of chips around fabrication faults, pattern growth and local reconfiguration around faults. This work primarily involves local reconfiguration around faults; however, the ideas developed in the other areas of array reconfiguration are related to the local reconfiguration problem and have been included for completeness. The last part of this section will describe other related research.

##### *1.1.1.2.1 Configuration of Chips Around Fabrication Faults*

One way to increase the production yield for chips is to reconfigure around faulty areas of the chips. The reconfiguration methods for handling fabrication faults are permanent. In addition, centralized control is typically used because the fault-tolerance of the fabrication process itself is not critical.

Lopez-Benitez and Fortes [Lop89] discuss a method of modeling fault-tolerant processor arrays which uses a modified stochastic petri net.

Two of the most important considerations involved in developing a chip which can be reconfigured around fabrication faults are area requirements and yield. Koren and Breur [Kor84] evaluate the chip area cost of fabrication fault-tolerance techniques and consider the overall effects on yield. Although the increase in chip area for redundancy may reduce the yield, the redundancy itself may be used to increase the yield.

Lam, et al. [Lam89] compare two approaches for reconfinguring systolic arrays which use cuts to determine which processing elements to bypass. Typically, some non-faulty cells are removed, as well as the faulty cells. Morton and Abreu [Mor86] present a chip design for an array of bit-slice processors which can be reconfigured around faulty processors to form bit-parallel or bit-serial words. The underlying structure of this SIMD architecture is a linear array. A usable chip can have no more than four faulty processors (that is, at least sixteen out of twenty must be good). In addition, a usable chip can have no more than two adjacent faulty processors because of restrictions on pin reconfiguration logic.

Chevalier and Saucier [Chev86] propose a programmable switch network for a Wafer-Scale Integration processor array. Saucier, et al. [Sau89] describe the hierarchical yield enhancement approach used for the European Large SIMD Array (ELSA), a Wafer-Scale Integration processor array.

Rushton and Jesshope [Rus86] propose a flexible processor array which allows bit-parallel operations, as well as bit-serial operations. Moore, et al. [Moo86] describe a yield enhancement approach for a bit-level systolic array which addresses the problems of input/output routing, testing, and clock distribution, as well as the problem of bypassing defects.

Kuo and Fuchs [Kuo87] address the problem of assigning spare rows and columns efficiently to reconfigure a VLSI array. Graph theory is used in the analysis. Two different algorithms for allocating spares are given. Hasan and Liu [Has88] present a method which uses critical sets to reconfigure arrays by swapping rows and columns with spare rows and columns. This algorithm works in polynomial time to determine the minimum set of rows and columns which must be replaced.

Singh [Sin88] presents a method of adding redundant elements to a large area VLSI processor array. This method is only useful if each cell is complicated enough so it has a relatively high probability of being defective. Although this method could be applied to other topologies, this paper concentrates on rectangular arrays. Singh [Sin87] presents a fault tolerant modular binary tree architecture. This strategy allows spares to be shared across the boundaries of the modules.

Lee and Frieder [Lee86] address the problem of embedding computation graphs into large arrays which have many faulty cells. Fabrication faults are

assumed to be clustered together. This approach is related to the pattern growth approaches which are described in a later section.

Trotter and Moore [Tro89] describe an array for image processing which uses algorithmic fault tolerance and graceful degradation to tolerate fabrication faults. Banerjee and Abraham [Ban86a] present a probabilistic analysis of algorithmic fault tolerance. Algorithmic fault tolerance for arrays is discussed in [Cos88]. Anfinson and Luk [Anf88] propose a linear algebraic model for algorithmic fault tolerance. The use of coding in the design of fault tolerant systolic arrays is presented in [Che86]. Reconfiguration of Random Access Memory (RAM) around fabrication faults is an important area of reconfiguration. Although the techniques are applied to memory cells rather than to processors, many of the techniques give insight into reconfiguration for processor arrays. Lombardi and Huang [Lom88] present techniques for reconfiguring (or repairing) RAM which use redundant rows and columns of spares. Mazumder and Yih [Maz89] describe a method for reconfiguring memory arrays which uses neural nets.

*1.1.1.2.2 Pattern Growth*

Another area of reconfiguration for arrays is pattern growth. A pattern growth algorithm embeds a computation graph into a fault-tolerant graph. Some pattern growth algorithms are developed so an architecture can be used for more than one computation graph. Other pattern growth algorithms are developed to embed a computation graph in a potentially faulty array. If local

reconfiguration fails, a pattern growth algorithm can be used to "grow" the computation graph in another part of the processor array. It is possible that pattern growth algorithms developed to allow more than one computation graph to be used with a particular architecture could be extended to "grow around" faulty processors.

Snyder [Sny82] discusses a very flexible array architecture called the Configurable Highly Parallel (Chip) computer which can be used to implement algorithms requiring different architectures. In other words, the Chip computer uses a pattern growth algorithm to embed the desired computation graph. This architecture could be modified to handle faulty processors. Gupta and Soffa [Gup87] discuss a reconfigurable long instruction word computer.

De Groot, et al. [DeG87] discuss the SPRINT (Systolic Processor with a Reconfigurable Interconnection Network of Transputers) which was developed to experimentally evaluate different systolic architectures and algorithms. The SPRINT architecture can operate as either an SIMD or an MIMD system.

Koren [Kor81] attacks the problem of embedding computation graphs into a fault-tolerant reconfigurable array. Each cell in the proposed architecture consists of both an application processor and a communication processor. The algorithms developed were chosen to be simple based on the projection that the failure rate of a processor on a VLSI chip is low and since more complex algorithms would require more memory and thus reduce the total potential array size. The algorithms presented assume a rectangular grid

as the array structure. Sengupta, et al. [Sen87] present a method of designing optimally fault tolerant regular architectures which have a small diameter.

Rosenberg [Ros83] developed Diogenes, an approach for developing fault tolerant arrays of large, identical processing elements which can be easily tested. This approach uses a linear array of processing elements which are connected by a bus structure. Chung, et al. [Chu83] describe a method for developing Diogenes designs for any particular interconnection network; that is, they address the problem of embedding a computation graph in a Diogenes design. Koren [Kor86] clarifies several differences between his approach and Diogenes.

Varman, et al. [Var84] present a fault tolerant VLSI processing array used for matrix multiplication. This approach incorporates both hardware and software. The architecture presented is similar to the Chip architecture.

Varman and Ramakrishnan [Var86] present a fault tolerant processing array which is tailored to the problem of matrix multiplication. This approach is somewhat of an extension of the Diogenes approach.

Menzilcioglu, et al. [Men89] describe the evaluation of a two-dimensional array of powerful processors. Their design uses multiplexed communication channels. A mapping algorithm is used to map the logical array to the physical array.

Gordon [Gor87] also discuss the problem of embedding binary trees in arrays. Hassan and Agarwal [Has86] present a modular fault tolerant binary

tree architecture. Distributed fault tolerance in trees is discussed in [Hos87]. Yield enhancement for trees is discussed in [How88]. Howells and Agarwal [How87] propose a reconfiguration scheme to increase the yield of binary tree architectures as well as increase the reliability of the operating system. An approach to reconfigurable tree architectures which uses a subtree oriented approach is discussed by Lowrie and Fuchs [Low87]. The problem of reconfiguring embedded graphs in hypercubes in the presence of faults is discussed in [Che88].

*1.1.1.2.3 Local Reconfiguration Around Faults*

The last area of reconfiguration in systolic arrays to be considered is local reconfiguration around faults. Many diverse approaches to the problem have been proposed. These techniques can also be used to reconfigure around fabrication faults.

Sami and Stefanelli [Sam83] present several reconfigurable processing array structures. Reconfiguration is viewed as a logical renaming of the cells to assign tasks. These structures have different ways of including spare elements: by adding a column of spares, by adding two columns of spares and by adding a row of spares and a column of spares. Sami and Stefanelli [Sam85] extend these ideas in their "fault-stealing" approach. In this approach, when a processing element is identified as being faulty, reconfiguration takes place along that element's row with each processor taking over the duties of its neighbors until a spare element is reached. When

reconfiguration is halted along the row, spares are "stolen" from the row below. These approaches are also discussed in [Sam86]. Lombardi, et al. [Lom87] present an index mapping approach using fault stealing. Gentile, et al. [Gen84] present a switch design for such reconfigurable arrays. Distante, et al. [Dis89], using the same idea of logical renaming used in the designs presented above, present an algorithm that considers faulty interconnection links explicitly, as well as faulty processing elements.

In [Sam84], Sami and Stefanelli present a very different approach which is based on time-redundancy. With this technique the responsibilities of the faulty processor are taken over by another good processor with its own responsibilities rather than by a spare. Spare processing phases are used. Although this approach results in a much slower array, there is very little increase in the area required for the chip. Many of these approaches are summarized in [Neg86]. Antola, et al. [Ant88] integrate structure redundancy and time redundancy techniques into a single reconfigurable array design. In separate research, Majumdar, et al. [Maj90] propose a linear array which employs triple time redundancy. Each computation is performed by three processors and the results are voted. Graceful degradation takes place.

Uyar and Reeves [Uya85] [Uya88] address the problem of achieving fault tolerance in MIMD processing arrays through task redistribution. Although only single faults are considered, the algorithms could be extended

to cover multiple faults. This approach assumes a toroidally connected near-neighbor mesh.

Ishikawa, et al. [Ish86] describe the Hierarchical Array Processor (HAP), a powerful and reliable computer which is based on the nearest neighbor mesh MIMD processing array.

Kung, et al. [Kun86] present a distributed reconfiguration algorithm which can handle both transient and permanent faults. The algorithm assumes a switch lattice similar to that used in the Chip [Sny82]. Each processing element is divided into a communication part and a computation part. The reconfiguration algorithm uses the concept of compensation paths which are grown either horizontally or vertically depending on the state of the faulty processing element. Further discussion is provided in [Kun87]. Jean and Kung [Jea89] propose an array grid based on single-track switches. This grid structure has an associated globally-controlled reconfiguration algorithm for yield enhancement and a distributed reconfiguration algorithm for run-time reconfiguration.

Kim and Reddy [Kim87] [Kim88] present an approach for designing Easily Testable And Reconfigurable (ETAR) processing arrays which is similar to Diogenes.

Wang and Nelson [Wan87] present a novel approach to reconfiguration. Algorithms are presented for reconfiguration around one faulty processor and reconfiguration around two faulty processors. This approach describes the

processing elements as elements in the Galois Field GF($q^m$). The major disadvantage of this method is the requirement that each cell be able to be connected to any other cell.

Melhem [Mel89] describes a reconfiguration scheme for fault tolerant arrays which is composed of two phases. One phase is a local reconfiguration which reconfigures quickly when response time is strictly constrained. The other phase is a global reconfiguration which takes place when the restraints on response time are more relaxed. The second phase optimizes the reconfiguration to increase the probability of being able to reconfigure around future faults.

Hosseini [Hos89] describes a fault tolerant array design which is based on the idea of grouping the processors into clusters. Li, et al. [Li87] describe a data driven approach to array reconfiguration. Methods of designing reconfigurable cube-connected cycles architectures are presented in [Ban86c] and [Ban88].

Dutt and Hayes [Dut88] discuss reconfiguration strategies for tree architectures. Their work strives to minimize the number of spare nodes and edges which is needed. In later work [Dut89], they present a general approach to the reconfiguration of multiprocessors which are modeled by graphs.

Tsunoyama and Naito [Tsu88] present a general strategy for reconfiguring architectures which are modeled by a linear cellular automaton.

Maehle, et al. [Mae86] describe a graph model which is used for diagnosing and reconfiguring fault tolerant multiprocessor systems.

*1.1.1.2.4 Other Related Research*

The problem of fault diagnosis in multiple processor systems is discussed in [Ban86b] and [Chu86]. The problem of testing in arrays is discussed in [Chen86], [Sci88a], and [Sci88b]. LaForge [LaF89] considers the fault tolerance of arrays for different fault distributions.

Chen, et al. [Ch86] present a design using a crossbar switch to create an arbitrary network of bit-serial elements which can be easily changed. Popli and Bayoumi [Pop88] present an approach which specifies the testing and location of faults, as well as a globally controlled reconfiguration algorithm. Kumar and Tsai [Kum89] present an approach for mapping computations performed on two-dimensional systolic arrays to one-dimensional systolic arrays. Nayak, et al. [Nay90] characterize catastrophic fault patterns in linear systolic arrays.

The general reconfiguration of multicomputer systems are discussed in [Dav82], [Kar81], [Kar83], [Kar86], and [Kar87]. Yalamanchili and Aggarwal [Yal85] describe reconfiguration techniques for parallel architectures encompassing fault tolerant techniques and reconfiguration for computational structure. Pradhan [Pra85] discusses the use of general reconfigurable networks which also employ fault tolerance. An operating system for a reconfigurable multiprocessor system is discussed in [Bro86]. Fault tolerant

interconnection networks are discussed in [Dug86], [Lin86], [Sie79], and [Jin86]. A reconfiguration algorithm for a double-loop token-ring local area network is described by Rom and Shacham [Rom88].

Shombert and Siewiorek [Sho87] introduce a method of roving spares which is used to increase the availability of the system. These roving spares are used to increase the testing capabilities of the system.

Hayes [Hay76] presents a graph model for representing fault tolerant computing systems. Meyer and Pradhan [Mey88] present a graph theoretical approach to fault tolerant systems and introduce a new architecture called flip-trees. Ramamoorthy and Ma [Ram86] describe a stochastic model for systems in which failed components are not repaired. Sengupta, et al. [Sen87b] propose an approach for designing fault tolerant regular networks that allow any number of nodes and any number of connections per node, as long as there are more nodes than connections per node.

### 1.1.2 Hypercube Reconfiguration

The hypercube has emerged as an important architecture. It is a distributed-memory, message-passing multiprocessor in which homogenous processors coordinate their activities through sending messages through an interconnection network [Hea85]. The hypercube has been the focus of much investigation [Hay86a] [Hay86b] [Hay87] [Sei85] [Wil87] [Kol85]. Several related architectures have emerged--the perfect shuffle interconnection network [Sto71], the indirect binary n-cube microprocessor array [Pea77], the

multistage cube [Sie81], the cube-connected cycles architecture [Pre81], and the extra stage cube interconnection network [Ada84]. The steps involved in handling faults in hypercubes are basically the same as in arrays. Reconfigurable hypercube architectures have received some attention recently. The natural fault tolerance characteristics of the Boolean n-cube have been studied thoroughly.

Gray, et al. [Gra88] studied the fault tolerant characteristics of the Jet Propulsion Laboratory Hypercube. Armstrong and Gray [Arm81] have shown that the Boolean n-cube is n (one step) diagnosable. They also present a test algorithm which can diagnose any pattern of n faulty processors. Chu and Armstrong [Chu86] present a method for non-hardcore system level fault diagnosis using a testable diagnosis array which requires (n+1) clock periods for a Boolean n-cube. After diagnosis is performed, the system operates in a degraded mode with longer communication paths. The degraded mode involves a change in topology. For related interconnection networks which are n-cube-connected, such as the indirect binary n-cube network [Pea77], Dilger and Ammann [Dil84] propose an approach to distributed system level self-diagnosis in which the diagnostic information is provided by the interconnection structure of the network. Bhat [Bha83] presents a hypercube diagnosis algorithm which is $O(n^3)$. Abraham and Padmanabhan [Abr88] analyze the reliability and degradation of the hypercube based on embedding functional subcubes of different sizes.

The redundancy inherent in the hypercube architecture allows the system to degrade gracefully in the presence of faults [Ren86]. The robustness of the hypercube has been studied by Becker and Simon [Bec86]. Yang, et al. [Yan88] consider the network reliability of the hypercube assuming that all nodes are perfectly reliable, all links are equally likely to fail, and the failure of links is independent. Peercy and Banerjee [Pee90] describe a shortest-path, deadlock-free routing algorithm for faulty hypercubes. Chen and Shin [ChMS88] also address the problem of routing in a faulty hypercube. A framework for analyzing routing in faulty hypercubes is described by Gordon and Stout [Gor88]. Lee and Hayes [Lee88] present algorithms for routing and broadcasting in faulty hypercubes which rely on each node having limited information about the nodes in a defined neighborhood. Harry, et al. [Har88] describe the fault tolerant communication system for the B-HIVE generalized hypercube. Natural redundancy methods are also suitable for the general class of hypercube structures presented by Bhuyan and Agrawal [Bhu84]. When natural redundancy is used with graceful degradation, the topology of the system changes. However, in this approach the full capacity of the system is used. This approach is best for dispersed failures.

Rennels [Ren86] considers several issues involved in hypercube fault tolerance. An approach for concurrent fault detection and techniques for making the hypercube redundant are presented. The reconfiguration method presented uses extra connections in the (n+1)th dimension. Spares may be

general or assigned to a particular sub-cube. Crossbar switches are used to make the necessary connections. In each case, the topology is preserved and the system operates without degradation. Although these reconfiguration approaches yield a highly available system, the results are not adequate for long-life unmaintained systems. For these demanding applications some form of hierarchical redundancy is required. Rennels presents such a hierarchical approach.

Chau and Liestman [Cha89] present a reconfigurable hypercube design which uses spares assigned to subcubes. Unlike the crossbar switching approach proposed by Rennels, this approach uses a multistage switching network. This approach maintains the topology.

Aykanat and Ozguner [Ayk87] describe a conjugate gradient algorithm for a hypercube which uses mathematical properties of the algorithm to detect faults. Banerjee, et al. [Ban88] describe an approach for algorithm-based fault detection and present a reconfiguration strategy which uses spare nodes and links. In the reconfiguration strategy, some delay is added by using spare nodes for switching. This strategy covers all single faults. Banerjee and Stunkel [Ban88b] describe the use of algorithmic fault tolerance for Gaussian Elimination using a hypercube architecture. In a later paper, Banerjee [Ban90] describes two different strategies for reconfiguring hypercubes. These strategies use logical links to implement the architecture where each logical link is mapped to one or more physical links. Some performance degradation

results from this mapping process. In the first strategy, spare nodes are attached to selected processors. In the second, approach spares are inserted into links. Das and Kim [Das89] present an analytical model for computing task-based availability for a hypercube based on the minimum number of nodes required for a task. A task-based reliability model is also developed by Kim, et al. [Kim89].

Chen, et al. [Che88b] present reconfiguration algorithms for loops and multi-dimensional grids embedded in a hypercube architecture. Dutt and Hayes [Dut88b] address the problem of allocating subcubes. Kandlur and Shin [Kan88] address the subcube allocation problem for hypercubes with node failures.

## 1.2 Summary

Three models are developed as part of this work: the Local Supervisor Model, the Tessellation Automata Model, and the Interconnected Finite State Machine Model. These models are described in detail in following chapters. To illustrate the use off these models, reconfigurable arrays and hypercubes are considered. Reconfigurable arrays have been well-studied, while reconfigurable hypercubes have received less investigation. The importance of these two architectures makes them good choices to illustrate the models' uses.

## 1.3 Structure of the Dissertation

This dissertation proposes three models as the foundation for evaluating and designing distributed reconfigurable systems for fault tolerant, highly

reliable applications: the Local Supervisor Model, the Tessellation Automata Model, and the Interconnected Finite State Machine Model. The use of each model is discussed; however, the use of the most general of these models, the IFSMM, will be demonstrated most fully.

Chapter 2 describes the development of the Local Supervisor Model (LSM). In the LSM, local supervisors work together in parallel to change the system's global structure through a series of separate actions that require only short communication paths between close neighbors. The use of the LSM in comparing and designing systems is discussed. A redundant array architecture is used to demonstrate the LSM's value by developing a reconfiguration algorithm, referred to as the LSM Array Algorithm.

Chapter 3 introduces the Tessellation Automata Model (TAM) which overcomes the principle limitations of the LSM. In the TAM, each cell determines its next state based on its current state and its neighbors' current states. Every time step, each cell communicates its state information to all of its neighbors.

Chapter 4 presents the Interconnected Finite State Machine Model (IFSMM) which is more convenient than the TAM. In the IFSMM, each cell determines its next state and the outputs to its neighbors based on its current state and inputs from its neighbors. For applications in which a cell sends different information to various neighbors, the IFSMM is more convenient to use than the TAM. In addition, it clarifies the algorithm by separating output

information and state information. Several applications of the IFSMM are described.

Chapter 5 explains the use of the IFSMM in improving distributed algorithms and strategies. Two examples are presented. First, the reconfiguration strategy used in the Local Supervisor Model is enhanced. An improved version of the Array Reconfiguration Algorithm presented in Chapter 2 is described. Second, the Direct Reconfiguration Algorithm [Sam86] is improved. Thus, the IFSMM can be used to evaluate and improve reconfiguration approaches, as well as specific reconfiguration algorithms.

Chapter 6 describes the use of the IFSMM to systematically compare distributed algorithms. To illustrate the method, the White-Gray Local algorithm [Whi88] is compared to the improved Array Reconfiguration Algorithm described in Chapter 5.

Chapter 7 describes the use of the IFSMM to design a distributed algorithm for a particular application. All aspects of the design process are considered. A reconfigurable hypercube architecture is developed to illustrate the design procedure. Finally, Chapter 8 develops conclusions and outlines directions for future research.

# 2.0 Local Supervisor Model

Yanney and Hayes [Yan82] [Yan86] developed a distributed recovery strategy. They used this recovery strategy as the basis of reconfiguration algorithms for specific tree and loop networks. In this research, the same strategy is used as the basis for a reconfiguration algorithm for a specific array architecture.

Our initial interest in this strategy was as an application problem to be used in the analysis of the role of modeling tools in the design of highly reliable, highly parallel systems [Gra88]. This recovery strategy can be used as the basis for reconfiguration algorithms for different architectures. However, as the strategy was used, its potential as a general model of the reconfiguration process became evident.

In this chapter, the distributed recovery strategy presented by Yanney and Hayes is described. Then, the development of an array reconfiguration algorithm based on this strategy and the problems encountered is discussed.

Finally, a Local Supervisor Model based on the Yanney-Hayes distributed recovery strategy is defined, and its use is discussed.

## 2.1 Description of the Distributed Recovery Strategy

The Yanney-Hayes Distributed Recovery Strategy can be used as the foundation of reconfiguration algorithms for diverse architectures. In this section, the Distributed Recovery Strategy is discussed. First, some basic definitions are stated. Then, the fault model used with the strategy is described. Last, the strategy itself is presented.

### 2.1.1 Basic Definitions

Basic definitions related to graphs are given by Bondy and Murty [Bon76]. A graph is an ordered triple of the form $G = (V(G), E(G), \Psi_G)$. An example graph $G1$ is shown in Figure 2. $V(G)$ is a non-empty set of vertices, or nodes. For the example graph, $V(G1) = \{t, u, v, w, x\}$. $E(G)$ is a set of edges which is disjoint from $V(G)$. For the example graph, $E(G1) = \{a, b, c, d, e, f, g\}$. $\Psi_G$ is an incidence function which associates an **unordered** pair of vertices with each edge. Since the incidence function associates an unordered pair of vertices with each edge, the notation $uv$ means the same as $vu$. For the example graph, the incidence function $\Psi_{G1}$ is defined as follows:

$\Psi_{G1}(a) = tu$
$\Psi_{G1}(b) = tu$
$\Psi_{G1}(c) = tv$
$\Psi_{G1}(d) = uw$
$\Psi_{G1}(e) = vx$

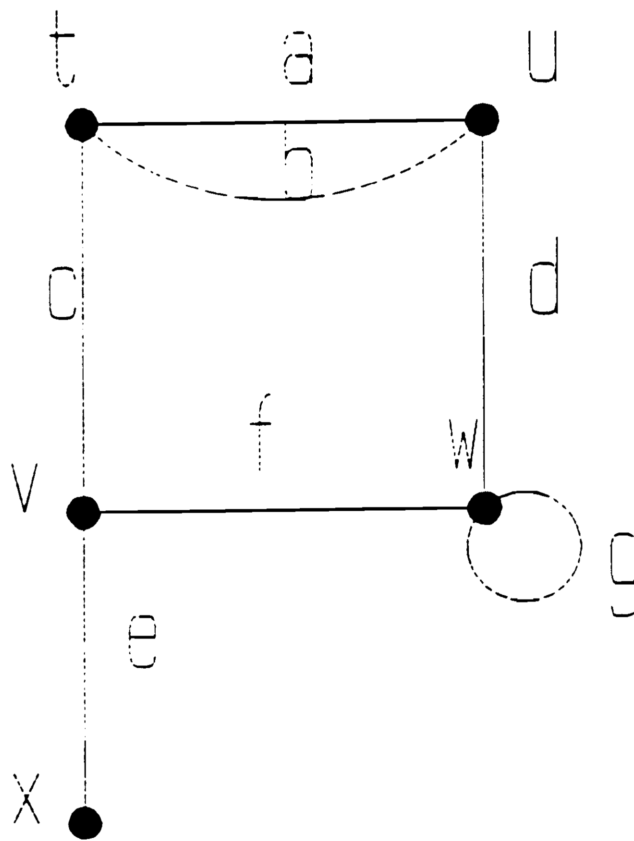



**Figure 2.** A Graph

$\Psi_{G1}(f) = vw$
$\Psi_{G1}(g) = ww$

The vertices associated with an edge $e$ are called the ends of $e$. For example, vertices $u$ and $w$ are the ends of edge $d$. An edge is said to join the vertices which are its ends. So, edge $d$ joins $u$ and $w$. An edge and its ends are said to be incident. So, $d$ and $u$ are incident, and $d$ and $w$ are incident. A link is an edge with distinct ends. A loop is an edge with identical ends. For example, $d$ is a link, and $g$ is a loop. A simple graph is a graph which has no loops and no more than one edge joining each pair of vertices. The graph $G1$ is not a simple graph.

A directed graph, or digraph, is a graph in which each link is assigned a direction, or orientation. More formally, a digraph is an ordered triple of the form $D = (V(D), A(D), \Psi_D)$. $V(D)$ is a non-empty set of vertices. $A(D)$ is a set of arcs which is disjoint from $V(D)$. An example digraph $D1$ is given in Figure 3. In this example, $V(D1) = \{t, u, v, w, x\}$, while $A(D1) = \{l, m, n, o, p, q, r\}$. $\Psi_D$ is an incidence function which associates an ordered pair of vertices with each arc. For digraph $D1$,

$\Psi_{D1}(l) = (t,u)$
$\Psi_{D1}(m) = (u,t)$
$\Psi_{D1}(n) = (v,t)$
$\Psi_{D1}(o) = (w,u)$
$\Psi_{D1}(p) = (w,v)$
$\Psi_{D1}(q) = (v.x)$
$\Psi_{D1}(r) = (w,w)$

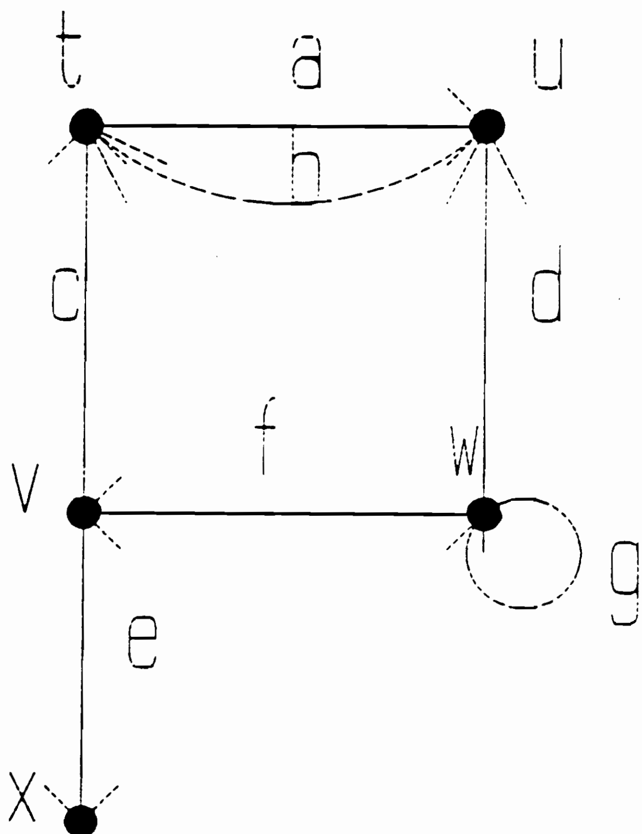



**Figure 3.** A Directed Graph

Suppose $u$ and $v$ are vertices, and $a$ is an arc such that $\Psi_D(a) = (u,v)$. Then $u$ is called the tail of $a$, and $v$ is called the head of $a$. For example, $v$ is the tail of $q$, and $x$ is the head of $q$. Each digraph $D$ has an underlying graph $G$. The vertex sets of $D$ and its underlying graph $G$ are identical. And, for each arc in $D$, there is an associated edge in $G$ with the same ends. Graph $G1$ (given in Figure 2) is the underlying graph of digraph $D1$ (given in Figure 3).

A graph $H = (V(H), E(H), \Psi_H)$ is a subgraph of $G = (V(G), E(G), \Psi_G)$ if $V(H)$ is a subset of $V(G)$, $E(H)$ is a subset of $E(G)$, and $\Psi_H$ is the restriction of $\Psi_G$ to $E(H)$. The graph shown in Figure 4 is a subgraph of $G1$. The induced subgraph $G[V']$ is the subgraph of $G$ with vertex set $V'$ whose edges are the edges of $G$ for which both ends are elements of $V'$. Let $V' = \{t, u, v, w\}$. Then, Figure 5 shows the induced subgraph $G1[V']$.

Let $G = (V(G), E(G), \Psi_G)$. Let $H = (V(H), E(H), \Psi_H)$. Then, $G$ and $H$ are isomorphic if and only if there exists bijections $\Theta$ and $\Phi$ where $\Theta: V(G) \rightarrow V(H)$ and $\Phi: E(G) \rightarrow E(H)$ such that $\Psi_G(e) = uv$ if and only if $\Psi_H(\Phi(e)) = \Theta(u)\Theta(v)$. The two graphs shown in Figure 6 are isomorphic. The graph $G2$ is defined as follows:

$V(G2) = \{a, b, c, d\}$
$E(G2) = \{l, m, n, o, p\}$

$\Psi_{G2}(l) = ab$
$\Psi_{G2}(m) = bb$
$\Psi_{G2}(n) = bd$
$\Psi_{G2}(o) = bc$
$\Psi_{G2}(p) = cd$

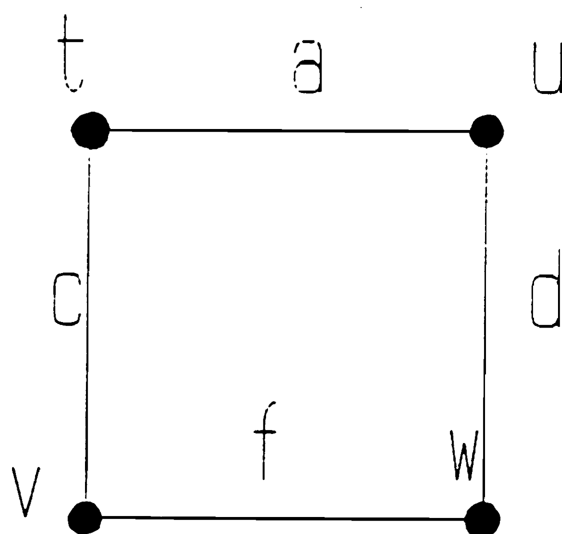



**Figure 4.** A Subgraph of *G1*

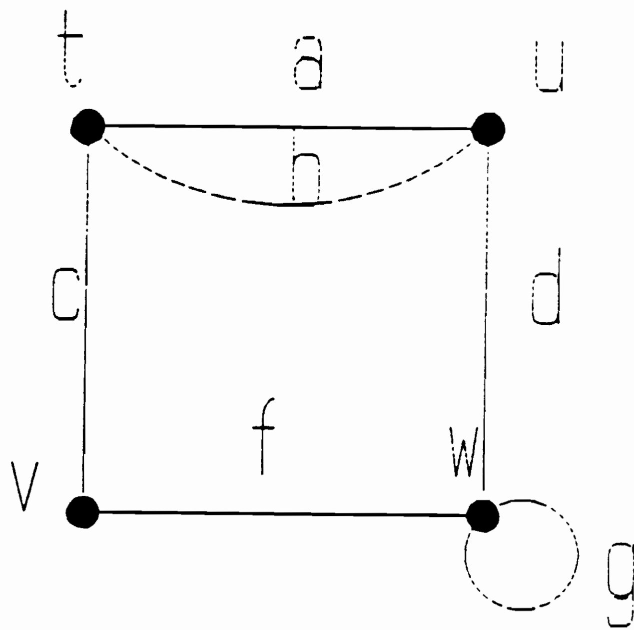



**Figure 5.** An Induced Subgraph

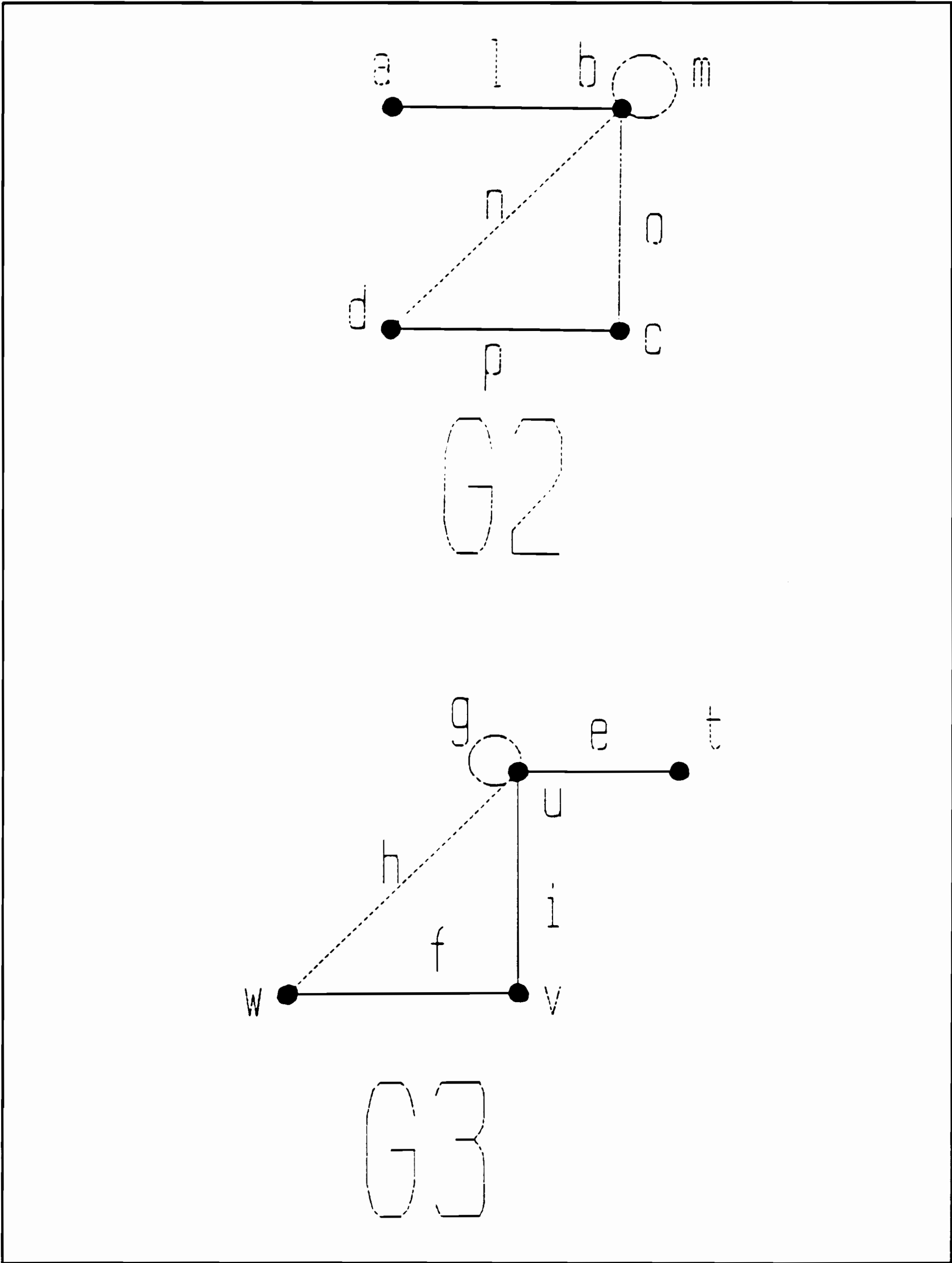



**Figure 6.** Isomorphic Graphs

And, the graph *G3* is defined as follows:

$V(G3) = \{t, u, v, w\}$
$E(G3) = \{e, f, g, h, i\}$

$\Psi_{G3}(e) = tu$
$\Psi_{G3}(f) = vw$
$\Psi_{G3}(g) = uu$
$\Psi_{G3}(h) = uw$
$\Psi_{G3}(i) = uv$

The bijection $\Theta$ is defined as follows:

$\Theta(a) = t$
$\Theta(b) = u$
$\Theta(c) = v$
$\Theta(d) = w$

And, the bijection $\Phi$ is defined as follows:

$\Phi(l) = e$
$\Phi(m) = g$
$\Phi(n) = h$
$\Phi(o) = i$
$\Phi(p) = f$

Notice that the following holds for each edge in *G2*:

$$\begin{aligned}
\Psi_{G3}(\Phi(l)) &= \Psi_{G3}(e) &= \Theta(a)\Theta(b) &= tu \\
\Psi_{G3}(\Phi(m)) &= \Psi_{G3}(g) &= \Theta(b)\Theta(b) &= uu \\
\Psi_{G3}(\Phi(n)) &= \Psi_{G3}(h) &= \Theta(b)\Theta(d) &= uw \\
\Psi_{G3}(\Phi(o)) &= \Psi_{G3}(i) &= \Theta(b)\Theta(c) &= uv \\
\Psi_{G3}(\Phi(p)) &= \Psi_{G3}(f) &= \Theta(c)\Theta(d) &= vw
\end{aligned}$$

Thus, the graphs *G2* and *G3* shown in Figure 6 are isomorphic.

We will use a definition of labeled graphs similar to that of Wilson [Wil79]. A labeled graph is a graph *G* for which each node is assigned a distinguishing name. More precisely, a labeled graph is a tuple $(G, \Gamma)$ where $\Gamma$ is a one-to-one mapping from $V(G)$ onto the set $\{l_1, l_2, l_3, \ldots, l_n\}$ where $n =$

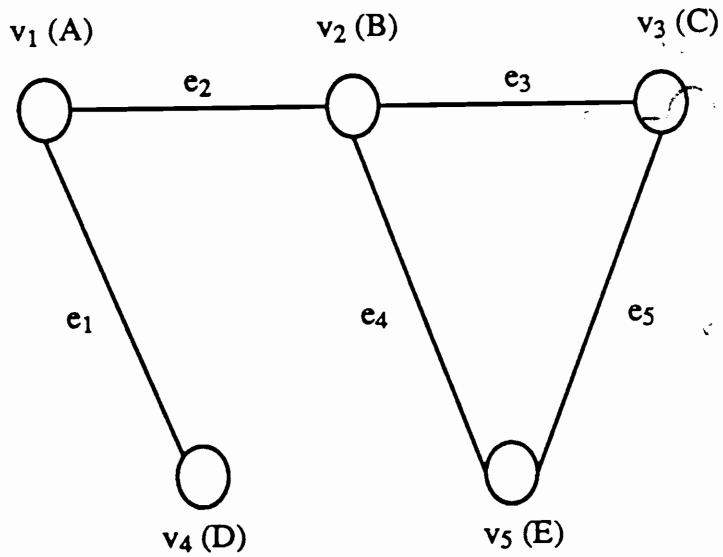



**Figure 7.** A Labeled Graph

$|V(G)|$ and $l_i$ is a label. An example of a labeled graph is given in Figure 7. The name of each vertex is placed in parenthesis beside the label of the node for clarity. For this example, $\Gamma$ is defined as follows:

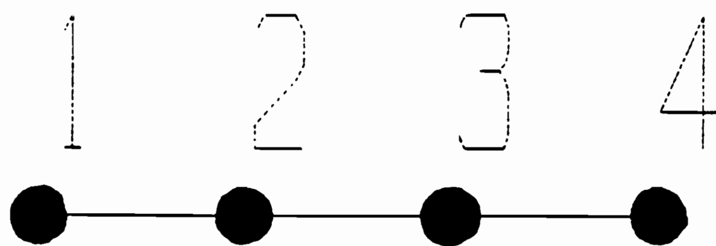
$$\Gamma(v_1) = A$$
$$\Gamma(v_2) = B$$
$$\Gamma(v_3) = C$$
$$\Gamma(v_4) = D$$
$$\Gamma(v_5) = E$$

Vertices may be referred to by their labels rather than separate names when no confusion will result.

Two labeled graphs $G_1$ and $G_2$ are isomorphic, which will be referred to as L-isomorphic for clarity, if there is some isomorphism between $G_1$ and $G_2$ for which the labeling of the vertices is preserved. Figure 8 shows two labeled graphs which are isomorphic but not L-isomorphic.

A graph model for fault-tolerant computing systems has been proposed by Hayes [Hay76]. This model uses a facility graph to describe the system. Each node represents a facility in the system. Each edge represents access between facilities. Facility graphs may be directed or undirected, and they may or may not be labeled.

A facility graph $S$ can be used to define a computing system in which the nodes represent the facilities (including spare facilities) available and the edges represent the access links between facilities. Suppose that the graph in Figure 9 is the facility graph for a computing system $S1$. A facility graph $A$ can be used to define an algorithm associated with the system. The nodes of



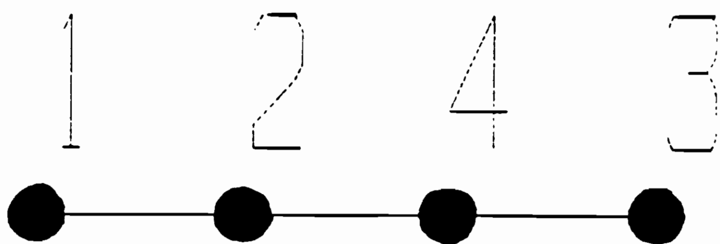




**Figure 8.** Two Graphs which are not L-Isomorphic

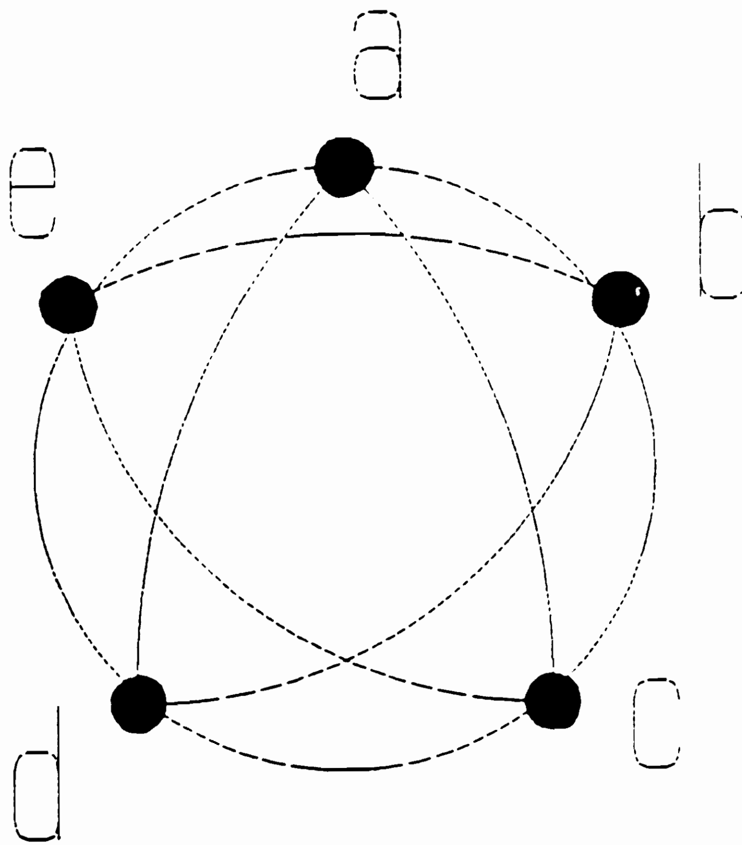



**Figure 9**. Facility graph for a Computing System

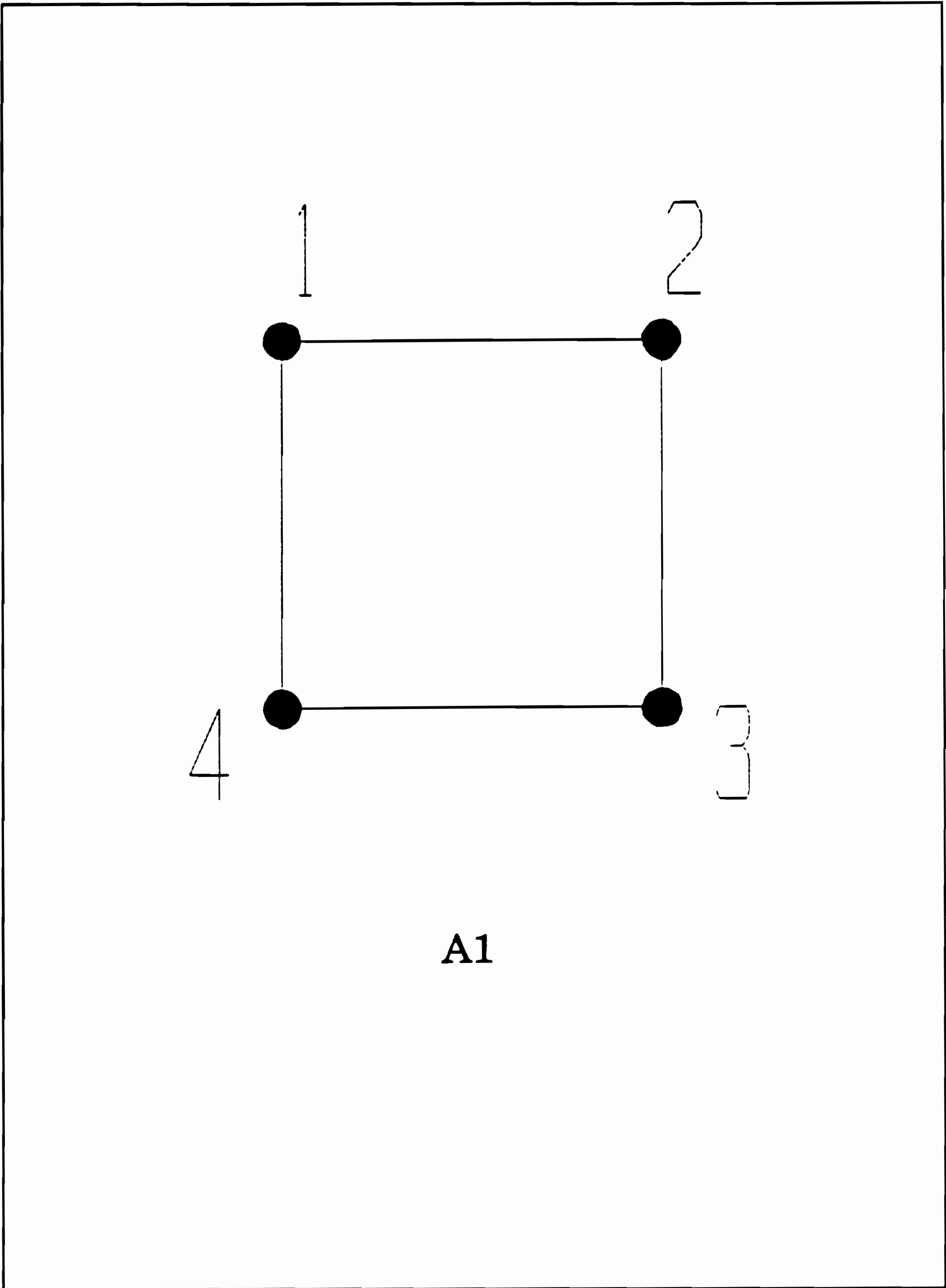



**Figure 10.** Facility Graph for an Algorithm

$A$ are the facilities required by the algorithm, while the edges of $A$ are the access links required between facilities. Suppose that the graph in Figure 10 is the facility graph for an algorithm *A1*. An algorithm can be executed on a system, if there exists a subgraph of $S$ that is isomorphic to $A$. Algorithm *A1* can be executed on *S1* since there is a subgraph of *S1* that is isomorphic to *A1*--for example, b$\rightarrow$1, c$\rightarrow$2, d$\rightarrow$3, e$\rightarrow$4.

A fault in a facility is represented by removing the facility's associated node from its facility graph $S$. All edges incident on that node are also removed. Thus, the graph representing the faulty system is the subgraph induced by the set of vertices representing the non-faulty facilities. Figure 11 shows the facility graph for the system when the facility corresponding to node $a$ is faulty. Faulty communication links may be described by inserting dummy nodes which are removed from the system if the access link is faulty. [Hay76]

### 2.1.2 The Distributed Recovery Strategy Fault Model

The Yanney-Hayes Distributed Recovery Strategy [Yan82] [Yan86] uses facility graphs to describe the system. The Basic Graph, $G_b$, is a labeled $n$-node facility graph. $G_b$ identifies the minimum system required to perform a given set of software tasks (algorithms) where the label of each node corresponds to the task assigned to the computing system's node. Thus, $G_b$ is similar to a facility graph representing an algorithm. The Redundant Graph, $G_r$, is an unlabeled facility graph. $G_r$ identifies the facilities and access links available in the system. Thus, $G_r$ is a facility graph representing the

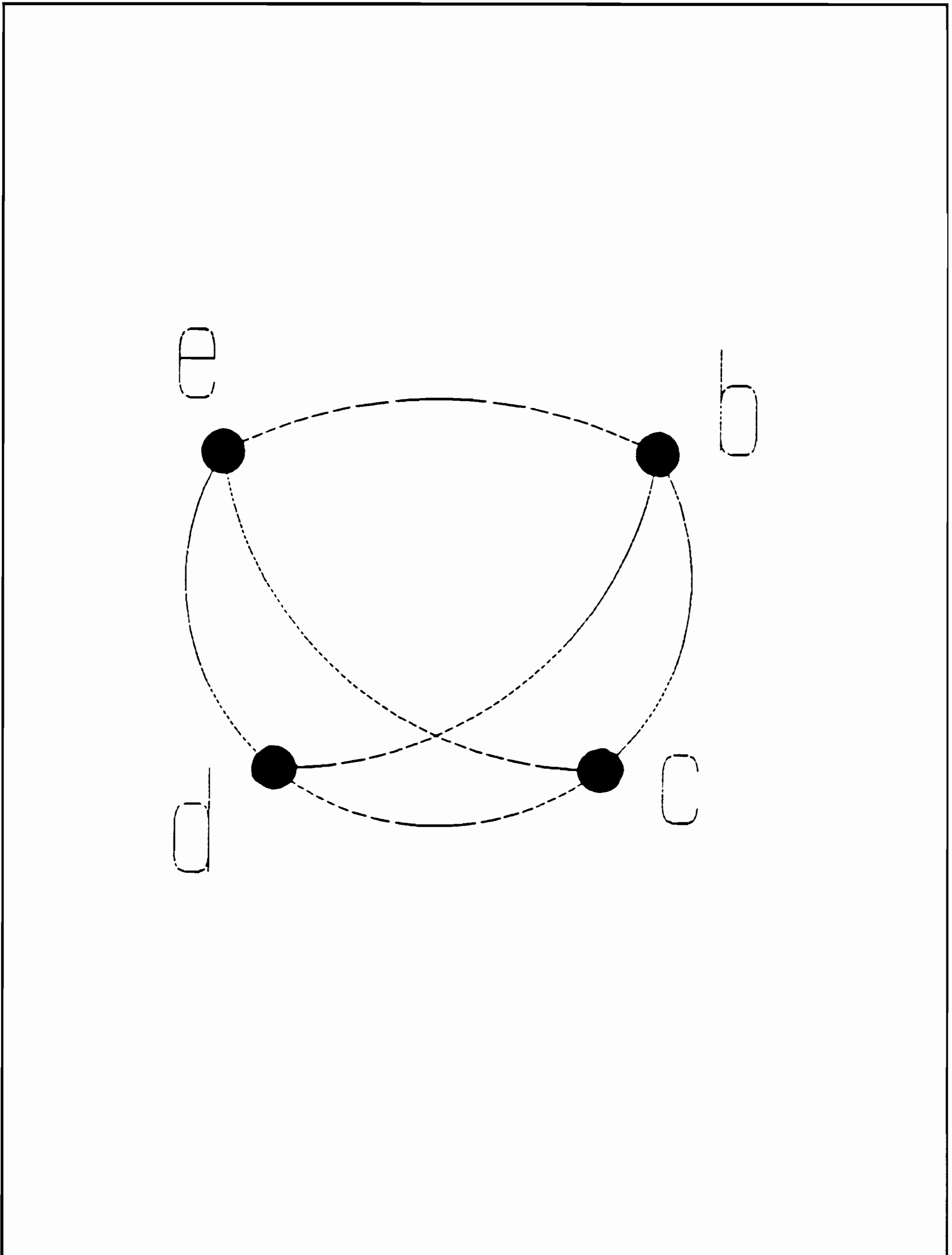



**Figure 11.** Facility Graph for Faulty Computing System with Node a Faulty

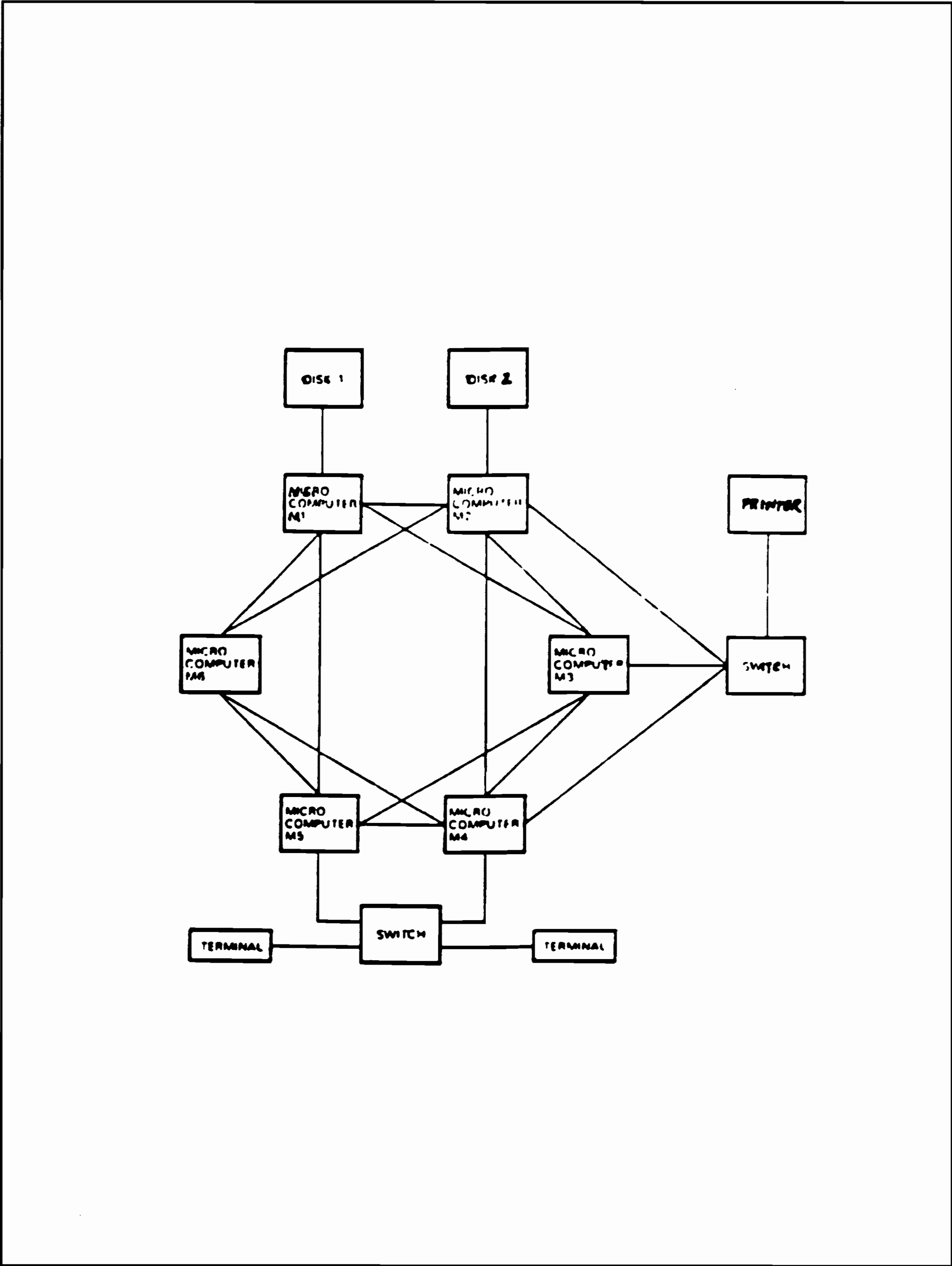



**Figure 12.** 6-Computer Basic Fault-Tolerant System [Yan86]

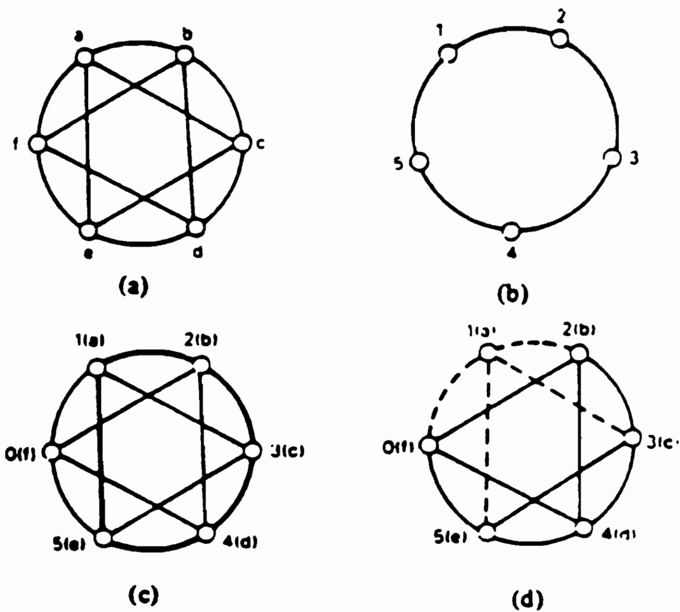



**Figure 13.** Graph models for the BFS [Yan86]

computing system. To indicate that $G_b$ is the underlying basic graph, $G_r$ is also written $G_r[G_b]$. The Configuration $G_c$ is a labeled graph whose related unlabeled graph is isomorphic to $G_r$. Each node is labeled with its associated state. The states that are used as labels of $G_b$ are called the task states. Nodes in the configuration graph can also have the spare state or the faulty state. In our research, task states are also called computation states.

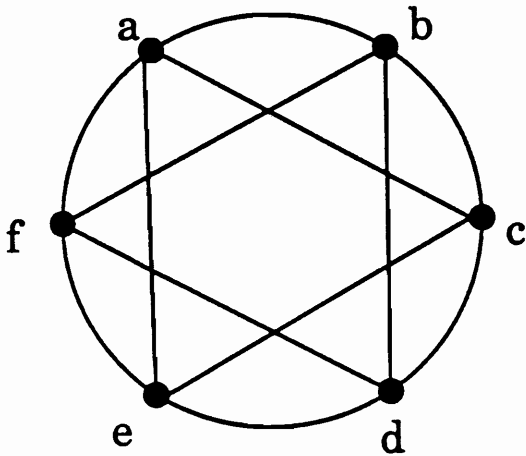
Yanney and Hayes denote task states by the labels 1, 2, 3, ... n, the spare state is denoted by the label 0, and the faulty state is denoted by the label -1. In this research, the rules governing the choice of labels were relaxed to include any meaningful labels for computation states, and the words SPARE and FAULTY for the spare and faulty states, respectively. The state $S(Gr)$ for the $m$-node system described by $G_r$ is the $m$-tuple $S(x_1, x_2, \ldots, x_m) = (S(x_1), S(x_2), \ldots, S(x_m))$ where $S(x_i)$ refers to the state of node $x_i$. For a configuration $G_c$ to be a valid configuration, two conditions must be true. First, it must contain a subgraph which is L-isomorphic to $G_b$. Second, if $G_b$ has $n$ nodes, then $G_c$ must have exactly $n$ nodes which have computation states. [Yan82]

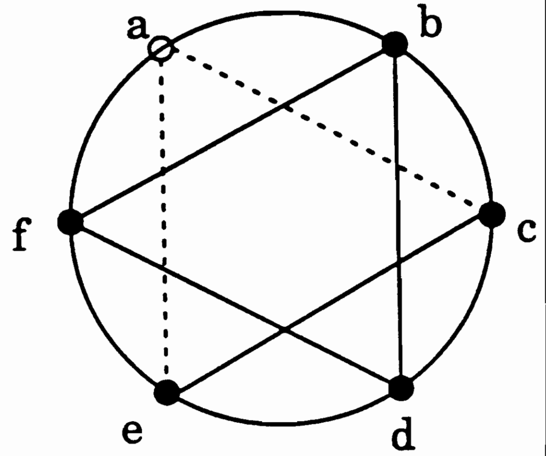
For example, Figure 12 shows the six-computer Basic Fault-Tolerant System (BFS) [Sch84]. The redundant graph for the BFS is shown in Figure 13(a). Each microcomputer of the BFS has a corresponding node in the redundant graph. Each direct link between microcomputers in the BFS has a corresponding edge in the redundant graph. To operate properly, the BFS requires a 5-node cycle. Figure 13(b) shows the basic graph representing the

minimum system configuration for correct operation. Suppose the initial state of the system is *S(a, b, c, d, e, f) = (1, 2, 3, 4, 5, 0)* which is shown in Figure 13(c). This configuration is valid since the configuration graph contains a subgraph which is L-isomorphic to the basic graph. Suppose a fault affects node *a*. Since the new configuration shown in Figure 13(d) does not have a node in state *1*, it does not have a subgraph that is L-isomorphic to the basic graph. Thus, it is not a valid configuration.

The problem of reconfiguring a system to give a valid configuration is related to the graph isomorphism problem. The graph isomorphism problem is the problem of finding an efficient algorithm to determine whether two graphs are isomorphic. An efficient algorithm is an algorithm which is of polynomial complexity. Many methods for determining whether two graphs are isomorphic have been proposed [Rea77]. This problem is known to be in NP, but it is not known to be NP-complete [Cip90].

The reconfiguration problem can be described as follows. Given an initial valid configuration. Suppose a fault occurs which affects the system. Find a subgraph in the redundant graph which is isomorphic to the unlabeled basic graph. Then, label that subgraph to match the labeled basic graph. [Yan82] Consider the BFS system discussed previously. Figure 13 shows the redundant graph, the basic graph, and an initial valid configuration. Suppose a fault affects node *a* as shown in Figure 13. Then, the reconfiguration problem is to find a subgraph in the faulty redundant graph which is



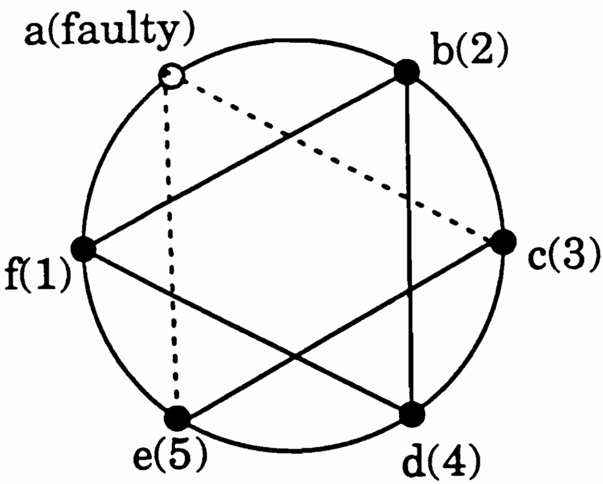




**Figure 14.** Example of the Reconfiguration Problem

isomorphic to the underlying unlabeled graph of $G_b$. The redundant graph corresponding to the faulty system is shown in Figure 14(a). It does contain a subgraph which is isomorphic to the underlying unlabeled graph of $G_b$ as shown in Figure 14(b). So, the subgraph is labeled to match the basic graph. One such labeling is shown in Figure 14(c). The reconfiguration problem which is not confined to a particular reconfiguration algorithm is the problem of determining whether reconfiguration can occur at all.

### 2.1.3 The Distributed Recovery Strategy

The general distributed recovery strategy developed by Yanney and Hayes uses the graph theoretical approach described above to define a procedure by which local actions produce a global reconfiguration of the architecture to handle a fault. As described previously, a fault is represented in $G_r[G_b]$ by removing the corresponding node and all edges incident on that node [Yan86]. The Distributed Recovery Strategy assumes that each node can determine whether or not each of its neighbors is faulty by some means. For convenience, a faulty cell is assigned a state of -1, and a spare cell is assigned a state of 0. An error $E(s_i)$ is present in the system if no node in the configuration has state $s_i$ which corresponds to a node in $G_b$.

In the Distributed Recovery Strategy, cells respond to specific error conditions which are assigned to them. The Distributed Recovery Strategy does not address the problem of how these error response assignments, also called error response sets, are determined. A diagram describing the

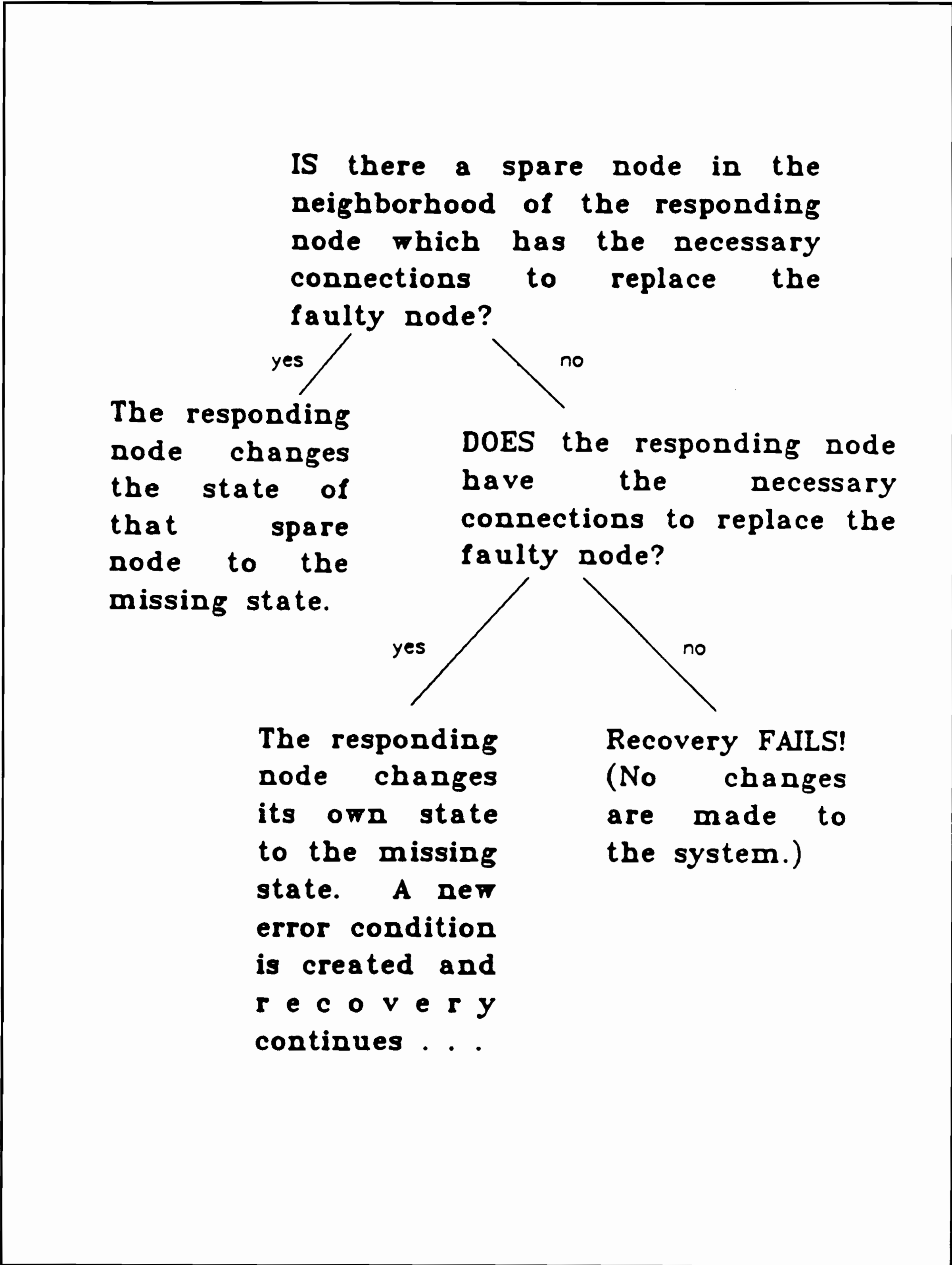



**Figure 15.** The Distributed Recovery Strategy

Distributed Recovery Strategy is given in Figure 15. The previous state of the faulty node is referred to as the missing state since it is no longer present in the system. The node which responds to an error condition will be referred to as the responding node. The Distributed Recovery Strategy is described by the following:

1. If there is a spare node in the neighborhood of the responding node which has the necessary connections to replace the faulty node, then the responding node changes the state of that spare node to the missing state (which was previously the state of the faulty node).

2. If there are no spare nodes in the neighborhood of the responding node which have the necessary connections to replace the faulty node and if the responding node has the necessary connections to replace the faulty node, then the responding node changes its own state to the missing state (which was previously the state of the faulty node). This state change creates a new error condition to which another node could respond the recovery process will continue.

3. If the spares in the neighborhood of the responding node and the responding node itself do not have the necessary connections to assume the missing state (which was previously the state of the faulty node), then the recovery attempt fails and no changes are made to the system.

In summary, the responding node will attempt to find a spare to take over the missing state. If no spare can be found, then the responding node takes over the missing state, and the recovery process continues. If no spare can be found and the responding node cannot take over the missing state, then the recovery attempt fails.

The Distributed Recovery Strategy makes several assumptions. These assumptions are listed in Figure 16. The induced subgraph referred to in Assumption 7 is the subgraph induced by the vertex set consisting of $x_i$ and all

1) Each active node $x_i$ can detect all errors from a specified set $\{E(s_j)\}$ occurring in its neighborhood $N(x_i)$. The detection process is assumed to be complete and not subject to failure.

2) In response to detecting an error, $x_i$ reconfigures the state of $N(x_i)$ according to a specified recovery strategy R. The reconfiguration process is also assumed to be fault free, with $x_i$ automatically maintaining all the state information on $N(x_i)$ needed to execute R.

3) The error $E(s_j)$ corresponds to the removal, due to failure or a recovery step, of the state $s_j$ from the system.

4) The error sets associated with every pair of nodes is disjoint.

5) The failure modes and recovery strategy are constrained so that no more than one error at a time can be present in the system.

6) When reconfiguring its neighborhood in response to an error $E(s_j)$, a local supervisor $x_i$ assigns state $s_j$ to a spare node, if possible; otherwise it changes its own state to $s_j$.

7) The neighborhood $N(x_i)$ of each node $x_i$ is restricted to the induced subgraph comprising $x_i$ and all adjacent nodes.

**Figure 16.** Assumptions Underlying the Distributed Recovery Model [Yan86]

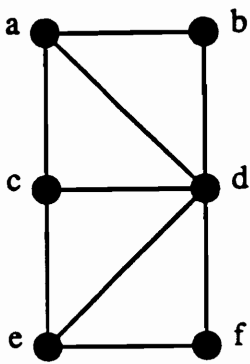
nodes adjacent to it. Another assumption should be added: the union of all the error response sets is the set of all possible errors (which is all of the states except the fault state and the spare state).

The distributed recovery strategy is illustrated in the following example. Figure 17(a) shows the redundant graph for a computer system. Figure 17(b) shows the basic graph representing the tasks to be implemented using that system. Figure 17(c) shows the initial valid configuration of the basic graph embedded in the redundant graph. The error response assignments are chosen as follows:

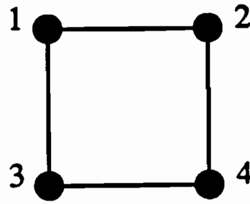
| State | Errors Response Assignments |
|---|---|
| 1 | none |
| 2 | 1 |
| 3 | 4 |
| 4 | 2 |

If node *b* becomes faulty, the error condition *E(2)* is generated. This configuration is shown in Figure 17(d). Since state 4 is assigned to respond to error *E(2)*, node *d* which has state 4 is the local supervisor. Since no node in the neighborhood of node *d* has the required adjacencies to take over state 2 and since node *d* does have all of the required adjacencies, node *d* takes over state 2 creating error *E(4)*. This invalid configuration is shown in Figure 17(e).

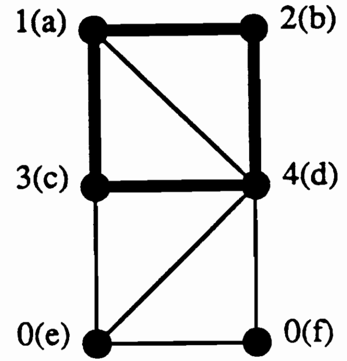
Since a valid configuration does not exist, the reconfiguration process continues. The state assigned to respond to error *E(4)* is state 3. So, node *c* looks for a spare node in its neighborhood with the required adjacencies to take



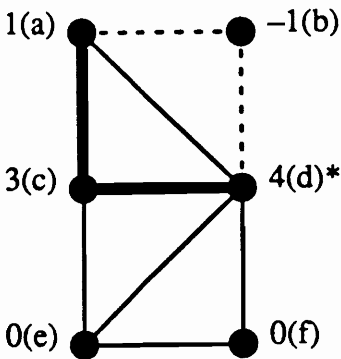

(a)



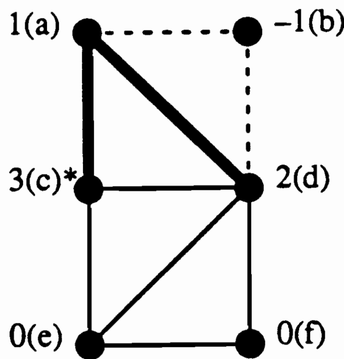

(b)



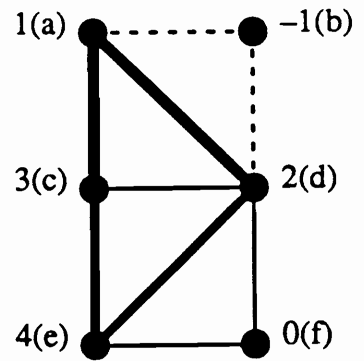

(c) Initial Configuration




(d) E(2)




(d) E(4)




(e) Final Configuration

**Figure 17.** An Example showing the Distributed Recovery Strategy

over state 4. State 4 must be adjacent to states 2 and 3. Node *e* meets the adjacency requirements and is assigned state 4. Thus, a spare is found and reconfiguration is complete. The final configuration is shown Figure 17(f).

## 2.2 Application of the Distributed Recovery Strategy to an Array Architecture

The application of the Yanney-Hayes Distributed Recovery Strategy to the White-Gray array is described in [Gra88][DeB90]. The White-Gray architecture was chosen as the underlying fault-tolerant architecture because an Architecture Design and Assessment System (ADAS) hardware model of the architecture and a reconfiguration algorithm had previously been developed. This choice decreased the time required for model development and also provided an algorithm to which the Distributed Recovery Strategy-based algorithm could be compared. Details of the White-Gray Reconfigurable Fault Tolerant Cellular Array are given in [Whi88]. The basic graph, $G_b$, is a 3 X 3 systolic array with Von Neumann connections (North, South, East, West). $G_b$ is shown in Figure 18. The redundant graph corresponding to the White-Gray array is shown in Figure 19. The size of the redundant graph was chosen somewhat arbitrarily to be a 7 X 7 array. This size is large enough to allow the investigation of all of the situations that need to be considered. The model could easily be extended to different dimensions. This section describes the ADAS model, summarizes simulation results, presents related proofs, and identifies limitations of the Distributed Recovery Strategy.

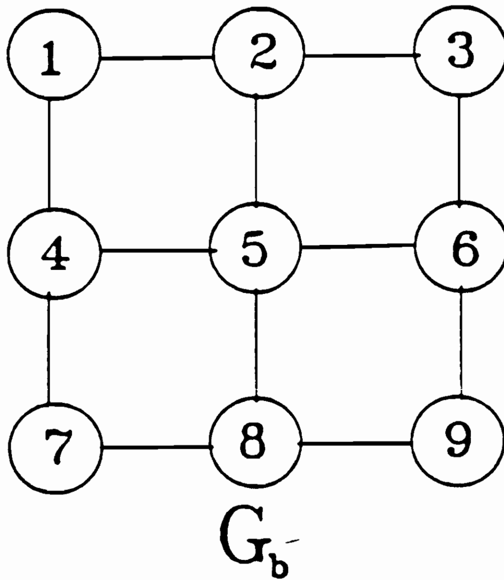



**Figure 18.** The Basic Graph

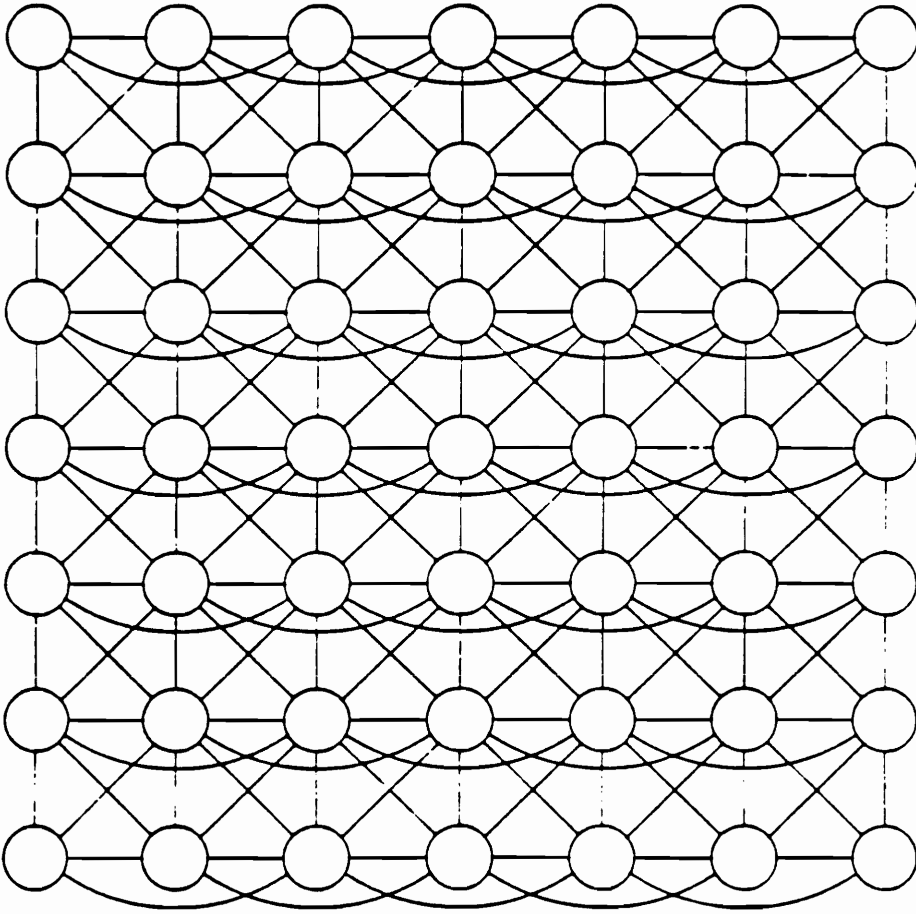



**Figure 19**. The Redundant Graph

### 2.2.1 Description of the ADAS Model

The Architecture Design and Assessment System (ADAS) was used to model the Distributed Recovery Strategy-Based algorithm as applied to the White-Gray array. In this section, the capabilities of ADAS are briefly described and then the ADAS model of the algorithm is discussed.

The Architecture Design and Assessment System (ADAS) [Fra85] developed by Research Triangle Institute is an integrated set of tools which can be used to evaluate and assess many aspects of a computer system. A color graphics interface is used to indicate visually the behavior of the system. The features of ADAS used in this research will be described in the following paragraphs.

Using the graph editor feature of ADAS, a graph is developed to describe the behavior of the system. A graph is a modified Petri net. Each input to a node has a threshold value and a consume value associated with it. Similarly, each output has a produce value associated with it. When the number of tokens present on each input reaches or exceeds the threshold associated with the input, then the node is ready to fire. A firing delay is associated with each node. The firing delay is the length of time between the moment that tokens are present on all inputs and the node fires. When a node fires, tokens are consumed at each input line, and tokens are produced at each output line. The number of tokens consumed are specified by the consume value, and the number of tokens produced are specified by the produce value. The graph is

simulated at a high level to determine whether architecture performance goals are met. In addition, each node in the graph can be associated with a hardware component. Thus, the utilization of hardware components can be determined.

A C simulation tool works with the Petri net simulator to perform functional modeling. This feature can be used to associate C routines with nodes in the system graph. In this research, the C simulation feature was used for functional modeling.

ADAS was used to model the Distributed Recovery-based algorithm for the White-Gray array. The graph representing the White-Gray array is described in detail in [Gra88]. This graph is shown in Figure 20. It models the fault-tolerant array represented by $G_r[G_b]$. Each *CELL__* node corresponds to a node in the redundant graph. In addition, several nodes are provided for control. For this implementation, these control nodes are only used to start the simulation. All other timing control is based on the firing delays of the *CELL__* nodes and the requirement that data be present on all input lines for a node to fire. Each *split__* node is used only for token distribution. The firing delay of each *split__* node is zero. The firing delay for each *CELL__* node is set to one to represent one arbitrary period.

Each *CELL__* node has eleven input ports (0 through 10) and ten output ports (0 through 9). For each *CELL__* node, input port 10 is connected to a *split__* node and is used only for controlling the start of the simulation. All of

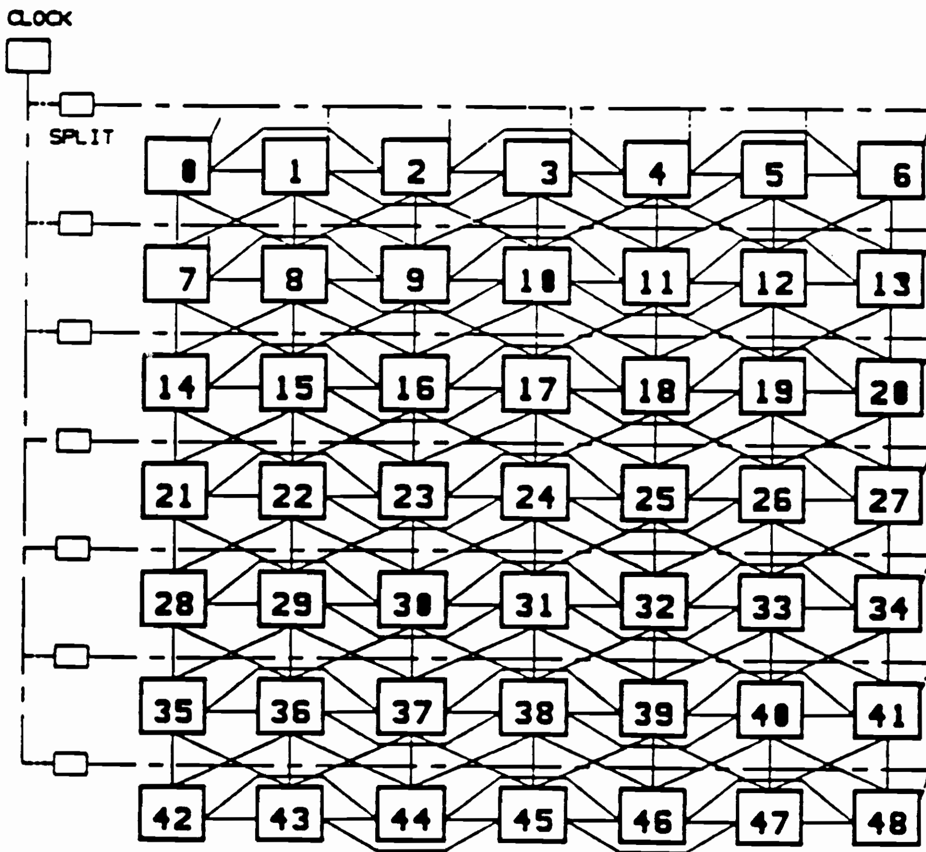



**Figure 20.** The ADAS Hardware Graph corresponding to the Redundant Graph [GRA88]

| Port Number | Neighbor to which Port Connects |
|---|---|
| 0 | Far West |
| 1 | West |
| 2 | Northwest |
| 3 | North |
| 4 | Northeast |
| 5 | Far East |
| 6 | East |
| 7 | Southeast |
| 8 | West |
| 9 | Southwest |

**Figure 21.** The ADAS Hardware Graph Port Assignments [Gra88]

the rest of the input ports are connected to neighboring *CELL__* nodes, and all of the output ports are connected to neighboring *CELL__* nodes. The connections between nodes are summarized in Figure 21.

The C simulation capability of ADAS was used to simulate the reconfiguration algorithm. C routines were written to describe the behavior of each type of node. Very simple routines were written for the *clock__* and *split__* nodes. These routines simply produce the necessary tokens to begin the simulation. A more complicated C routine was developed for the *Cell__* nodes. This routine was written in a general manner to provide as much flexibility as possible. These routines can be modified easily to use different error response sets by changing the error response look-up table. These routines can also be modified slightly and used to implement the Yanney-Hayes strategy for different architectures. The basic graph is determined by another look-up table which allows different basic graphs to be chosen with very little change to the C routines. The routine associated with the *Cell__* nodes follows the Distributed Recovery Strategy as given in the Distributed Recovery Strategy description above. The actual C routines associated with the nodes in the ADAS hardware graph are given in Appendix A.

### 2.2.2 Simulation Results

The Yanney-Hayes algorithm does not specify how the error response set should be chosen. So, several error response sets were tried. The error response chosen seemed to work fairly well; however, it may not be the best

| CELL STATE | ERROR RESPONSE ASSIGNMENTS |
| --- | --- |
| 1 | --- |
| 2 | E(1) |
| 3 | E(2) |
| 4 | --- |
| 5 | E(4) |
| 6 | E(3),E(5) |
| 7 | --- |
| 8 | E(7),E(9) |
| 9 | E(6),E(8) |

**Figure 22.** Error Response Assignment for Yanney-Hayes Algorithm [Gra88]

choice. The error response assignment chosen is shown in Figure 22. A method for choosing the error response set to be used is described in a following section.

ADAS simulations were run for all single faults and all possible double sequential faults to determine whether reconfiguration takes place correctly for these simple fault occurrences. A double sequential fault is defined as "two single faults separated in time so reconfiguration in response to the first fault has completed before the second fault occurs" [Gra88]. Since the Distributed Recovery Strategy assumes that only one fault will be present in the system at a given time, multiple faults (coincident or near-coincident) cannot be handled. The final configuration after the double sequential fault E(9)E(6) is shown in Figure 23. Reconfigurations for all single and double sequential faults are given in [Gra88].

The simulation results for the Distributed Recovery Strategy-based array application will be compared to the results for the local mode of the White-Gray algorithm to provide a perspective. The White-Gray algorithm also includes a global algorithm which allows recovery for most cases in which the local mode fails. The global algorithm is more complex and requires more time than the local algorithm. In addition, this global algorithm could be used with any other local algorithm including the Distributed Recovery Strategy-based array algorithm. Thus, the comparison used is appropriate.

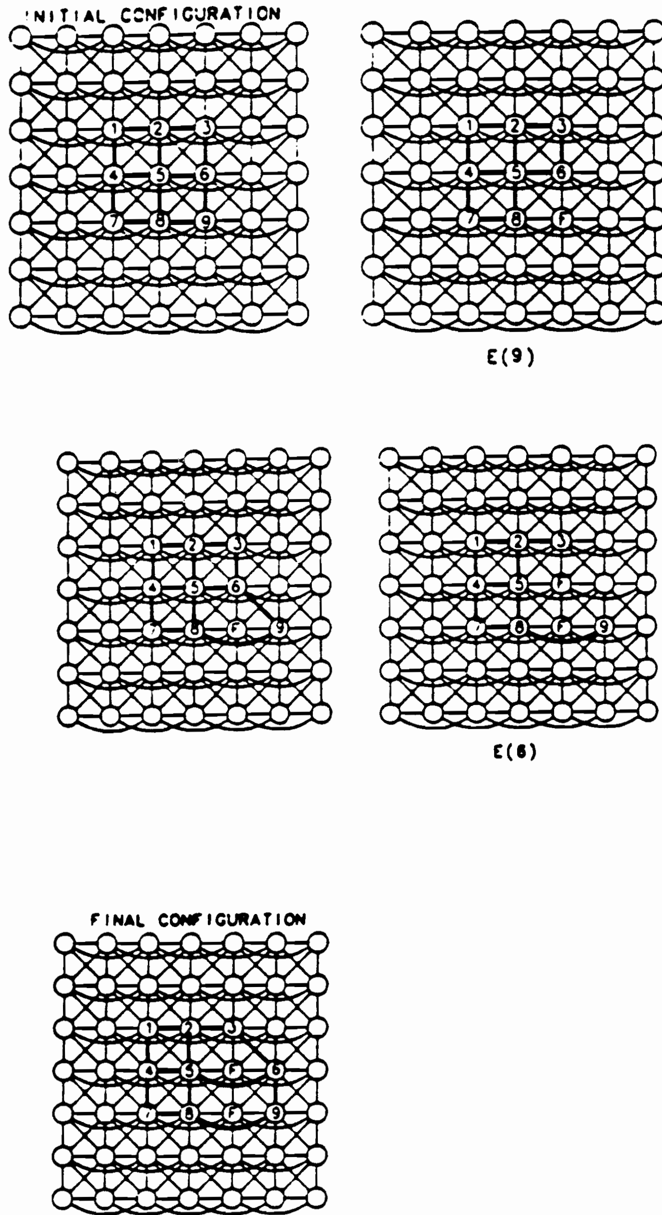



**Figure 23.** Reconfiguration for a Double Sequential Fault in Cells 9 and 6 [Gra88]

The coverage of the Distributed Recovery Strategy-based array algorithm and the White-Gray algorithms are summarized in Figure 24. The Distributed Recovery Strategy-based array algorithm covers all single faults, as does the White-Gray Local algorithm. This coverage of single faults by the Distributed Recovery Strategy-based array algorithm was expected since the error response set was chosen so the recovery strategy would not fail for any single fault.

The coverage of the Distributed Recovery Strategy-based array algorithm and the White-Gray Local algorithm for all cases of two faults present in the system are also given in Figure 24. Coincident faults are defined to be faults which occur within a clock cycle of each other so they appear simultaneous. Near coincident faults are defined to be faults for which reconfiguration in response to one is taking place when the other faults occur. The White-Gray Local algorithm covers 75% of double coincident faults and 75% of double near coincident faults. The Distributed Recovery Strategy-based array algorithm is not designed to cover any coincident or near-coincident faults because of the critical assumption of the Distributed Recovery Strategy that only one fault be present in the system at a given time. If more than one fault were present in the system at one time, the results would not be defined. It is likely that some coincident and near-coincident faults would be covered in an actual implementation, but it would be a side-effect of the algorithm and not due to the algorithm itself. The extension of the Distributed Recovery Strategy-based algorithm to handle multiple faults is described in a later chapter.

| Fault Class | YH | WG Local | WG Global |
| --- | --- | --- | --- |
| Single Faults | 100% | 100% | - |
| Coincident Double Faults | 0% | 75% | 25% |
| Near Coincident Double Faults | 0% | 75% | 25% |
| Sequential Double Faults | 83% | 67% | 33% |

**Figure 24.** Summary of Coverage from ADAS Simulations [Gra88]

The Distributed Recovery Strategy-based algorithm will cover some double sequential faults and will fail for others. The White-Gray Local reconfiguration strategy will cover double sequential faults in which both faults are not in the same row of the array, 67% of double sequential faults. The Distributed Recovery Strategy-based array algorithm covers 83% of double sequential faults.

The behavior of the Distributed Recovery Strategy-based array algorithm for the double sequential fault E(9)E(9) is particularly troubling. This double sequential fault results in the reconfiguration algorithm entering an infinite loop. This type of failure must be identified and handled through the use of a timer or counter. The choice of error response set can also reduce or eliminate this type of failure. More information about how the choice of error response set affects the occurrence of this type of failure is given in a later section.

Whenever the White-Gray Local algorithm cannot reconfigure correctly, the White-Gray Global algorithm is invoked. For all cases of two faults in the system for which the White-Gray Local algorithm fails, the White-Gray Global algorithm can reconfigure correctly. Similarly, a global algorithm could be used with the Distributed Recovery Strategy-based array algorithm to improve overall coverage for the system.

The recovery times for the different local algorithms were also compared. A compilation of recovery times from the ADAS simulations is given in

| Error Condition | Faulty Cell | 2-Phase YH Steps | 2-Phase WG Steps |
|---|---|---|---|
| E(1) | 16 | 1 | 3 |
| E(2) | 17 | 2 | 2 |
| E(3) | 18 | 1 | 1 |
| E(4) | 23 | 1 | 3 |
| E(5) | 24 | 2 | 2 |
| E(6) | 25 | 1 | 1 |
| E(7) | 30 | 1 | 3 |
| E(8) | 31 | 2 | 2 |
| E(9) | 32 | 1 | 1 |

**Figure 25.** Time Analysis of Single Faults from ADAS Simulations [Gra88]

| | | Coincident or Near-Coincident Faults | | Double Sequential Faults | |
|---|---|---|---|---|---|
| Fault Sequence | Faulty Cells (YH) | 2-Phase YH Steps | 2-Phase WG Steps | 2-Phase YH Steps | 2-Phase WG Step |
| E(1)E(1) | 16,15 | ----- | ----- | 4 | Fails |
| E(1)E(2) | 16,17 | Fails | Fails | Fails | Fails |
| E(1)E(3) | 16,18 | Fails | Fails | 2 | Fails |
| E(1)E(4) | 16,23 | Fails | 3 | 2 | 6 |
| E(1)E(5) | 16,24 | Fails | 3 | 3 | 5 |
| E(1)E(6) | 16,24 | Fails | 3 | 2 | 4 |
| E(1)E(7) | 16,30 | Fails | 3 | 2 | 6 |
| E(1)E(8) | 16,31 | Fails | 3 | 3 | 5 |
| E(1)E(9) | 16,32 | Fails | 3 | 2 | 4 |
| E(2)E(1) | 17,16 | Fails | Fails | Fails | Fails |
| E(2)E(2) | 17,18 | ----- | ----- | Fails | Fails |
| E(2)E(3) | 17,19 | Fails | Fails | 3 | Fails |
| E(2)E(4) | 17,23 | Fails | 3 | 3 | 5 |
| E(2)E(5) | 17,24 | Fails | 2 | 4 | 4 |
| E(2)E(6) | 17,25 | Fails | 2 | 3 | 3 |
| E(2)E(7) | 17,30 | Fails | 3 | 3 | 5 |
| E(2)E(8) | 17,31 | Fails | 2 | 4 | 4 |
| E(2)E(9) | 17,32 | Fails | 2 | 3 | 3 |
| E(3)E(1) | 18,16 | Fails | Fails | 2 | Fails |
| E(3)E(2) | 18,17 | Fails | Fails | Fails | Fails |
| E(3)E(3) | 18,19 | ----- | ----- | 3 | Fails |
| E(3)E(4) | 18,23 | Fails | 3 | 2 | 3 |
| E(3)E(5) | 18,24 | Fails | 2 | 3 | 3 |
| E(3)E(6) | 18,25 | Fails | 1 | 2 | 2 |
| E(3)E(7) | 18,30 | Fails | 3 | 2 | 4 |
| E(3)E(8) | 18,31 | Fails | 2 | 3 | 3 |
| E(3)E(9) | 18,32 | Fails | 1 | 2 | 2 |

**Figure 26.** Time Analysis of Double Sequential Faults from ADAS Simulations (Part 1) [Gra88]

| Fault Sequence | Faulty Cells (YH) | 2-Phase YH Steps | Coincident or Near-Coincident Faults<br>2-Phase WG Steps | 2-Phase YH Steps | Double Sequential Faults<br>2-Phase WG Step |
|---|---|---|---|---|---|
| E(4)E(1) | 23,16 | Fails | 3 | 2 | 6 |
| E(4)E(2) | 23,17 | Fails | 3 | 3 | 5 |
| E(4)E(3) | 23,18 | Fails | 3 | 2 | 4 |
| E(4)E(4) | 23,22 | ----- | ----- | 4 | Fails |
| E(4)E(5) | 23,24 | Fails | Fails | Fails | Fails |
| E(4)E(6) | 23,25 | Fails | Fails | 2 | Fails |
| E(4)E(7) | 23,30 | Fails | 3 | 2 | 6 |
| E(4)E(8) | 23,31 | Fails | 3 | 3 | 5 |
| E(4)E(9) | 23,32 | Fails | 3 | 2 | 4 |
| | | | | | |
| E(5)E(1) | 24,16 | Fails | 3 | 3 | 5 |
| E(5)E(2) | 24,17 | Fails | 2 | 4 | 4 |
| E(5)E(3) | 24,18 | Fails | 2 | 3 | 3 |
| E(5)E(4) | 24,23 | Fails | Fails | Fails | Fails |
| E(5)E(5) | 24,25 | ----- | ----- | Fails | Fails |
| E(5)E(6) | 24,26 | Fails | Fails | Fails | Fails |
| E(5)E(7) | 24,30 | Fails | 3 | 3 | 5 |
| E(5)E(8) | 24,31 | Fails | 2 | 4 | 4 |
| E(5)E(9) | 24,32 | Fails | 2 | 3 | 3 |
| | | | | | |
| E(6)E(1) | 25,16 | Fails | 3 | 2 | 4 |
| E(6)E(2) | 25,17 | Fails | 2 | 3 | 3 |
| E(6)E(3) | 25,18 | Fails | 1 | 2 | 2 |
| E(6)E(4) | 25,23 | Fails | Fails | 2 | Fails |
| E(6)E(5) | 25,24 | Fails | Fails | Fails | Fails |
| E(6)E(6) | 25,26 | ----- | ----- | Fails | Fails |
| E(6)E(7) | 25,30 | Fails | 3 | 2 | 4 |
| E(6)E(8) | 25,31 | Fails | 2 | 3 | 3 |
| E(6)E(9) | 25,32 | Fails | 1 | 2 | 2 |

**Figure 27.** Time Analysis of Double Sequential Faults from ADAS Simulations (Part 2) [Gra88]

| Fault Sequence | Faulty Cells (YH) | Coincident or Near-Coincident Faults 2-Phase YH Steps | Coincident or Near-Coincident Faults 2-Phase WG Steps | Double Sequential Faults 2-Phase YH Steps | Double Sequential Faults 2-Phase WG Step |
|---|---|---|---|---|---|
| E(7)E(1) | 30,16 | Fails | 3 | 2 | 6 |
| E(7)E(2) | 30,17 | Fails | 3 | 3 | 5 |
| E(7)E(3) | 30,18 | Fails | 3 | 2 | 4 |
| E(7)E(4) | 30,23 | Fails | 3 | 2 | 6 |
| E(7)E(5) | 30,24 | Fails | 3 | 3 | 5 |
| E(7)E(6) | 30,25 | Fails | 3 | 2 | 4 |
| E(7)E(7) | 30,29 | ----- | ----- | 4 | Fails |
| E(7)E(8) | 30,31 | Fails | Fails | Fails | Fails |
| E(7)E(9) | 30,32 | Fails | Fails | 2 | Fails |
| E(8)E(1) | 31,16 | Fails | 3 | 3 | 5 |
| E(8)E(2) | 31,17 | Fails | 2 | 4 | 4 |
| E(8)E(3) | 31,18 | Fails | 2 | 3 | 3 |
| E(8)E(4) | 31,23 | Fails | 3 | 3 | 5 |
| E(8)E(5) | 31,24 | Fails | 2 | 4 | 4 |
| E(8)E(6) | 31,25 | Fails | 2 | 4 | 3 |
| E(8)E(7) | 31,30 | Fails | Fails | Fails | Fails |
| E(8)E(8) | 31,32 | ----- | ----- | Fails | Fails |
| E(8)E(9) | 31,26 | Fails | Fails | 3 | Fails |
| E(9)E(1) | 32,16 | Fails | 3 | 2 | 4 |
| E(9)E(2) | 32,17 | Fails | 2 | 3 | 3 |
| E(9)E(3) | 32,18 | Fails | 1 | 2 | 2 |
| E(9)E(4) | 32,23 | Fails | 3 | 2 | 4 |
| E(9)E(5) | 32,24 | Fails | 2 | 3 | 3 |
| E(9)E(6) | 32,25 | Fails | 1 | 2 | 2 |
| E(9)E(7) | 32,30 | Fails | Fails | 2 | Fails |
| E(9)E(8) | 32,31 | Fails | Fails | Fails | Fails |
| E(9)E(9) | 32,33 | ----- | ----- | Loops | Fails |

**Figure 28.** Time Analysis of Double Sequential Faults from ADAS Simulation (Part 3) [Gra88]

Figure 25 for single faults. Similarly, recovery times for double sequential faults are given in Figure 26, Figure 27, and Figure 28.

The Distributed Recovery Strategy-based array algorithm resulted in reconfiguration processes that take fewer time steps than the White-Gray Local algorithm for 33% of the single faults. The Distributed Recovery Strategy-based array algorithm never took more time steps than the White-Gray Local algorithm for single faults. The Yanney-Hayes algorithm performed better in terms of time steps than the White-Gray algorithm for about 52% of double sequential faults. The White-Gray Local algorithm only took fewer times steps than the Distributed Recovery Strategy-based array algorithm for 1 double sequential fault (about 1%).

For both algorithms, a time unit represents a two-phase step. However, the meaning of the two-phase step differs for each of the algorithms. For the Distributed Recovery Strategy-based array algorithm, the first phase is an exchange of state information. The second phase requires the execution of a complex decision algorithm. This decision algorithm uses stored data which varies with time. The first phase for the White-Gray algorithm is an exchange of state information (similar to that for the Distributed Recovery Strategy-based array algorithm). The second phase consists of decoding this information (10 bits) and conditionally setting control signals. For the White-Gray algorithm, the second phase is quite simple and requires little time. Consequently, the Distributed Recovery Strategy-based array algorithm will

require longer to execute than the White-Gray algorithm in all cases. By addressing the problem of the physical implementation of the Yanney-Hayes algorithm, this recovery time difference could be reduced.

### 2.2.3 Related Proofs

This section consists of several proofs which relate to the Distributed Recovery Strategy-based array algorithm for the White-Gray array with error response set as described previously (the Array Reconfiguration Algorithm). For convenience, state names are defined using Euclidean coordinates. Although this state naming convention is a deviation from that of the Distributed Recovery Strategy, it does not materially affect the proofs since the transformation given in Figure 29 can be used to uniquely transform each state name given in Euclidean coordinate notation to a state name which follows the convention of the Distributed Recovery Strategy. The error response set chosen is described in Figure 30. The notation X(i,j) is used to represent the cell which is located in the ith row and the jth column of the array. To simplify the language of the proofs, the error response set will be defined using a rule format which is equivalent to the error response set stated in Figure 30. This rule format is given in Figure 31.

For the following proofs, the neighborhood of a cell will be described using Euclidean coordinate notation. The row/column coordinates for the neighbors of a cell are found by adding the neighborhood indices to the cell's coordinates. For example, for the neighborhood {(0,1), (1,0)}, the cell X(2,3) has

| Euclidean Coordinate | Distributed Recovery Strategy |
|---|---|
| S(0,0) | 1 |
| S(0,1) | 2 |
| S(0,2) | 3 |
| S(1,0) | 4 |
| S(1,1) | 5 |
| S(1,2) | 6 |
| S(2,0) | 7 |
| S(2,1) | 8 |
| S(2,2) | 9 |

**Figure 29.** State Name Transformation

| CELL STATE | ERROR RESPONSE ASSIGNMENTS |
| --- | --- |
| $S_{0,0}$ | --- |
| $S_{0,1}$ | $E(S_{0,0})$ |
| $S_{0,2}$ | $E(S_{0,1})$ |
| $S_{1,0}$ | --- |
| $S_{1,1}$ | $E(S_{1,0})$ |
| $S_{1,2}$ | $E(S_{1,1})$, $E(S_{0,2})$ |
| $S_{2,0}$ | --- |
| $S_{2,1}$ | $E(S_{2,0})$, $E(S_{2,2})$ |
| $S_{2,2}$ | $E(S_{2,1})$, $E(S_{1,2})$ |

**Figure 30.** Error Response Set

1) For $i = 0,1,2$ and $j = 0,1$, the node in state $S(i,j+1)$ responds to $E(i,j)$.

2) For $i = 0,1$, and $j = 2$, the node in state $S(i+1,j)$ responds to $E(i,j)$.

3) For $i = 2$ and $j = 2$, the node in state $S(i,j-1)$ responds to $E(i,j)$.

**Figure 31.** Error Response Set Rules

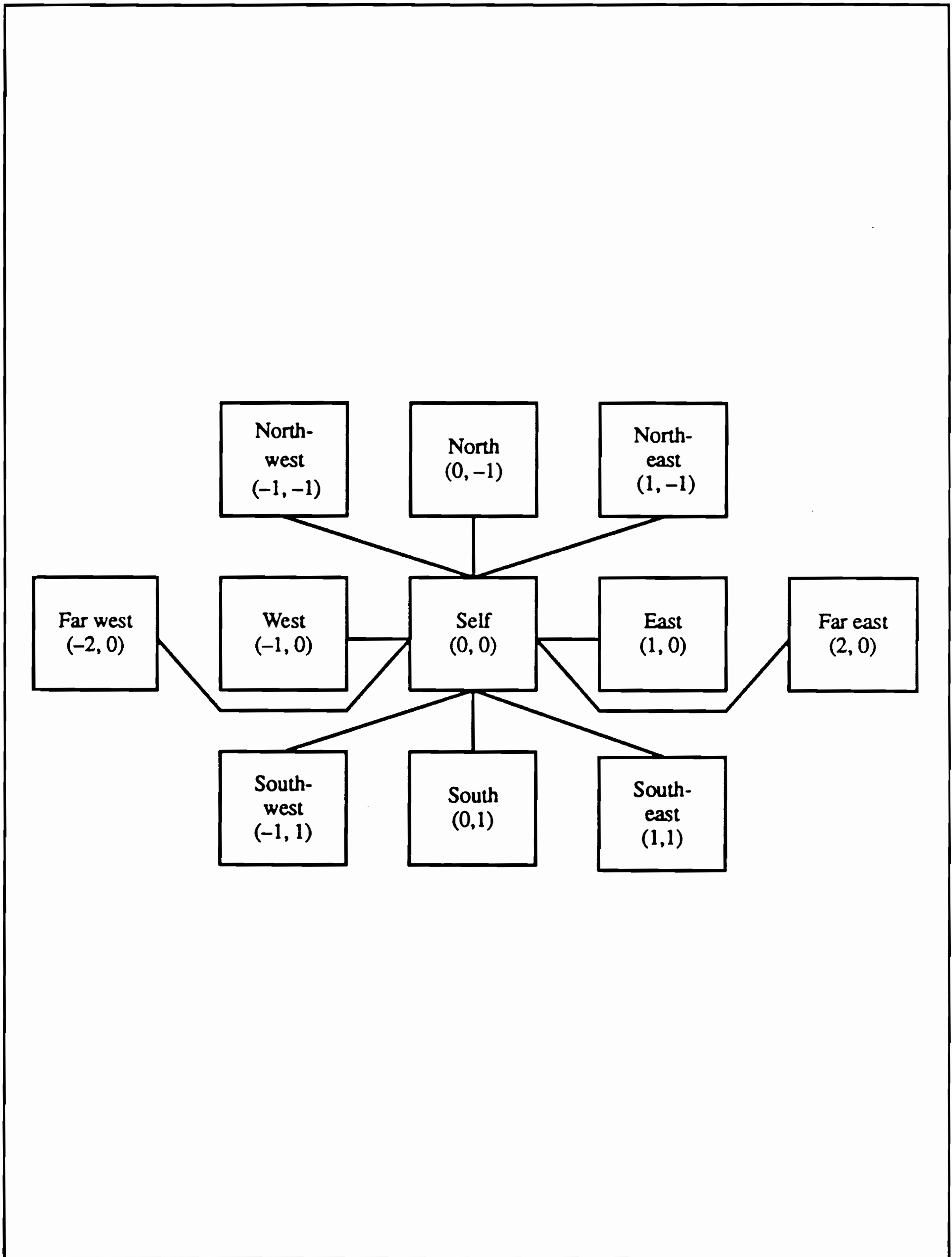



**Figure 32.** The Neighborhood of the Redundant Graph

cells X(2,4) and X(3,3) as neighbors. The cell itself may be included in the neighborhood if desired. Each cell in the array (except cells affected by boundary conditions), has 11 neighbors including itself: northwest, north, northeast, far west, west, self, east, far east, southwest, south, southeast. This neighborhood is shown in Figure 32. Using the Euclidean coordinate notation, the neighborhood is {(-1,-1), (-1,0), (-1,1), (0,-2), (0,-1), (0,0), (0,1), (0,2), (1,-1), (1,0), (1,1)}. For clarity, the notation S(i,j) will be used to denote the state with name (i,j). The cell X(i,j) may or may not have state S(i,j) at any given time. The notation E(i,j) denotes the error condition related to the state S(i,j) being absent from the system.

In this section, two lemmas are given which involve the connections of the neighbors of a cell to the cell's other neighbors. Then, the reconfiguration algorithm is considered for a 3 X 3 basic graph for various cases: the west column, the middle column, and the east column except the south-most cell, the south-most cell in the east column, and the spare elements. Finally, Theorem 2 considers reconfigurations for an N X M basic graph using a similar division into cases.

<u>Lemma 1</u>. Given arbitrarily large graph $G_r$ with neighborhood {(-1,-1), (-1,0), (-1,1), (0,-2), (0,-1), (0,0), (0,1), (0,2), (1,-1), (1,0), (1,1)}. Given another graph $G_b$ with Von Neumann neighborhood {(0,0), (-1,0), (0,-1), (1,0), (0,1)}. Then, node X(i,j+1), the east neighbor of node X(i,j), and node X(i,j-1), the west

neighbor of node X(i,j), are connected to each node in the Von Neumann neighborhood of node X(i,j).

**Proof.**

**Case 1. East Neighbor (node X(i,j+1))**

Node X(i,j+1) is connected to nodes X(i-1,j), X(i-1,j+1), X(i-1,j+2), X(i,j-1), X(i,j), X(i,j+1), X(i,j+2), X(i,j+3), X(i+1,j), X(i+1,j+1) and X(i+1,j+2). Nodes X(i,j), X(i-1,j), X(i,j-1), X(i+1,j) and X(i,j+1) are the nodes in the Von Neumann neighborhood of node X(i,j). Therefore, the east neighbor of node X(i,j) is connected to every node in the Von Neumann neighborhood of X(i,j).

**Case 2. West Neighbor (node X(i,j-1))**

By symmetry, this case is also true.

Lemma 2. Given arbitrarily large graph $G_r$ with neighborhood {(-1,-1), (-1,0), (-1,1), (0,-2), (0,-1), (0,0), (0,1), (0,2), (1,-1), (1,0), (1,1)}. Given another graph $G_b$ with Von Neumann neighborhood {(0,0), (-1,0), (0,-1), (1,0), (0,1)}. Then, the north neighbor of node X(i,j), node X(i-1,j), is connected to every node in the Von Neumann neighborhood of node X(i,j) except the south neighbor, node X(i+1,j). Similarly, the south neighbor of node X(i,j), node X(i,j+1), is connected to every node in the Von Neumann neighborhood of node X(i,j) except the north neighbor, node X(i-1,j).

**Proof.**

**Case 1. North Neighbor (node X(i-1,j))**

Node X(i-1,j) is connected to nodes X(i-2,j-1), X(i-2,j), X(i-2,j+1), X(i-1,j-2), X(i-1,j-1), X(i-1,j), X(i-1,j+1), X(i-1,j+2), X(i,j-1), X(i,j) and X(i,j+1). Notice that the (0,0), (-1,0), (0,-1), and (0,1) neighbors of X(i,j), which are the nodes X(i,j), X(i-1,j), X(i,j-1) and X(i,j+1), are in the redundant graph's neighborhood of node X(i-1,j). Thus, the north neighbor of node X(i,j) is connected to every node in the Von Neumann neighborhood of node X(i,j) except the south neighbor.

**Case 2. North Neighbor (node X(i-1,j))**

By symmetry, this case is also true.

THEOREM 1. The Distributed Recovery Strategy-based array algorithm for the White-Gray array reconfigures correctly for single faults for the basic graph $G_b$ (3 x 3) embedded in the arbitrarily large graph $G_r$. (The cases are divided as shown in Figure 33).

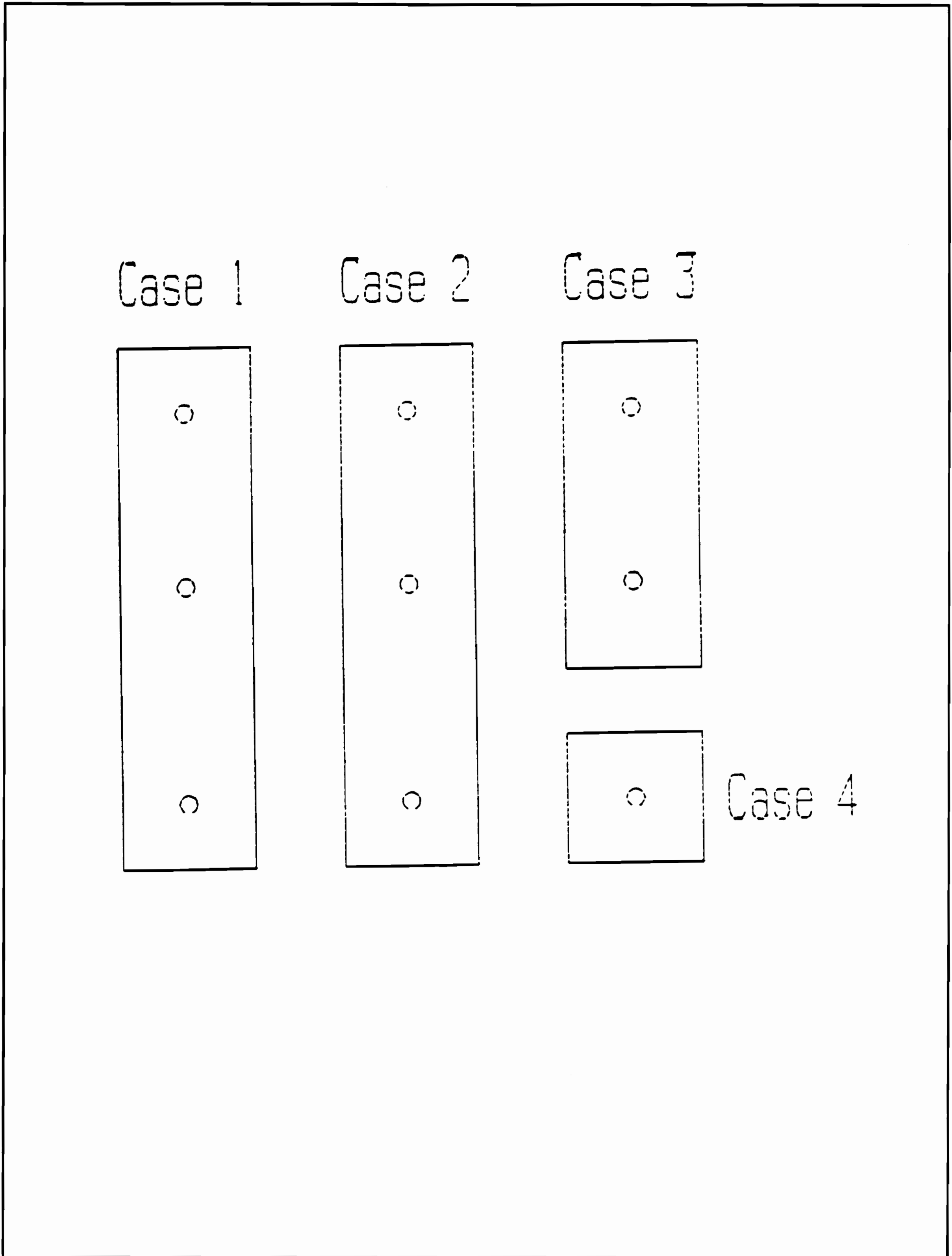



**Figure 33.** Case Division for Theorem 1.

**Proof.**

**Case 1: i = 0,1,2 and j = 0**

By Rule 1, the node in state S(i,j+1) responds to error E(i,j). Since no faults have occurred before this one, node X(i,j+1) is the node in State S(i,j+1). Since no other faults have occurred before this one, all nodes except node X(i,j) are not faulty and node X(i,j) is faulty. Node X(i,j+1) is connected to node X(i,j-1). Node (i,j-1) is not faulty and since j = 0, node X(i,j-1) is a spare (since no other faults have occurred). Since node X(i,j-1) is the direct west neighbor of node X(i,j), node X(i,j-1) is connected to all of the Von Neumann neighbors of node X(i,j), by Lemma 1. Thus, the potential spare, X(i,j-1) has all of the required adjacencies. Therefore, reconfiguration takes place as the spare takes over the missing state.

**Case 2: i = 0,1,2 and j = 1**

By Rule 1, the node in state S(i,j+1) responds to error E(i,j). Since no faults have occurred before this one, node X(i,j+1) is the node in State S(i,j+1). Since no other faults have occurred before this one, all nodes except node X(i,j) are not faulty and node X(i,j) is faulty. The only nodes which are connected to both the east neighbor of the faulty processor (X(i,j-1)) and the responding node X(i,j+1) are the north and south neighbors (X(i-1,j) and X(i+1,j)) of the faulty node and the responding node itself (X(i,j+1)). Thus, these nodes are the only nodes which might be able to take over the state of the faulty node. However,

by Lemma 2, the north neighbor of the faulty node and the south neighbor of the faulty node are not connected. If the north neighbor is a spare, then this node is on the edge of $G_b$ and the south neighbor must be connected to the node which takes over state S(i,j). Similarly, if the south neighbor is a spare, then this node is on the edge of $G_b$ and the north neighbor must be connected to the node which takes over state S(i,j). Both the north and south neighbors cannot be spares. Thus, no spare adjacent to the responding node can take over the state of the faulty node. Since the responding node is the direct east neighbor of the faulty node by Lemma 1, it has all of the necessary connections and thus by the reconfiguration algorithm, the responding node X(i,j+1) takes over state S(i,j). Error E(i,j+1) then exists.

**Case 2a: i = 0,1 j = 1**

The node in state S(i+1, j+1) responds to error E(i,j+1) by Rule 2. Since the only nodes which have changed states from their original states are the faulty node X(i,j) and the node X(i,j+1), each other node X(n,m) has state S(n,m) for n≠i and m≠j, j+1. Thus, the responding node with state S(i+1, j+1) is the node X(i+1,j+1). If a node is connected to all of the nodes in states S(i,j), S(i,j+2), S(i-1,j+1), and S(i+1,j+1), then it will have at least the required adjacencies to take over the state S(i,j+1). The nodes with these states are X(i,j+1), X(i,j+2), X(i-1,j+1) and X(i+1,j+1). These nodes are a subset of the nodes in the Von Neumann neighborhood of X(i,j+1). The east neighbor of

node X(i,j+1) is node X(i,j+2). Thus by Lemma 1, node X(i,j+2) is connected these nodes. Since j = 1, node X(i,j+2) is a spare. Since X(i,j+2) is a spare with the required connections to take over state S(i,j+1), reconfiguration can occur.

**Case 2b: i = 2 j = 1**

By Rule 3, the node in state S(i,j) responds to error E(i,j+1). The only nodes which have changed states from their original states are the faulty node X(i,j) and the node X(i,j+1), each other node X(n,m) has state S(n,m) for n≠i and m≠j, j+1. Thus, the responding node with state S(i,j) is the node X(i,j+1). If a node is connected to the nodes in states S(i,j) and S(i-1,j+1), then it will have the required adjacencies to take over the state S(i,j+1). The nodes with these states are X(i,j+1) and X(i-1,j+1). These nodes are a subset of the nodes in the Von Neumann neighborhood of X(i,j+1). The east neighbor of node X(i,j+1) is node X(i,j+2). Thus by Lemma 1, node X(i,j+2) is connected these nodes. Since j = 1, node X(i,j+2) is a spare. Since X(i,j+2) is a spare with the required connections to take over state S(i,j+1), reconfiguration can occur.

**Case 3: i = 0,1 and j = 2**

By Rule 2, the node in state S(i+1,j) responds to error E(i,j). Since no faults have occurred before this one, node X(i+1,j) is the node in State S(i+1,j). Since no other faults have occurred before this one, all nodes except node X(i,j) are not faulty and node X(i,j) is faulty. Node X(i+1,j) is connected to node X(i,j+1). Since j = 2, node X(i,j+1) is a spare. Since node X(i,j+1) is the direct

east neighbor of node X(i,j), node X(i,j+1) is connected to all of the Von Neumann neighbors of node X(i,j), by Lemma 1. Thus, the potential spare, X(i,j+1) has all of the required adjacencies. Therefore, reconfiguration can take place.

**Case 4: i = 2 and j = 2**

By Rule 3, the node in state S(i,j-1) responds to error E(i,j). Since no faults have occurred before this one, node X(i,j-1) is the node in State S(i,j-1). Since no other faults have occurred before this one, all nodes except node X(i,j) are not faulty and node X(i,j) is faulty. Node X(i,j-1) is connected to node X(i,j+1). Since j = 2, node X(i,j+1) is a spare. Since node X(i,j+1) is the direct east neighbor of node X(i,j), node X(i,j+1) is connected to all of the Von Neumann neighbors of node X(i,j), by Lemma 1. Thus, the potential spare, X(i,j+1) has all of the required adjacencies. Therefore, reconfiguration can take place.

**Case 5: i ≠ 0,1,2 and j ≠ 0,1,2 (Spares)**

Each node X(n,m) is not a spare for n = 0,1,2 and m = 0,1,2. Since no other faults have occurred, every other node is a non-faulty spare. Thus, if node X(i,j) is faulty, reconfiguration will not be needed since node X(i,j) is not a node in the basic graph $G_b$.

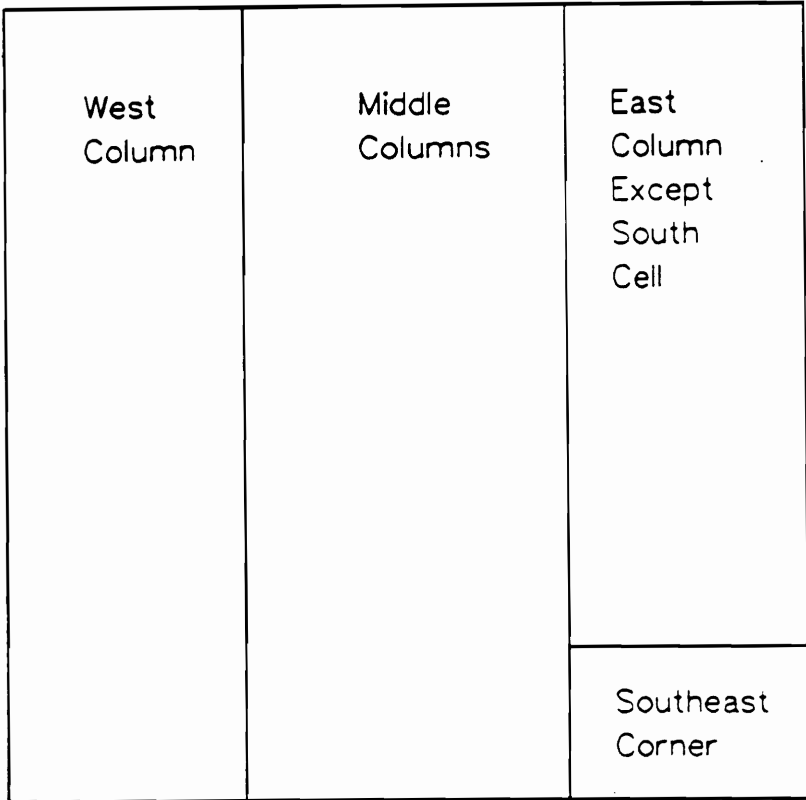



**Figure 34.** Case Division for Theorem 2.

Theorem 2. The Distributed Recovery Strategy-based array reconfiguration algorithm for the White-Gray array reconfigures correctly for single faults for the basic graph $G_b$ (N x M) embedded in the arbitrarily large graph $G_r$. (The case division is shown in Figure 34).

**Case 1: i = 0,1,...,N-1 and j = 0 (Direct Proof)**

By Rule 1, the node in state S(i,1) responds to error E(i,j). Since no faults have occurred before this one, node X(i,1) is the node in State S(i,1). Since no other faults have occurred before this one, all nodes except node X(i,0) are not faulty and node X(i,0) is faulty. Node X(i,1) is connected to node X(i,-1). Node (i,-1) is not faulty and it is a spare (since no other faults have occurred). Since node X(i,-1) is the direct west neighbor of node X(i,0), node X(i,-1) is connected to all of the Von Neumann neighbors of node X(i,0), by Lemma 1. Thus, the potential spare, X(i,-1) has all of the required adjacencies. Therefore, reconfiguration can take place.

**Case 2: i = 0,1,...,N-1 and j = 1,...,M-2 (Induction on M)**

**Base Case: M = 3 (That is, i = 0,1,...,N-1 and j = 1)**

By Rule 1, the node in state S(i,2) responds to error E(i,1). Since no faults have occurred before this one, node X(i,2) is the node in State S(i,2). Since no other faults have occurred before this one, all nodes except node X(i,1)

are not faulty and node X(i,1) is faulty. The only nodes which are connected to the east neighbor of the faulty processor (X(i,0)) and the responding node X(i,2) are the north and south neighbors (X(i-1,1) and X(i+1,1)) of the faulty node and the responding node itself (X(i,2)). Thus, these nodes are the only nodes which might be able to take over the state of the faulty node. However, by Lemma 2, the north neighbor of the faulty node and the south neighbor of the faulty node are not connected. If the north neighbor is a spare, then this node is on the edge of $G_b$ and the south neighbor must be connected to the node which takes over state S(i,1). Similarly, if the south neighbor is a spare, then this node is on the edge of $G_b$ and the north neighbor must be connected to the node which takes over state S(i,1). Both the north and south neighbors cannot be spares. Thus, no spare adjacent to the responding node can take over the state of the faulty node. Since the responding node is the direct east neighbor of the faulty node, it has all of the necessary connections and thus by the reconfiguration algorithm, the responding node X(i,2) takes over state X(i,1). Then, error E(i,2) is present.

**Case 2a: i = 0,1,...,N-2 and j = 1**

By Rule 2, the node S(i+1,2) responds to error E(i,2). Since the only nodes which have changed states from their original states are the faulty node X(i,1) and the responding node X(i,2), each other node X(n,m) has state S(n,m) for $n \neq i$ and $m \neq 1,2$. Thus the responding node with state S(i+1,2) is the node

X(i+1,2). If a node is connected to all of the nodes in states S(i,1), S(i,3), S(i-1,2), and S(i+1,2), then it will have at least the required adjacencies to take over the state S(i,2). The nodes with these states are X(i,2), X(i,3), X(i-1,2) and X(i+1,2). These nodes are a subset of the nodes in the Von Neumann neighborhood of X(i,2). The east neighbor of node X(i,2) is node X(i,3). Thus by Lemma 1, node X(i,3) is connected these nodes. Node X(i,3) is a spare. Since X(i,3) is a spare with the required connections to take over state S(i,2), reconfiguration can occur.

**Case 2b: i = N-1 j = 1**

By Rule 3, the node in state S(i,1) responds to error E(i,2). The only nodes which have changed states from their original states are the faulty node X(i,1) and the original responding node X(i,2), each other node X(n,m) has state S(n,m) for $n \neq i$ and $m \neq 1,2$. Thus, the responding node with state S(i,1) is the node X(i,2). If a node is connected to the nodes in states S(i,1) and S(i-1,2), then it will have the required adjacencies to take over the state S(i,2). The nodes with these states are X(i,2) and X(i-1,2). These nodes are a subset of the nodes in the Von Neumann neighborhood of X(i,1). The east neighbor of node X(i,2) is node X(i,3). Thus by Lemma 1, node X(i,3) is connected these nodes. Node X(i,3) is a spare. Since X(i,3) is a spare with the required connections to take over state S(i,2), reconfiguration can occur.

**Induction hypothesis**: Assume reconfiguration can take place for any M. That is, i = 0,1,...,N-1 and j = 1,2,...,M-2.

**Induction Step**: Show that reconfiguration can take place under the induction hypothesis for i = 0,1,...,N-1 and j = 1,2,...,M-1.

By Rule 1, the node in state S(i,j+1) responds to error E(i,j). Node X(i,j+1) is in State S(i,j+1). Since no other faults have occurred before this one, all nodes except node X(i,j) are not faulty and node X(i,j) is faulty. The only nodes which are connected to the east neighbor of the faulty processor (X(i,j-1)) and the responding node X(i,j+1) are the north and south neighbors (X(i-1,j) and X(i+1,j)) of the faulty node and the responding node itself (X(i,j+1)). Thus, these nodes are the only nodes which might be able to take over the state of the faulty node. However, by Lemma 2, the north neighbor of the faulty node and the south neighbor of the faulty node are not connected. So, if the north neighbor is a spare, then this node is on the top edge of $G_b$ and the south neighbor must be connected to the node which takes over state S(i,j). Similarly, if the south neighbor is a spare, then this node is on the bottom edge of $G_b$ and the north neighbor must be connected to the node which takes over state S(i,j). Thus, neither the north neighbor nor the south neighbor can take over the error state S(i,j). Both the north and south neighbors cannot be spares. Thus, no spare adjacent to the responding node can take over the state of the faulty

node. Since the responding node is the direct east neighbor of the faulty node, it has all of the required connections. Thus by the reconfiguration algorithm, the responding node X(i,j+1) takes over state X(i,j). No nodes to the east of the responding node (X(i,j+1)) are affected by this change. Then the error condition E(i,j+1) occurs. The node in state S(i,j+2) responds the error by Rule 1. Thus, node X(i,j+2) is the node in state S(i,j+2). So, since no other nodes are affected by the previous change, by the induction hypothesis reconfiguration can take place for the rest of the columns. Therefore, reconfiguration can take place for i = 0,1,...,N and j = 1,...,M-2.

**Case 3: i = 0,1,...,N-2 and j = M-1 (Direct Proof)**

By Rule 2, the node in state S(i+1, M-1) responds to error E(i,M-1). Since no faults have occurred before this one, node X(i+1,M-1) is the node in State S(i+1,M-1). Since no other faults have occurred before this one, all nodes except node X(i,M-1) are not faulty and node X(i,M-1) is faulty. Node X(i+1,M-1) is connected to node X(i,M). Node (i,M) is a spare. Since node X(i,M) is the direct east neighbor of node X(i,M-1), node X(i,M) is connected to all of the Von Neumann neighbors of node X(i,M-1), by Lemma 1. Thus, the potential spare, X(i,M) has all of the required adjacencies. Therefore, reconfiguration can take place.

**Case 4: i = N-1 and j = M-1. (Direct Proof)**

By Rule 3, the node in state S(N-1, M-2) responds to error E(N-1,M-1). Since no faults have occurred before this one, node X(N-1,M-2) is the node in State S(N-1,M-2). Since no other faults have occurred before this one, all nodes except node X(N-1,M-1) are not faulty and node X(N-1,M-1) is faulty. Node X(N-1,M-2) is connected to node X(N-1,M). Node X(N-1,M) is a spare. Since node X(N-1,M) is the direct east neighbor of node X(N-1,M-1), node X(N-1,M) is connected to all of the Von Neumann neighbors of node X(N-1,M-1), by Lemma 1. Thus, the potential spare, X(N-1,M) has all of the required adjacencies. Therefore, reconfiguration can take place.

**Case 5: i ≠ 0,1,...,N-1 and j ≠ 0,1,2,...,M-1 (Spares--Direct Proof)**

Each node X(n,m) is not a spare for n = 0,1,...,N-1 and m = 0,1,...,M-1. Since no other faults have occurred, every other node is a non-faulty spare. Thus, if node X(i,j) is faulty, reconfiguration will not be needed since node X(i,j) is not a node in the basic graph $G_b$.

### 2.2.4 Limitations of the Distributed Recovery Strategy

The Yanney-Hayes Distributed Recovery Strategy is appropriate for many different architectures. However, it does not provide any method for designing a reconfigurable system. It does provide an outline of a reconfiguration algorithm, but the details of the design process are not

specified. A model based on this strategy needs to give a clear method for the design of reconfigurable systems to be useful.

Another difficulty with the Distributed Recovery Strategy is the requirement that only one error be present in the system at a given time. Multiple faults and near-coincident faults cannot be handled. In highly reliable systems, multiple faults must be handled. Modifications need to be made to the strategy to provide these important reconfiguration capabilities. These problems will be further considered in a later chapter.

## 2.3 Description of the Local Supervisor Model

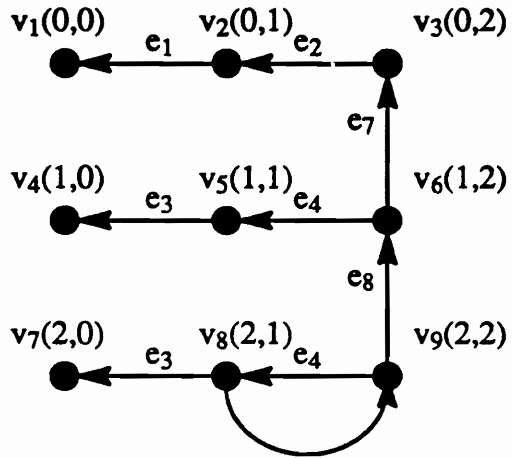
In the application of the Distributed Recovery Strategy to an array architecture, the need for a method of designing algorithms based on this strategy became evident. In addition, the use of the Distributed Recovery Strategy as the basis of a model of distributed reconfiguration was identified as potentially valuable. This section describes the Local Supervisor Model which is based on the Distributed Recovery Strategy and which provides a method for designing distributed reconfiguration algorithms.

For the ADAS simulations of the Distributed Recovery Strategy-based array algorithm, an error response set was chosen simply by considering several choices and making the choice that reconfigured correctly for the most single and double sequential faults. However, there is no defined way to determine whether the error response set chosen was the optimum choice. The

attempt to choose the best possible error response set identified the need for a method which would produce a "good" error response set.

An easily understood description is needed to choose error response sets. Error response sets can be described in several equivalent formats. Previously in this report, a table has been used and descriptive rules have been used. In this section, a third form of representing error response sets is suggested. The suggested approach is particularly useful for making the error response assignments. It describes the error response sets in a visual way which allows quick comprehension of the actual assignments. This section will also suggest several guidelines which should be used in choosing an error response set.

A directed graph (digraph) is proposed as a way to describe the error response sets. Let the tail of an edge be connected to the responding state. Let the head of an edge be connected to the error state. The digraph will be superimposed on the basic graph $G_b$. This graphical representation presents an easily understandable picture of the error response assignments. The digraph method of representing the error response sets is equivalent to the representation used in Yanney and Hayes' paper [Yan86]. The set of all of the nodes which are connected to the node S(i,j) for which the tail of the connecting edge is at the node S(i,j) are the error conditions to which the node in state S(i,j) responds. The error response digraph for the implementation described is shown in Figure 35(a). Using the formal definition of a directed graph, V(D)



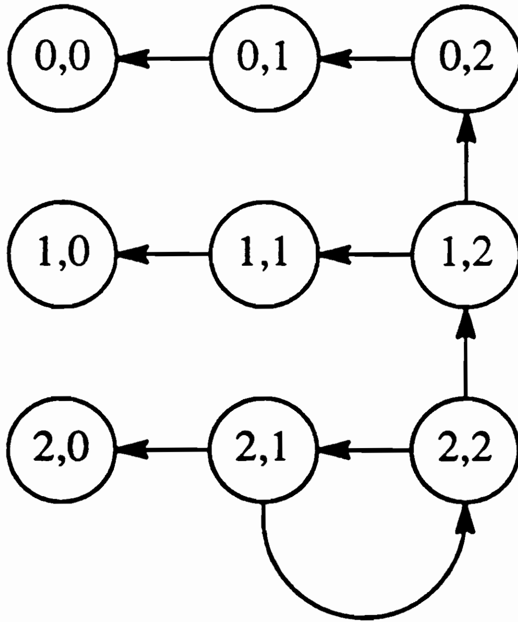




**Figure 35.** Error Response Digraph

= $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$ and A(D) = $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9\}$. The incidence function $\Psi_D$ is defined as follows:

$$\Psi_D(a_1) = (v_2, v_1)$$
$$\Psi_D(a_2) = (v_3, v_2)$$
$$\Psi_D(a_3) = (v_5, v_4)$$
$$\Psi_D(a_4) = (v_6, v_5)$$
$$\Psi_D(a_5) = (v_8, v_7)$$
$$\Psi_D(a_6) = (v_9, v_8)$$
$$\Psi_D(a_7) = (v_6, v_3)$$
$$\Psi_D(a_8) = (v_9, v_6)$$
$$\Psi_D(a_9) = (v_8, v_9)$$

Figure 35(b) shows a simpler sketch of the error response digraph which does not include node and arc names.

The three absolute constraints which must be met in assigning the error response digraph are given in Figure 36. In summary, there must exist an edge in $G_b$ with endpoints at the responding state and the error state. Also, each node in the digraph must have exactly one head of an arc incident on it. Some guidelines which should be used in conjunction with these constraints to choose a good error response digraph have been identified. These guidelines are listed in Figure 37.

If a cycle exists in the error response digraph, there is a possibility of the reconfiguration process continuing endlessly. If the cycle is large, it is less likely that the infinite reconfiguration condition will occur than for a smaller cycle. The size of the cycle is important, because large cycles provide more opportunities for a spare to take over before the infinite reconfiguration condition occurs. Thus, short cycles should be avoided or chosen so that an

1) The responding state must be connected to the error state in the basic graph

2) Only one state responds to a particular error state

3) Each error state is responded to by some other state

**Figure 36.** Constraints on Error Response Set Choice

1) Avoid short cycles

2) It is desirable for the responding node to be connected in the redundant graph to all of the error node's neighbors in the basic graph for the initial case

3) Choose assignments so that growth tends to go in the same direction

**Figure 37.** Guidelines for Error Response Set Choice

infinite reconfiguration condition cannot occur. For example, if a cycle is chosen in which one of the nodes in the cycle cannot take over the error state, then an infinite loop cannot occur. In other words, if some responding node in the cycle is not connected to all of the neighbors of the error state in $G_b$, then reconfiguration will fail in a normal way without looping endlessly.

Another guideline was developed to provide for the continuation of the reconfiguration process if a spare is not located immediately. If the reconfiguration process can continue, then at least it will not fail immediately. If the responding state is connected to all of the neighbors in the error state in $G_b$ and a spare to take over the error state is not available, then the responding state can take over the error state (which creates a new error condition) and the reconfiguration process can continue. Thus, in choosing the digraph, you should try to choose edge assignments so that the responding state is connected to all of the error states' neighbors in the basic graph $G_b$ for the initial configuration.

For multiple faults, reconfiguration is more likely to be able to take place if the growth of the basic graph $G_b$ embedded in $G_r$ tends to grow in the same directions. If reconfiguration results in growth in opposite directions, the graph tends to be pulled apart.

With these guidelines in mind, consider the error response set chosen previously for the White-Gray array. The error response set chosen is fairly good. It follows the second and third guidelines; however, the chosen error

response set contains a short cycle which can result in the infinite looping error. This infinite looping error occurs for the double sequential fault E(9)E(9).

The Local Supervisor Model is more versatile and more practical than the preliminary strategy presented by Yanney and Hayes. The algorithm designed using this strategy can be implemented as described above. However, the implementation could be approached differently. The LSM can be used to design the behavior of an algorithm. Then, the implementation may be streamlined to produce a faster, more efficient implementation than the straight-forward implementation of the strategy.

## 2.4 Use of the Local Supervisor Model

The guidelines stated for choosing error response assignments enable the LSM to be used to design algorithms as well as describe algorithms. Other areas which also need to be considered are the use of spares as local supervisors, the difficulty of predicting the algorithm's coverage, and the handling of multiple faults and near-coincident faults. These problems will be addressed in a later chapter. By having a model which can be used to describe algorithms presented by other researchers, a more precise and theoretical study of reconfiguration algorithms is possible.

A modified version of the Local Supervisor Model is developed in a later chapter which addresses the problems listed above. The Local Supervisor Model is also used to describe the White-Gray Local Algorithm. In addition,

an improved LSM Array algorithm based on the White-Gray array is presented which has better coverage than the White-Gray Local Algorithm.

## 2.5 Summary

The Distributed Recovery Strategy [Yan86] provides a backbone for distributed reconfiguration algorithms. The Distributed Recovery Strategy has been applied to trees and cycles [Yan86]. In this research, the Distributed Recovery Strategy was used as a basis for a reconfiguration algorithm for a redundant array architecture. The Array Reconfiguration Algorithm was modeled using an Architecture Design and Assessment System (ADAS). The algorithm covers all single faults and 83% of double sequential faults. The Distributed Recovery Strategy does not allow multiple faults to be present in the system.

The Array Reconfiguration Algorithm is proven to reconfigure correctly for all single faults with a basic graph of size 3 X 3, and in general for a basic graph of size N X M. For convenience in handling boundary conditions, the redundant array is assumed to be arbitrarily large.

The Distributed Recovery Strategy has two major limitations. First, it does not provide a method for designing reconfigurable systems. It only provides an outline for a reconfiguration strategy. Second, it only allows one error to be present in the system at a given time. This limitation means that no multiple or near-coincident faults can be handled.

The Local Supervisor Model (LSM) uses the Distributed Recovery Strategy as a basis of a method for designing reconfigurable systems. Thus, the LSM corrects the first problem with the Distributed Recovery Strategy by providing a method of design. The LSM is also limited to handling only a single error in the system at a time. This restriction will be addressed in a later chapter.

The Local Supervisor Model provides a method for designing distributed reconfigurable systems. The LSM can also be used as a tool for evaluation and comparison. The use of LSM in the comparison of the Distributed Recovery Strategy-based array reconfiguration algorithm described in this chapter and the White-Gray Local algorithm is discussed in a later chapter.

# 3.0 Tessellation Automata Model

The shortcomings of the Local Supervisor Model lead to a new model which is based on tessellation automata. First, this chapter describes the model. Then, a formal definition is stated. Finally, this model is used to describe existing algorithms.

## 3.1 Description of the Tessellation Automata Model

The Local Supervisor Model presented in Chapter 2 provides a method for describing, comparing, and designing systems. However, it does not provide any method for describing the implementation and other details of an algorithm. It only allows the description of the algorithm's behavior. The Local Supervisor Model does not allow coincident and near-coincident faults. A modification of the LSM presented in a later chapter partially overcomes this difficulty. The Local Supervisor Model provides little flexibility in describing algorithms. In many cases, a more flexible model is needed.

The LSM is based on the principle that for distributed algorithms, local behavior is based only on local information. In addition, it uses the idea that some local control is necessary to coordinate the activities of cells. This idea

underlies the use of local supervisors to determine what happens. The Tessellation Automata Model (TAM) presented in this chapter is an extension of the ideas of the LSM. In the TAM, each cell determines its actions based on its neighbors' states and its own state. In contrast to the LSM, a local supervisor is not required to direct local activities. This difference allows the TAM to be more flexible. A tradeoff for increased flexibility is that the TAM does not provide the same details of a strategy that the LSM does. However, the TAM has several parameters which can be used to compare and evaluate systems. These same parameters can be used to develop a design which meets certain specifications. The clear specification of system parameters gives the TAM an advantage over the LSM.

Tessellation automata was chosen as the basis for this model, because it is inherently distributed in nature. In any distributed algorithm, local decisions are based on local information. In the TAM, the neighborhood specifies which cells' information is needed to determine a cell's activity. Thus, tessellation automata was a natural choice for a modeling tool.

Before discussing the details of the TAM, tessellation automata need to be described. Reasons for choosing tessellation automata as the model's basis are given. The differences between the TAM and tessellation automata are also discussed.

### 3.1.1 Tessellation Automata

Tessellation Automata (TA) are closely related to cellular automata. In the most general form, cellular automata does not require a regular architecture [Bur70]. A tessellation automaton is a formal structure based on the idea of an infinite n-dimensional array of finite state machines. Each finite state machine in the tessellation automata is called a cell. The cells in the TA are connected in a uniform fashion to other cells. The cells that are connected to a particular cell are called the neighbors of that cell and comprise its neighborhood. The spatial relationships between a cell and its neighbors are regular throughout the array. There are an infinite number of cells in the structure. [Yam70]

Formally, a tessellation automaton is defined as

$$M = (A, E^d, X, I)$$

$A$ denotes the state alphabet of the tessellation automata $M$. It is a set containing all possible states for all of the cells in the automaton. $d$ is a positive integer denoting the tessellation dimension. $E^d$, the tessellation array, is a $d$-dimensional Euclidean space. $E^d$ is the set of all $d$-tuples of integers. The elements of $E^d$ are used as names for the cells in the automaton. $X$ is an $n$-tuple of distinct $d$-tuples of integers where $n$ is the number of cells to which each cell is directly connected. $X$ is called the neighborhood index and is used to define the interconnection pattern for the array. To determine the $n$ neighbors of a cell, each $d$-tuple in $X$ is added to the $d$-tuple which is the cell's

name. $I$ is a set of next state functions. At a given time step, the same next state function is applied to each cell. [Yam70] A simplification can be made by allowing only a single next state function, $\sigma$. [Kum84] This simplification does not affect my model.

### 3.1.2 Reasons for Choosing Tessellation Automata as a Basis for a Model

A model is needed which can be used for any regular, distributed architecture. A regular architecture is an architecture for which the degree of all cells is the same. Regular architectures include many of the most useful parallel architectures, such as hypercubes and nearest-neighbor meshes. The model should also be flexible enough to describe complicated algorithms, as well as simple algorithms. If distributed systems can be described, the systems with global control can also be described as a special case. The model should have parameters that differentiate architectures in a manner that is valuable in considering them as fault tolerant systems. A hierarchical model would allow different levels of modeling to accomplish different goals.

In many respects, the tessellation automata fits these requirements. It can be used for any system whose architecture can be considered to be a Euclidean space of processors with uniform connections between neighbors. Tessellation automata can be used to describe distributed systems, since local transformations are used to effect global changes.

In other ways, tessellation automata do not directly meet these requirements. Tessellation automata cannot be conveniently used to model all regular architectures; however, they are convenient for a great number of them. In addition, a more flexible format for the next state function is needed. The next state function is not conducive to modeling complicated algorithms because of the very large number of states typically involved. The parameters of a tessellation automaton are appropriate for a fault tolerant system model, but more information could be helpful. No direct hierarchical description is associated with tessellation automata.

In summary, the parameters of the model need to be defined more clearly with the goal of modeling reconfigurable systems in mind. Also, a hierarchical view of the modeling process needs to be developed. These problems of using the tessellation automata directly for modeling the system are overcome in the Tessellation Automata Model.

### 3.1.3 Description of the Tessellation Automata Model

The potential difficulties of using tessellation automata as the basis for modeling distributed algorithms are overcome by the Tessellation Automata Model. This section identifies the differences between the proposed Tessellation Automata Model and Tessellation Automata, and then describes the TAM in detail.

The TAM has an associated graph structure which allows all point-symmetric architectures to be modeled. A graph G is <u>point-symmetric</u> if each

pair of vertices $u$ and $v$ are similar. Two vertices are similar if for some automorphism of G, $\alpha(u) = v$ [Har69]. The TAM removes the tessellation automata's requirement of embedding the architecture in a Euclidean array. The TAM model also adds boundary cells so all boundary conditions can be handled gracefully. This addition allows the modeling of all architectures which are point-symmetric except for the boundaries--for example, a bounded mesh architecture. The Tessellation Automata Model allows the use of pseudocode in the description of the next state function to increase flexibility and reduce complexity. The Tessellation Automata Model also specifies the relationships between the tessellation automata parameters and reconfigurable architectures. Finally, the Tessellation Automata Model is hierarchical.

In general terms, the Tessellation Automata Model (TAM) describes a system by a graph and a next state function. The system represented by the TAM must be regular except for boundary conditions. The initial state of the system is also incorporated. Each processing element in the system has a corresponding cell in the TAM. All cells in the TAM that correspond to processing elements are termed active cells. The TAM handles boundary conditions conveniently through the use of special boundary cells, also called inactive cells.

Each processing element in the system is represented by a node in the graph. Communication links between processing elements are represented by edges between the nodes. The graph is actually represented by listing the

neighbors of each processing element. The order that the neighbors of a processing element are listed indicates their relative position to that processing element. For example, in a system where each node has north, south, east, and west neighbors, the north neighbor could be assigned to the first position in the ordered list, the south neighbor could be assigned to the second position in the ordered list, and so on. The choice of ordering is arbitrary but must be used consistently throughout the graph description.

The next state function, $\sigma$, is applied to each cell in the model at each time step. The next state of each cell is determined by the states of the cells in its neighborhood. A cell's neighborhood is the set of cells to which it is adjacent. A cell's neighborhood typically includes the cell itself.

Boundary conditions are handled in the model by adding "inactive cells" to the graph. Inactive cells are cells that have a constant state and that do not correspond to processing elements in the actual system. They may represent inputs or outputs, or they may represent default boundary values. The degree of all active cells is required to be the same. By adding inactive cells, the degree of each active cell can be controlled.

## 3.2 Formal Definition of the Tessellation Automata Model

Formally, the Tessellation Automata Model is defined as follows:

$$TAM = (Q, A, B, G, \sigma, c_0)$$

where

$Q$ is a finite, non-empty set of cell states

$A$ is a finite set of active cells

$B$ is a finite set of inactive boundary cells

$G$ is an $n$-tuple of $p$-tuples where $n = |A|$ and $p$ is the number of neighbors that each active cell has.

$\sigma: Q^p \rightarrow Q$ is the next state function

$c_i: A \cup I \rightarrow Q$ is the global configuration at time $i$

Notice that $c_0$ is the global configuration at time 0.

The modeling process is hierarchical and iterative in nature. Different level models can be developed depending on the analysis needs. For example, to choose between a number of alternatives, you can

1. Develop a high-level model and quickly eliminate the most unsuitable alternatives.
2. Expand the high-level model into a more detailed intermediate model which considers the total algorithm. This step can be used to eliminate all but a small number (one to three) of algorithms from consideration.
3. Develop a simulation model to study the details of a small number of selected algorithms and make a final choice.

The time required to develop a model is related to the level of detail needed. Most analysis does not require a detailed simulation model. This advantage allows most of the time to be spent analyzing the most promising candidates.

The modeling process is divided into three levels. Level 1 is the highest level of modeling--requiring the least amount of modeling time. Level 1 only

considers the most obvious parameters. Level 2, the middle level, considers all of the parameters of the model, but does not simulate the activity of the system. Level 2 requires more modeling time than Level 1. Level 3 is the lowest, most detailed level. It requires the most modeling time, since it simulates the detailed activities of the reconfigurable system.

Heuristic rules for modeling an existing architecture at each level have been developed. These heuristic rules provide a method for modeling existing architectures. First, a Level 1 model should be developed, then a Level 2 model, and finally a Level 3 model.

The following procedure should be used to develop a Level 1 model. This level defines the structure of the model and identifies important characteristics of the system being considered.

<u>Level 1</u>

1-1 Develop a facility graph [Hay76] to represent the redundant hardware system.

1-2 The set of active cells, A, corresponds to the nodes of the graph developed in Step 1-1.

1-3 Add boundary cells so that the graph is regular and has consistent neighborhood structure. These boundary cells are actually just the boundary conditions for the regular architecture. Any nodes which require the same boundary conditions can be connected to the same boundary cell. Consequently, there are no

restrictions on the degree of a boundary cell. B is the set of these boundary cells. The resulting graph corresponds to G in the TAM.

1-4 Identify the information that is passed between nodes, and the information that is needed by the cell at each time step to determine its next state. Add any needed edges to the graph to show this flow of information.

After developing a Level 1 TAM, a Level 2 TAM can be developed. The following guidelines should be used. A Level 2 model identifies all of the TAM parameters.

Level 2

2-1 Identify all situations in which the state changes and also in which new information needs to be sent to neighbors.

2-2 Write pseudocode or develop a table for the next state function, $\sigma$. The next state function should correspond directly with the hardware of the architecture.

This process is iterative. As you develop the Level 2 model, you will clarify the Level 1 model and add any overlooked information.

After Level 1 and Level 2 models have been developed, a Level 3 model is used. The Level 3 model simulates the activity of the reconfigurable system. The steps for developing a Level 3 model are outlined below.

Level 3

3-1 Using the pseudocode developed for LEVEL 2, write functions which implement the next state function, σ, using a computer programming language, such as "C" or "PASCAL."

3-2 Use a general structure main routine which performs all needed initialization and applies the next state function to each cell each time step.

3-3 Simulate the algorithm and observe its operation. Coverage and time requirements can be determined.

This hierarchical process is a very valuable way to consider many architectures quickly. It can be used to streamline the design and comparison processes. The modeling process is well-defined and can also be used to evaluate a single reconfigurable system.

The graph is represented by listing the neighboring nodes for each node in the graph. Although this is somewhat redundant, it is more convenient for simulation processing which is based on handling each node and information from the node's neighbors. Some of the redundancy is eliminated by listing the neighboring nodes for only the nodes corresponding to the active cells in the system. Since each inactive cell must be connected to some active cell, this description of the graph is valid. Since there are $n$ active nodes, and since the degree of each active node is $p$, an $n$-tuple of sets with $p$ elements is needed to describe the graph in the above format.

This graph representation completely defines the fault tolerant system; however, by adding information about the relationships between neighbors, a more useful description is generated. Rather than have an $n$-tuple of sets of $p$ elements, the graph is defined with an $n$-tuple of $p$-tuples. The order of the elements of the $p$-tuples describes the relationship between neighbors. Some analysis is required before this neighbor relationship information can be added. In the previous example, the modeler would have to identify which neighbor is the North neighbor, the East neighbor, the South neighbor, and the West neighbor. Then, the mesh can be represented where the first element of the 4-tuple is the NORTH neighbor, the second is the EAST, the third is the SOUTH, and the fourth is the WEST. In the example, this problem is not very difficult. However, in general, there may not be a readily identifiable pattern.

## 3.3 Using the Tessellation Automata Model to Describe Algorithms

This section describes the use of the Tessellation Automata Model as a tool for describing and evaluating reconfigurable systems. First, the Direct Reconfiguration Algorithm presented by Sami [Sam86] is considered. Then, the Yanney-Hayes Distributed Recovery Strategy [Yan86] is modeled. Next, the LSM Array Algorithm is modeled.

### 3.3.1 Using TAM to Model the Direct Reconfiguration Algorithm

The Direct Reconfiguration Algorithm presented by Sami [Sam86] is a straight-forward approach to array reconfiguration. The algorithm is

implemented in hardware. The Direct Reconfiguration Structure assumes that each cell tests itself. The signal e(i,j) is TRUE if the cell is faulty and FALSE if the cell is not faulty. Link failures are considered the same as if the cell sending information across that link were faulty. A row of spares and a column of spares are used.

When a faulty cell is detected, that cell must be marked as either a vertical fault or a horizontal fault. Vertical faults involve the spare in the spare row, while horizontal faults involve the spare in the spare column. After faults are marked as horizontal or vertical (determining the direction of reconfiguration), the functions of the cells are mapped to the active cells. No more than one fault in any row can be marked as horizontal, and no more than one fault in any column can be marked as vertical. A further limitation is imposed to simplify the assignment problem: the southern-most fault in a column is marked as a vertical fault. Horizontal reconfiguration, or renaming, is performed first, followed by vertical renaming. In other words, states are reassigned along rows and then along columns. The circuit which controls this reconfiguration is shown in Figure 38. If the signal $e_x(i,j)$ is TRUE, then the cell requires horizontal reconfiguration. If the signal $e_y(i,j)$ is TRUE, then the cell requires vertical reconfiguration. An example is shown in Figure 39. Further details of the direct reconfiguration structure are given in [Sam86].

The TAM for the Direct Reconfiguration Structure is given below. The hierarchical procedure for developing a model is used. A 6 X 6 direct

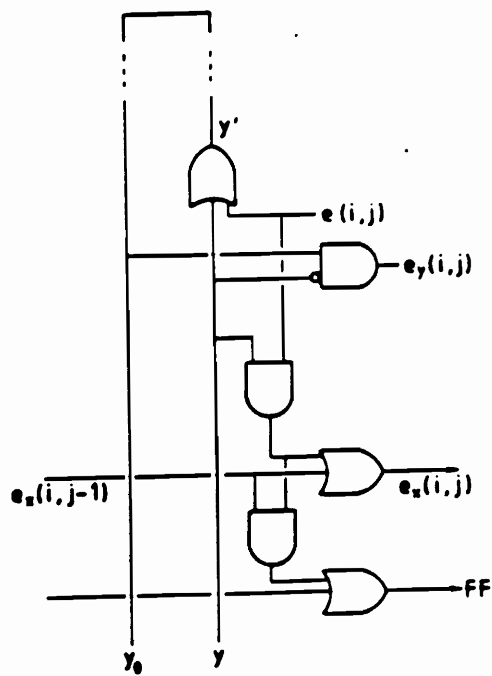
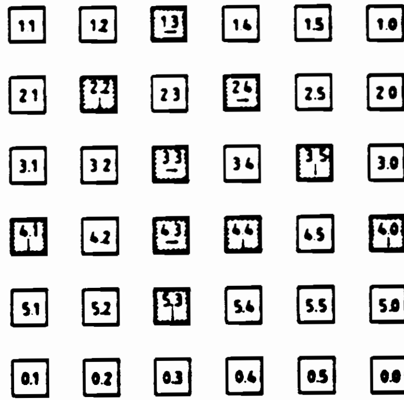
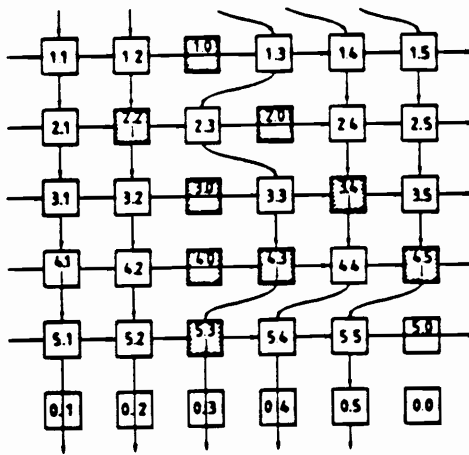



**Figure 38.** Control Circuit for the Direct Reconfiguration Structure [Sam86]



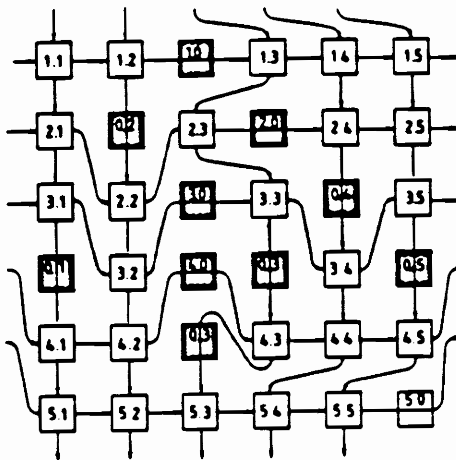




**Figure 39.** Example of Direct Reconfiguration [Sam86]

reconfiguration array is considered. One of the columns and one of the rows comprise the spare cells for the structure.

Level 1

1-1 The facility graph representing the redundant hardware system is a completely connected graph. A completely connected graph is used because the extreme cells must have complete knowledge of the rest of the system. In the direct reconfiguration structure, direct connections are not used between all cells. Instead, the information is passed through the combinational logic throughout the array. Thus, this combinational logic is hard core.

1-2 The set of active cells, A, is the set of nodes in the facility graph found in Step 1-1.

A = {(1,1), (1,2), (1,3), (1,4), (1,5), (1,0), (2,1), (2,2), (2,3), (2,4), (2,5), (2,0), (3,1), (3,2), (3,3), (3,4), (3,5), (3,0), (4,1), (4,2), (4,3), (4,4), (4,5), (4,0), (5,1), (5,2), (5,3), (5,4), (5,5), (5,0), (0,1), (0,2), (0,3), (0,4), (0,5), (0,0)}

1-3 A single boundary cell is added to the graph. Boundary cells can be added in many different ways. Edges are added between the active nodes and the boundary cell to make the degree of each active cell the same. Each active cell has the same neighborhood connections. The neighborhood connections for each active cell are shown in Figure 40.

1-4 Each cell passes its logical state to its south neighbor, its east neighbor, and its southeast neighbor. To determine its next state a cell needs to

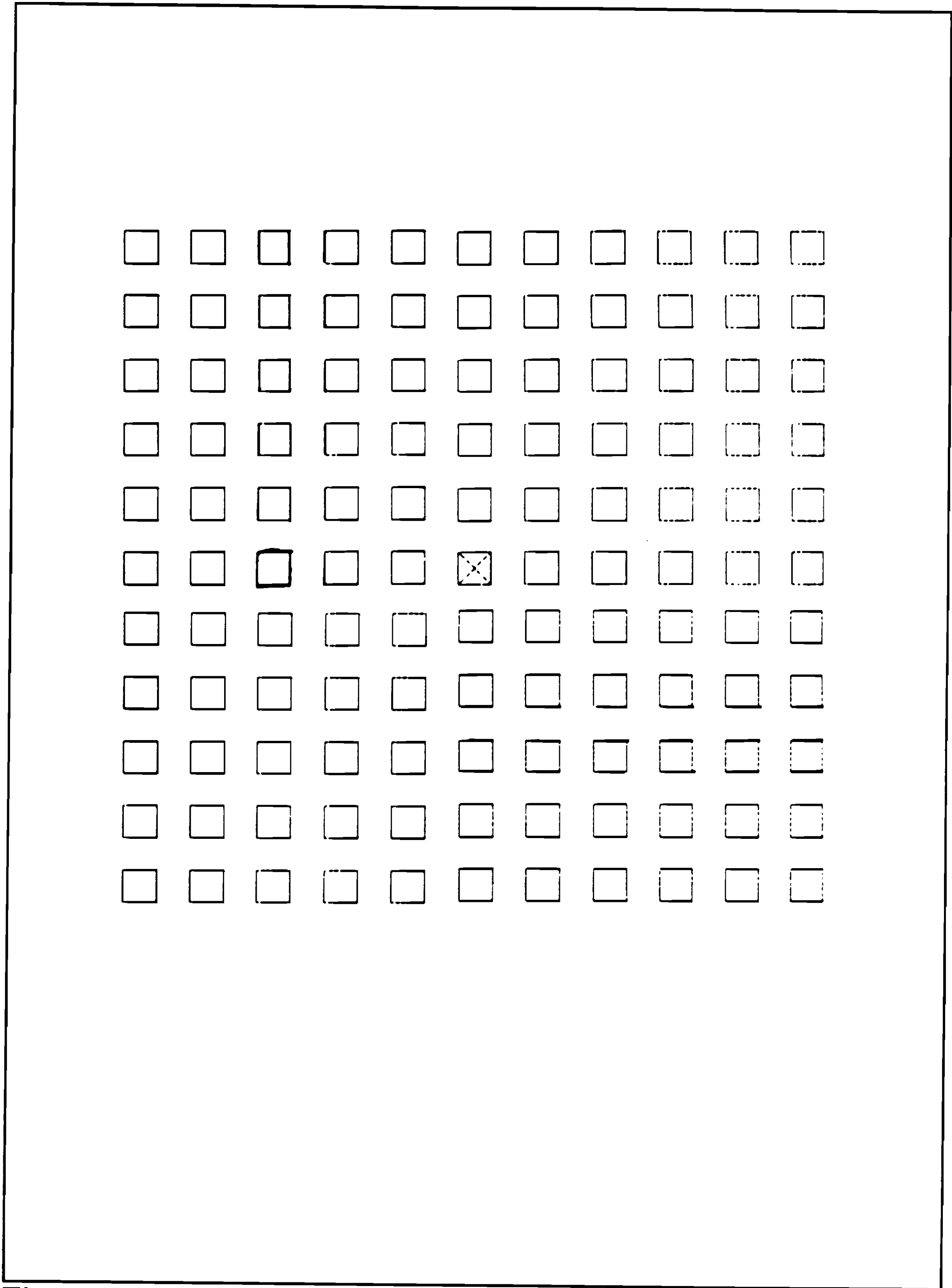


**Figure 40.** Neighborhood of TAM of Direct Reconfiguration

know its north neighbors's logical state, its west neighbor's logical state, its northwest neighbor's logical state, and information about where any faults in the system are. To determine whether a fault should be marked as horizontal or vertical, information about whether any cells in that column below the fault are faulty is needed. To determine what logical state a cell should have, the cell must know if any cell north of it in the column is marked as a vertical fault or any cell west of it in the row is marked as a horizontal fault. To determine whether any cell to the west of a cell in the cell's row is a horizontal fault, a cell must know whether any cell to its west and south is faulty. To know what logical state a cell should take during reconfiguration, whether or not it undergoes horizontal and/or vertical reconfiguration must be known. In addition, to determine what logical state it should take, the cell must know whether the cell to its west undergoes vertical reconfiguration and thus may need to know its northwest neighbor's logical state. So, a cell needs to know its west neighbor's logical state, its north neighbor's logical state and its northwest neighbor's logical state.

Since the TAM requires that the same information to be sent to each neighbor, all of the above information must be included in the cell's state. Also, since the connections must be consistent, each active cell must have the same neighborhood connections as each other active cell.

LEVEL 2

2-1 A cell's state changes when a fault occurs, when vertical renaming takes place, or when horizontal renaming takes place. Each cell must know whether any cell in its neighborhood is faulty. It should send its logical state to its south neighbor, east neighbor, and southeast neighbor whenever it changes. Similarly, each cell needs to know its north neighbor's, its west neighbor's and its northwest neighbor's logical states.

2-2 The following pseudocode describes the next state function. It closely corresponds with the system hardware. Each state is composed of two parts: the self-test result (e) and the logical state (ls).

```
/* determine_yin */
for (each active cell)
    yin = FALSE
    if (any neighbor below cell in same column is faulty)
        then yin = TRUE
/* determine_y0 */
for (each active cell)
    y0 = FALSE
    if (any neighbor in same column is faulty)
        then y0 = TRUE
/* determine_ex */
for (each active cell)
    ex = FALSE
    for (each neighbor in same row west of the cell itself and the
    cell itself)
        if (yin = TRUE and cell is faulty)
         then ex = TRUE
/* determine_ey */
for (each active cell)
    if (y0 = TRUE and yin = FALSE)
        then ey = TRUE
```

```
        else ey = FALSE
    /* h_recon */
    for (each active cell)
        if (ex = 0)
            then next_state = present_state
        else
            next_state.e = present_state.e
            next_state.ls = present_state.ls of west neighbor
    /* v_recon */
    for (each active cell)
        if (ey = FALSE)
            then next_state = present_state
        else
            next_state.e = present_state.e
            next_state.ls = present_state.ls of north neighbor
```

<u>LEVEL 3</u>

A Level 3 model was developed for the Direct Reconfiguration Algorithm. This model is described in detail in Appendix B. Simulations indicate that the algorithm performs as expected.

### 3.3.2 Using the TAM to Model the Distributed Recovery Strategy

The Distributed Recovery Strategy [Yan86] is described in detail in Chapter 2. This section describes the development of the Tessellation Automata Model for the Distributed Recovery Strategy.

<u>Level 1</u>

1-1 The facility graph representing the redundant hardware system is the redundant graph $G_r$ in the Distributed Recovery Strategy.

1-2 The set of active cells is the set of nodes in the graph ($G_r$) found in Step 1-1.

1-3 Add a single boundary cell, *b*, since all boundary conditions are the same. Add edges from active nodes to *b* so that the degree of each active node is the same.

1-4 The information passed from one node to each neighbor is the node's logical state and the node's neighbors' logical states. Also, takeover signals are needed to inform spares what state they should take and when. Each cell must know this information to determine its next state. Thus, this information is the state information for each cell in the TAM.

Level 2

2-1 A state changes when a local supervisor takes a new logical state, a spare takes a new logical state, a cell's neighbor's logical state changes, or a local supervisor sends a takeover signal to a spare.

2-2 For the next state function's pseudocode, the following notation is used to the state information.

> a[i,j] is the jth adjacency for the cell with the ith state
> e[i,j] is the jth response assignment for the cell with the ith state
> o[j] is a message to the jth neighbor

Each cell's state s is composed of the following information:

> trans[j] is the neighbor corresponding to the jth input
> a indicates the cell's logical state
> s[j] is the state of the jth neighbor

Pseudocode for the next state function:

```
CASE (s.a)
   COMPUTATION STATE:
      for (i = 1 to NUM_NEIGHBORS)
```

```
o[i] = NO_MSG
endfor
i=1
while ((i ≤ MAX_RESPONSE_ASSIGNMENTS) and (E[a,i] ≠
INVALID))
   error = TRUE
   j = 1
   while ((j ≤ NUM_NEIGHBORS) and (error = TRUE))
      if (s[j].a = E[a,i]) then
         error = FALSE
      endif
      j = j + 1
   endwhile
   if (error = TRUE) then
      look_for_spare = TRUE
      j = 1
      while ((j ≤ NUM_NEIGHBORS) and (look_for_spare = TRUE))
         if (s[j].a = SPARE) then
            spare_found = TRUE
            k = 1
            while ((k ≤ MAX_ADJACENCIES) and (spare_found =
            TRUE) and (A[j,k] ≠ INVALID))
               l = 1
               adj_found = FALSE
               while ((l ≤ NUM_NEIGHBORS) and
                     (adj_found = FALSE))
                  if (s[j].n[l] = A[j,k]) then
                     adj_found = TRUE
                  endif
               l = l + 1
               endwhile
               if (adj_found = FALSE) then
                  spare_found = FALSE
               endif
               k = k + 1
            endwhile
            if (spare_found = TRUE) then
               o[j] = E[a,i]
               look_for_spare = FALSE
            endif
         endif
         j = j + 1
      endwhile
```

```
            if (look_for_spare = TRUE) then
               can_take_state = TRUE
               k = 1
               while ((k ≤ MAX_ADJACENCIES) and (can_take_state =
               TRUE) and (A[j,k] ≠ INVALID))
                  l = 1
                  adj_found = FALSE
                  while ((l ≤ NUM_NEIGHBORS) and (adj_found =
                  FALSE))
                     if (s[l].a = A[j,k]) then
                        adj_found = TRUE
                     endif
                     l = l + 1
                  endwhile
                  if (adj_found = FALSE) then
                     can_take_state = FALSE
                  endif
                  k = k + 1
               endwhile
               if (can_take_state = FALSE) then
                  RECONFIGURATION FAILS!
               else
                  a = E[a,i]
               endif
            endif
         endif
      i = i + 1
      endwhile
   endcase

   FAULTY:
      do nothing
   endcase

   SPARE:
      for i = 1 to NUM_NEIGHBORS
         for j = 1 to NUM_NEIGHBORS
            if (s[i].o[trans[j]] ≠ NO_MSG) then
               a = s[i].o[j]
            endif
         endfor
      endfor
   endcase
```

Level 3

A Level 3 model would be developed in a similar manner to that of the Direct Reconfiguration Algorithm (see Appendix B).

### 3.3.3 Using the TAM to Model the LSM Array Algorithm

The Array Reconfiguration Algorithm is described in detail in Chapter 2. This section describes the TAM for this reconfiguration algorithm. This model is very similar to that of the Distributed Recovery Strategy presented in the previous section.

Level 1

1-1 The facility graph representing the redundant system is shown in Figure 41.

1-2 The set of active cells, A, is the set of nodes in the graph shown in Figure 41. (found in Step 1-1).

1-3 Add boundary cells all around the edges of the array. A single boundary cell could be used, but the use of several cells demonstrates how boundary cells can be assigned to make the boundary conditions more clear. The graph showing boundary conditions is given in Figure 42.

1-4 A cell's logical state should be passed to all its neighbors in the redundant system's facility graph. A cell's neighbors' logical states must also be passed to each neighbor. Some indicator must be given to spare cell to tell that spare to take on a specified state. To determine its next state, a cell must have information telling it which error conditions to

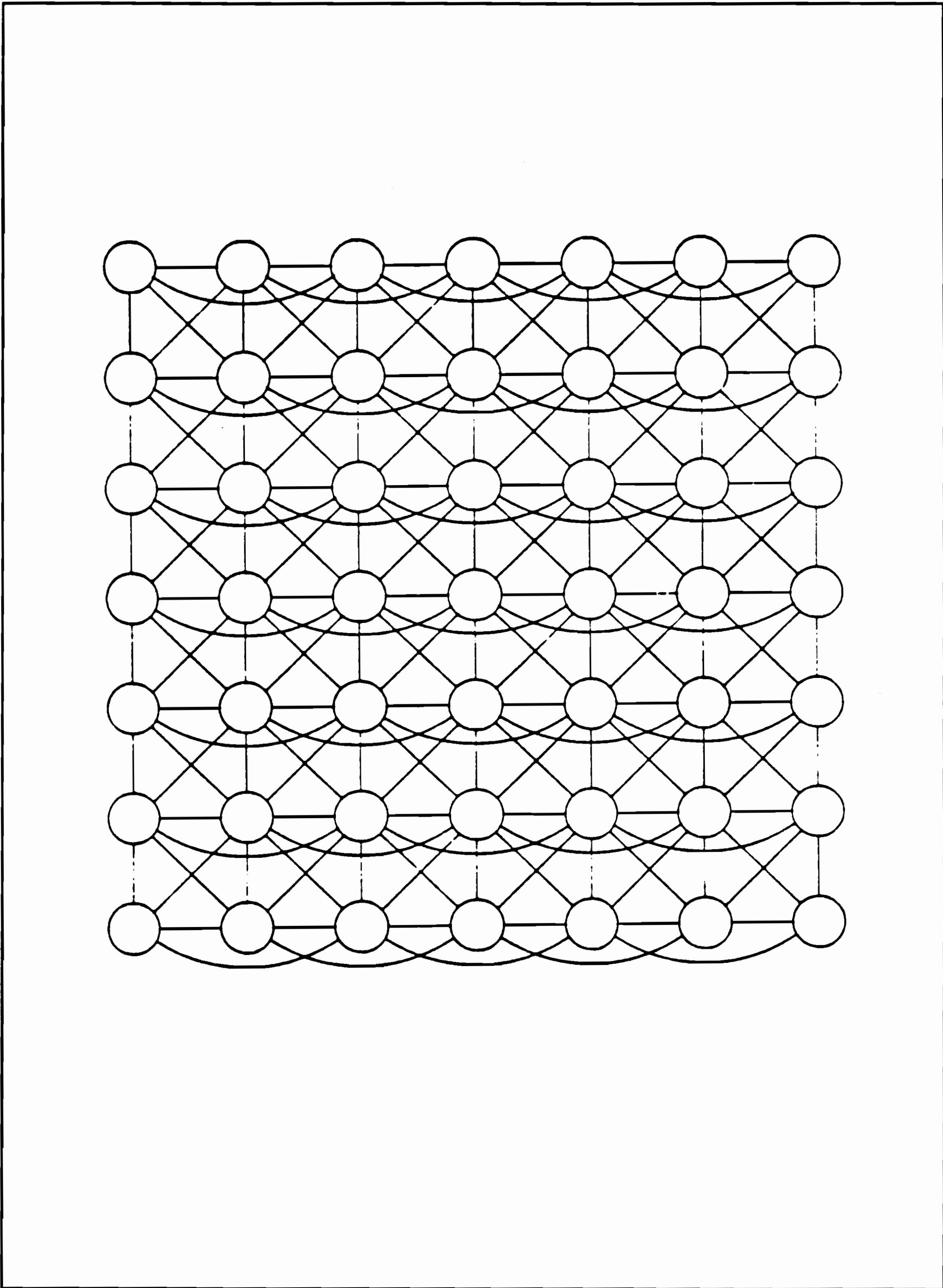


**Figure 41.** Facility Graph for the Array Reconfiguration Algorithm

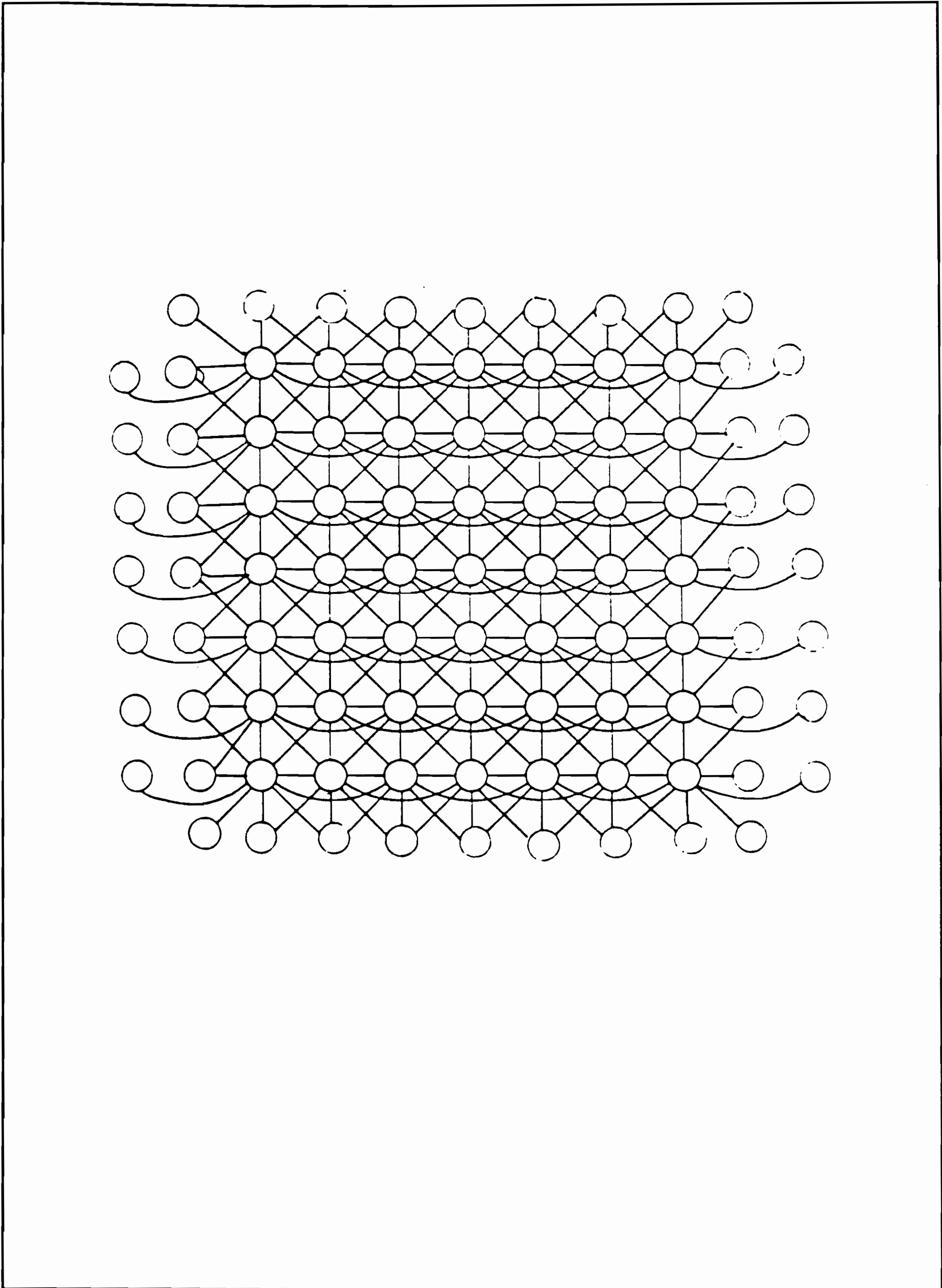


**Figure 42.** Graph for TAM of Array Reconfiguration Algorithm

respond to and the required adjacencies for those error states.

Level 2

2-1 At each time step, a cell's neighbors need to know its logical state and its neighbors' logical states. When a spare cell needs to take on a new state, the responding node must send it an appropriate message.

2-2 Pseudocode for the next state function is given below.

```
CASE (cell's activity indicator)
    FAULTY:
        do nothing

    NORMAL COMPUTATION STATE:
        if (no current values for neighbors states) then
            get current values from neighbors
        endif
        if (for each error response assignment, some cell
                in the neighborhood of the cell has that computation
                state)
            then do nothing
        else (error condition exists)
            if (no spare in neighborhood)
                if (cell can takeover) then takeover
                    error state
                else (cell can't takeover) then
                    RECONFIGURATION FAILS
                endif
            else (spare in neighborhood)
                if (spare has connections to takeover)
                    then tell spare to takeover
                else (spare can't takeover)
                    if (cell can takeover) then
                        takeover error state
                    else (cell can't takeover)
                        RECONFIGURATION FAILS
                    endif
                endif
            endif
        endif
```

```
SPARE:
    if (input from cell that says takeover state) then takeover state
    else (no takeover input)
        do nothing
    endif
END CASE
```

Level 3

A Level 3 model would be developed using the same procedure as used in developing the Level 3 model for the Direct Reconfiguration Algorithm (see Appendix B).

## 3.4 Summary

The Tessellation Automata Model (TAM) overcomes many of the shortcomings of the Local Supervisor Model. The Tessellation Automata Model relies on tessellation automata in which a cell's next state is based on its state and its neighbors' states. The tessellation automaton was chosen as a basis for this model, because it is ideally suited to the description of distributed systems.

The Local Supervisor Model can only be used to describe the behavior of an algorithm. It is not well-suited to describing and analyzing the implementation details. The TAM overcomes this difficulty by providing a flexible format which can be used to describe the implementation details of an algorithm, as well as the behavior of the algorithm. In addition, unlike the LSM, the Tessellation Automata Model does not restrict the algorithm to only

single faults. The Tessellation Automata Model can be used to describe algorithms which can handle multiple and near-coincident faults.

The TAM is a formal structure, but it also has an associated hierarchical description process. Three levels of Tessellation Automata Models are defined. Level 1 is a high level model which can be used to quickly identify important characteristics of an algorithm. Level 2 describes all essential aspects of the algorithm. And, Level 3 is a simulation model which allows the behavior of the system to be simulated in detail. Thus, in describing an algorithm, models only incorporate as many details as necessary.

The Tessellation Automata Model can be used to describe algorithms and approaches. Its use in describing the Direct Reconfiguration Algorithm [Sam86], the Distributed Recovery Strategy [Yan86], and the Array Reconfiguration Algorithm presented in Chapter 2 is discussed. The TAM identifies the substance of each algorithm. Thus, limitations, as well as advantages, can be easily identified. Since it is otherwise difficult to quickly identify limitations of algorithms developed by other designers, the TAM is a very useful tool.

The major limitation of the Tessellation Automata Model is that the information sent to cell's neighbors each time step is its state. When an algorithm sends different information to each neighbor, incorporating all the possible cases into the state information becomes cumbersome. This limitation is overcome by the Interconnected Finite State Machine Model described in the

following chapter.

# 4.0 Interconnected Finite State Machine Model

The TAM is a good model when the same information is communicated to each neighbor. However, when different neighbors are sent different information, the TAM is awkward. To use the TAM to model a system in which different neighbors are sent different information, some part of the state must be used to associate information with a particular neighbor. This problem is illustrated in the TAM for the Distributed Recovery Strategy when a "take over" signal is sent to a spare. The TAM not only is awkward in this respect, but also hides the state interconnection communication information in the state information so system analysis is more difficult.

This problem is overcome by using a model based on interconnected finite state machines. Each cell is represented by a finite state machine which has an output and an input associated with each of its neighbors. Thus, different information can be sent to each neighbor.

In this chapter, the Interconnected Finite State Machine Model is described in general terms. Then, a formal definition is stated. Next, the use

of the Interconnected Finite State Machine Model for describing algorithms is illustrated for several reconfiguration algorithms.

## 4.1 Description of the Interconnected Finite State Machine Model (IFSMM)

The Interconnected Finite State Machine Model (IFSMM) is an interconnection of Moore machines where the states of the Moore machines can be partitioned into two classes: one class includes the states of the model, the second class includes the outputs to the neighbors. Since the IFSMM has a next output function (instead of a present output function), the output information can be considered part of the state of the Moore machine. Thus, the IFSMM is just a specialized version of a Moore machine. This relationship is shown in Figure 43. For convenience, the next output function will also be referred to as the output function. The concept of the Interconnected Finite State Machine is similar to that of Communicating Finite Automata [Cas90] [Bra83].

The IFSMM development process, like the TAM development process, is hierarchical and iterative in nature. Different level models are developed to approach different problems. As in the TAM process, most situations will not require the development of detailed models. The levels of the IFSMM hierarchical modeling process are the same as for the TAM hierarchical modeling process. Three levels are used. Level 1 is a high level model. Level 2 includes more detail. And, the Level 3 model is very detailed.

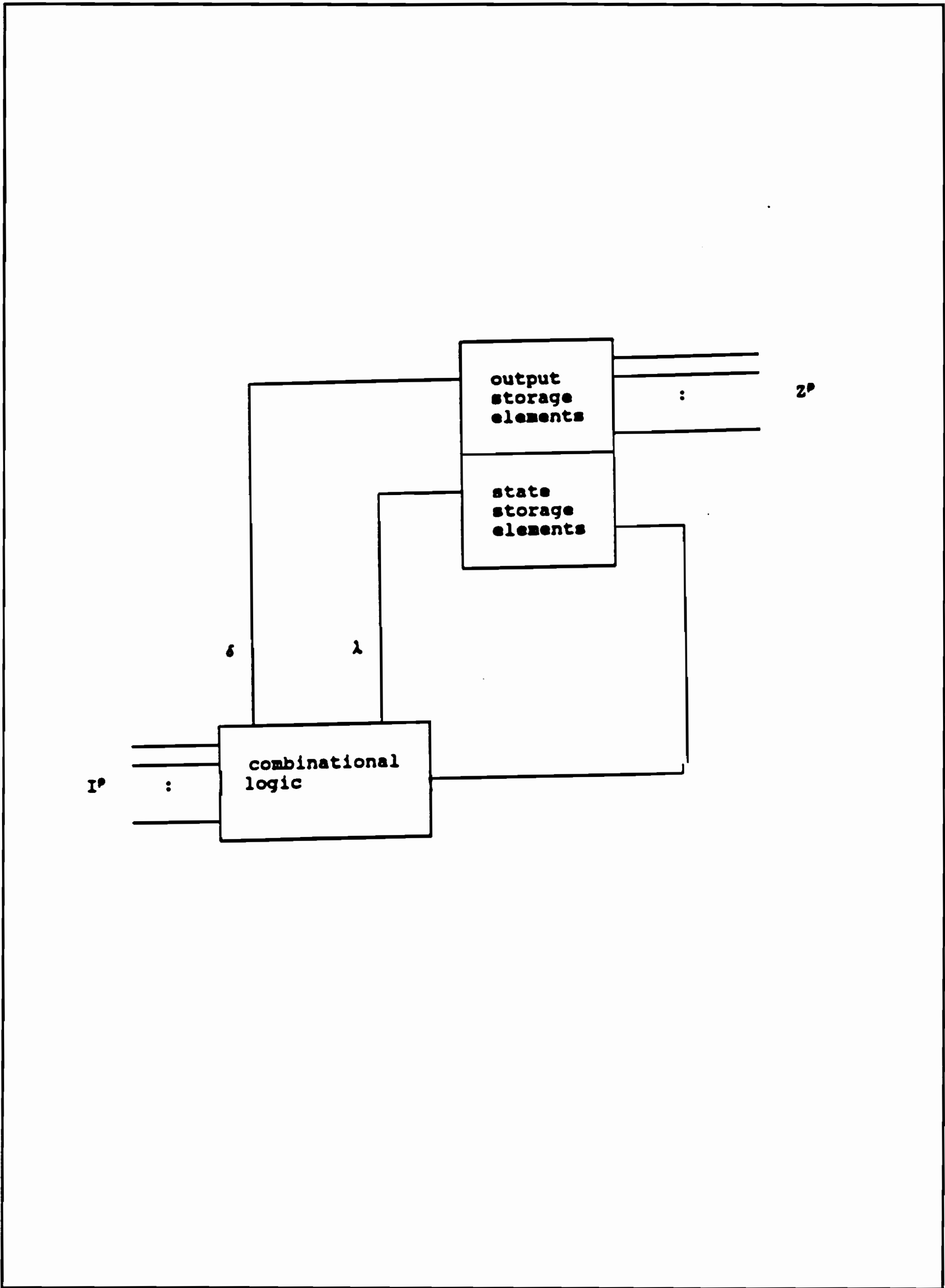



**Figure 43.** The Relationship between IFSMM Cell and the Moore Machine

The heuristic rules for modeling an existing algorithm using the IFSMM are very similar to those for modeling the TAM. However, because of the structural differences of the models, the processes are not identical. The IFSMM modeling process is outlined below.

LEVEL 1

1-1 Develop a facility graph [Hay76] to represent the redundant hardware system.

1-2 The set of active cells, $A$, corresponds to the nodes of the graph. Active cells include spare cells as well as computation cells.

1-3 Add boundary cells so the graph is regular and the neighborhood structure has consistent meaning. These boundary cells are actually just the boundary conditions for the regular architecture. Any nodes which require the same boundary conditions can be connected to the same boundary cell. Consequently, there are no restrictions on the degree of a boundary cell. $B$ is the set of these boundary cells.

1-4 Identify the information that is passed between nodes. This is the set of interconnection values, $I$.

1-5 Determine what information is needed by the cell at each time step to determine the output information. The set of cell states, $Q$, consists of this information.

Notice Steps 1-1 through 1-3 are identical for both the IFSMM and the TAM.

LEVEL 2

2-1 Identify all aspects of the algorithm in which a cell changes its outputs to other cells. This corresponds to the output function, $\lambda$. This is an iterative process. As the model is developed, overlooked information is added to it.

2-2 Identify information that each cell needs to know to determine what the new outputs will be. Don't try to identify **all** such information, just the information that each cell needs. This corresponds to the next state function, $\delta$. If information is needed from other cells, then the information really belongs with the output function, $\lambda$ (see Step 2-1).

2-3 Write pseudocode for the output function, $\lambda$. Remember, only inputs and state information can be used to determine the new outputs. Add any state information missed to the list from Step 2-2.

2-4 Write pseudocode for the next state function, $\delta$. You may need to add state information so you can keep track of what you need to do (e.g. a flag that tells how long you have gone without any inputs).

LEVEL 3

3-1 Using the pseudocode developed for Level 2, write functions which implement the output function, λ, and the next state function, δ using a computer programming language such as "C" or "PASCAL."

3-2 Use the general structure main routine. The main routine and some supporting routines written in C is given in Appendix C.

3-3 Simulate the algorithm and observe its operation. Coverage and time requirements can be determined, if desired.

## 4.2 Formal Definition of the Interconnected Finite State Machine Model

The Interconnected Finite State Machine Model is defined formally as follows:

$$IFSMM = (Q, A, B, I, G, \delta, \lambda, c_0)$$

where

$Q$ is a finite, non-empty set of cell states

$A$ is a set of active cells

$B$ is a set of boundary cells

$I$ is a set of interconnection (input and output) values

$G$ is an $n$-tuple of $p$-tuples of elements of $A \cup B$ where $n = |A|$ and $p$ is the number of neighbors that each active cell has

$\delta: Q \; X \; I^p \rightarrow Q$ is the next state function

$\lambda: Q \; X \; I^p \rightarrow I^p$ is the next output function

$c_i: A \cup B \rightarrow Q \times P$ is the global state at time $i$ (Thus, $c_0$ is the global state at time zero)

The parameter $G$ corresponds to the facility graph for the redundant system which includes the boundary cells. $G$ represents the interconnection structure of the nodes in the system by defining the neighbors of each node. Since there are $n$ active nodes in the system, and each node has $p$ neighbors, then $G$ is defined as an $n$-tuple of $p$-tuples. Active nodes can have both active and boundary cells as neighbors, so members of each $p$-tuple must be elements of $A \cup B$.

The formal definition of the Interconnected Finite State Machine Model will be illustrated with a simple example. Figure 44 shows a linear array with 4 processors. In this example, one of the processors is a spare, and each of the other 3 processors performs some computation identified by its respective computation state. Redundant connections are provided to allow reconfiguration. Information flows from west to east through states 1, 2, and 3. When a fault occurs, each cell to the east of the fault takes on its west neighbor's state. This simple algorithm assumes that each cell knows whether or not each of its neighbors is faulty. The modeling process described in the previous section will be used to develop the formal model for this system.

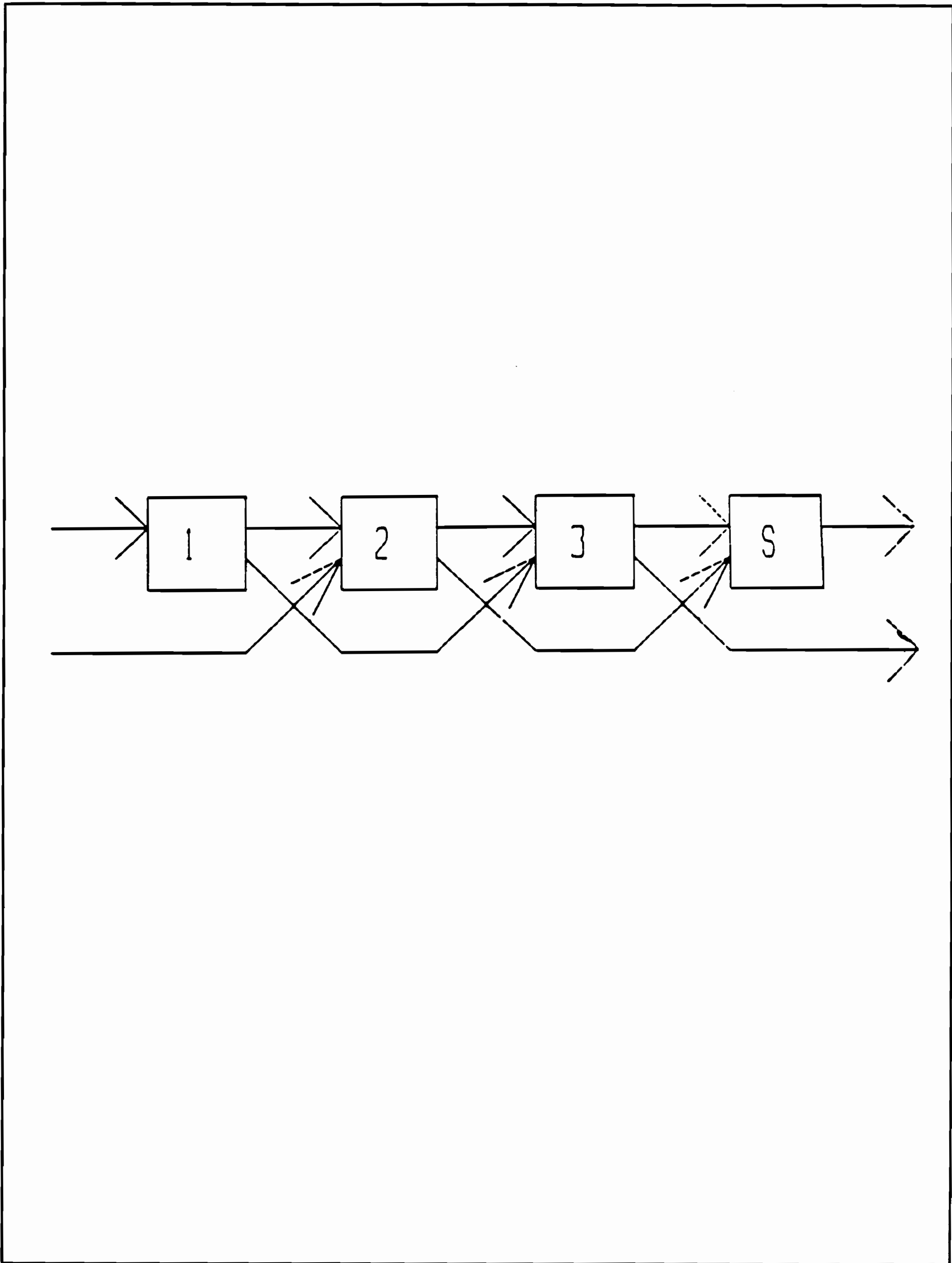



**Figure 44.** A Redundant Linear Array Block Diagram

LEVEL 1

1-1 Figure 45 shows the facility graph representing the redundant system. The facility graph has 4 nodes (one for each cell in the linear array). Redundant connections are shown between these nodes.

1-2 *A*, the set of active cells, consists of the nodes of the facility graph (See Figure 45).

1-3 To make the facility graph regular with consistent neighborhood connections, boundary cells are added as shown in Figure 46. The degree of each boundary cell node will not generally be the same as the degree of the nodes corresponding to the nodes in the original facility graph. With the addition of the boundary cells, the neighborhood structure is consistent. Each cell has four neighbors: far east, east, west, and far west. *B* is the set of these boundary cells. There is more than one way to add boundary cells. For example, boundary cells could be added as shown in Figure 47 instead.

1-4 The information passed between nodes, *I*, must be identified. To determine whether the system needs to be reconfigured, and if so how to reconfigure the system, each cell needs to know whether a fault has occurred to its west and what state it should take. In addition, each cell should know with which of its neighbors it should communicate during the computations (after reconfiguration is complete).

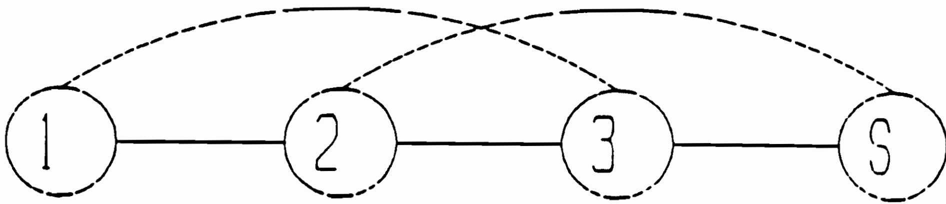



**Figure 45.** Facility Graph for the Redundant Linear Array

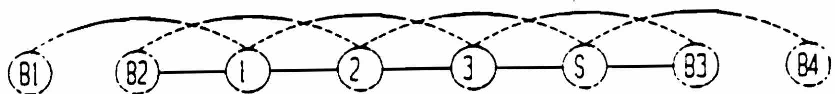



**Figure 46.** Redundant graph with Boundary Cells

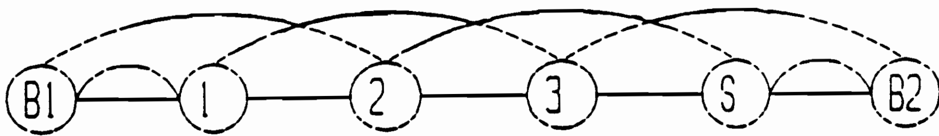



**Figure 47.** Alternative Way to Add Boundary Cells

1-5 To determine the output information, each cell needs to know its state and whether there is known to be a fault to the west.

Thus, the Level 1 model is defined.

Using the notation of the formal model. The set of active cells is defined by

$$A = \{1, 2, 3, S\}$$

where the states of the cells are used as the names of the vertices as well as the labels of the vertices for simplicity. The set of boundary cells is defined by

$$B = \{B1, B2, B3, B4\}$$

If an alternate choice of boundary cells was made, the set of boundary cells would necessarily be different.

The graph $G$ should also be defined using the notation of the model. The vertices of the graph are the active and boundary cells. $G$ is an $n$-tuple of $p$-tuples where $n = |A| = 4$ and $p$ is the number of neighbors that each active cell has. Thus, $p = 4$. The graph $G$ corresponds to the facility graph shown in Figure 46. So, $G$ is

$$( (B1, B2, 2, 3), (B2, 1, 3, S), (1, 2, S, B3), (2, 3, B3, B4) )$$

The first element of the n-tuple ($4$-tuple) corresponds to the node with state $1$. The second element corresponds to the node with state $2$, and the third node corresponds to the node with state $3$. The last element of the $4$-tuple corresponds to the node with the spare state. The elements of each $p$-tuple ($4$-tuple) correspond to the far west, west, east, and far east neighbors

respectively. Thus, for the first $p$-tuple which corresponds to the node with state *1*, *B1* is the far west neighbor, *B2* is the west neighbor, *2* is the east neighbor, and *3* is the far east neighbor.

A translated graph must be developed to use with the simulation program. In this form of the graph, each node name is translated to an integer value between 0 and n-1. The chosen translation for this graph is

$$((4, 5, 1, 2), (5, 0, 2, 3), (0, 1, 3, 6), (1, 2, 6, 7))$$

Other translations could be used. However, for convenience the standard routines assume that active cells are assigned to the lowest possible integer values and the boundary cells are assigned to the higher values.

The set of interconnection values, $I$, can be formally defined. The interconnection information must include a fault indicator which is true if a fault is known to have occurred to the west. This fault indicator will be denoted by $F$. The fault indicator is sent to a cell's east and far east neighbors. A cell must also send its state to its east neighbor. This information will be indicated by $s$. Some indication is also needed to identify active connections. The cells through which normal communication should take place can be specified by a 4-tuple $(a_{FE}, a_E, a_W, a_{FW})$ where $a_i$ is true if the connection is active. Thus, the preliminary definition of the set of interconnection values is

$$I_{preliminary} = \{(F, s, (a_{FE}, a_E, a_W, a_{FW}))\}$$

Thus, the set of interconnection values is defined in a straightforward manner.

The set of cell states, $Q$, has also been determined by the Level 1 modeling procedure. To determine the value of the fault indicator, $F$, that a cell sends to its east and far east neighbors, a cell must know whether its west neighbor or far west neighbor thinks that a fault has occurred to its west. This information is combined into a single fault indicator value which is kept as part of the state information. In addition, the cell must know whether its west or far west neighbors are faulty, since faulty cells cannot be trusted to provide correct information ($t_W$ and $t_{FW}$). Define $t_W$ to be true when the west neighbor is fault and false when the west neighbor is not faulty. $t_W$ is defined similarly. The set of cell states must also include the computation state of the cell itself. So, the preliminary definition of the set of cell states is defined by

$$Q_{preliminary} = \{(F, s_{self}, t_W, t_{FW})\}$$

Thus, we have completed a preliminary definition of the cell states.

After completing the Level 1 modeling process, we still do not have a good definition of the next state function, $\delta$, and the next output function, $\lambda$. These functions will be more fully defined in the Level 2 model. The Level 1 model can be used to make high level compare parameters such as storage and communication requirements.

The global state at time $0$ consists of the initial conditions for each cell. If we assume that at time $0$ no cells in the system are faulty, the global state at time $0$ can be defined as follows:

$$c_0(1) = (q(1), i_{far\ east}(1), i_{east}(1), i_{west}(1), i_{far\ west}(1))$$

$$c_0(2) = (q(2), i_{far\ east}(2), i_{east}(2), i_{west}(2), i_{far\ west}(2))$$
$$c_0(3) = (q(3), i_{far\ east}(3), i_{east}(3), i_{west}(3), i_{far\ west}(3))$$
$$c_0(S) = (q(S), i_{far\ east}(S), i_{east}(S), i_{west}(S), i_{far\ west}(S))$$

and

$$c_0(B1) = (q(B1), i_{far\ east}(B1), i_{east}(B1), i_{west}(B1), i_{far\ west}(B1))$$
$$c_0(B2) = (q(B2), i_{far\ east}(B2), i_{east}(B2), i_{west}(B2), i_{far\ west}(B2))$$
$$c_0(B3) = (q(B3), i_{far\ east}(B3), i_{east}(B3), i_{west}(B3), i_{far\ west}(B3))$$
$$c_0(B4) = (q(B4), i_{far\ east}(B4), i_{east}(B4), i_{west}(B4), i_{far\ west}(B4))$$

where

$$q(1) = (0, 1, 0, 0)$$
$$q(2) = (0, 2, 0, 0)$$
$$q(3) = (0, 3, 0, 0)$$
$$q(S) = (0, S, 0, 0)$$
$$q(B1) = (0, B1, 0, 0)$$
$$q(B2) = (0, B2, 0, 0)$$
$$q(B3) = (0, B3, 0, 0)$$
$$q(B4) = (0, B4, 0, 0)$$

$$i_{far\ east}(1) = (0, N, (0, 1, 1, 0))$$
$$i_{far\ east}(2) = (0, N, (0, 1, 1, 0))$$
$$i_{far\ east}(3) = (0, N, (0, 1, 1, 0))$$
$$i_{far\ east}(S) = (0, N, (0, 1, 1, 0))$$
$$i_{far\ east}(B1) = (0, N, (0, 1, 1, 0))$$
$$i_{far\ east}(B2) = (0, N, (0, 1, 1, 0))$$
$$i_{far\ east}(B3) = (0, N, (0, 1, 1, 0))$$
$$i_{far\ east}(B4) = (0, N, (0, 1, 1, 0))$$

$$i_{east}(1) = (0, 1, (0, 1, 1, 0))$$
$$i_{east}(2) = (0, 2, (0, 1, 1, 0))$$
$$i_{east}(3) = (0, 3, (0, 1, 1, 0))$$
$$i_{east}(S) = (0, S, (0, 1, 1, 0))$$
$$i_{east}(B1) = (0, B1, (0, 1, 1, 0))$$
$$i_{east}(B2) = (0, B2, (0, 1, 1, 0))$$
$$i_{east}(B3) = (0, B3, (0, 1, 1, 0))$$
$$i_{east}(B4) = (0, B4, (0, 1, 1, 0))$$

$$i_{west}(1) = (N, N, (0, 1, 1, 0))$$
$$i_{west}(2) = (N, N, (0, 1, 1, 0))$$
$$i_{west}(3) = (N, N, (0, 1, 1, 0))$$

$$i_{west}(S) = (N, N, (0, 1, 1, 0))$$
$$i_{west}(B1) = (N, N, (0, 1, 1, 0))$$
$$i_{west}(B2) = (N, N, (0, 1, 1, 0))$$
$$i_{west}(B3) = (N, N, (0, 1, 1, 0))$$
$$i_{west}(B4) = (N, N, (0, 1, 1, 0))$$

$$i_{far\ west}(1) = (N, N, (0, 1, 1, 0))$$
$$i_{far\ west}(2) = (N, N, (0, 1, 1, 0))$$
$$i_{far\ west}(3) = (N, N, (0, 1, 1, 0))$$
$$i_{far\ west}(S) = (N, N, (0, 1, 1, 0))$$
$$i_{far\ west}(B1) = (N, N, (0, 1, 1, 0))$$
$$i_{far\ west}(B2) = (N, N, (0, 1, 1, 0))$$
$$i_{far\ west}(B3) = (N, N, (0, 1, 1, 0))$$
$$i_{far\ west}(B4) = (N, N, (0, 1, 1, 0))$$

The states of the boundary cells will always be the same. This condition is necessary since the boundary cells do not actually exist. So, their states are hard-wired into the system. For example, a boundary cell will always have a state that is identified as a non-faulty state that is unable to act as a spare.

In developing the Level 2 model, we realized that a 4-tuple was not needed to specify active connections since each communication link only needed to be activated--no information was required about active links to other neighbors. So the set of interconnection values was revised:

$$I = \{(F, s, a)\}$$

$a$ is true if the link is active, otherwise a is false.

While writing the pseudocode for the next output function in the Level 2 model, additional information was identified as needed in the cell states. So, we will add the test results for the east and far east neighbors also to the information contained in the set of cell states. In addition, the state

information for each cell should include the list of active connections for each cell. The active connections are specified by a 4-tuple where $a_i$ is true if there is an active connection to neighbor $i$.

$$Q = \{(F, s_{self}, t_W, t_{FW}, t_E, t_{FE}, (a_{FE}, a_E, a_W, a_{FW}))\}$$

The letter 'N' will indicate a signal that is not defined (has no value). Using the second definition of $Q$, the global state at time $0$, assuming no faulty cells in the system, is

$$c_0(1) = (q(1), i_{\text{far east}}(1), i_{\text{east}}(1), i_{\text{west}}(1), i_{\text{far west}}(1))$$
$$c_0(2) = (q(2), i_{\text{far east}}(2), i_{\text{east}}(2), i_{\text{west}}(2), i_{\text{far west}}(2))$$
$$c_0(3) = (q(3), i_{\text{far east}}(3), i_{\text{east}}(3), i_{\text{west}}(3), i_{\text{far west}}(3))$$
$$c_0(S) = (q(S), i_{\text{far east}}(S), i_{\text{east}}(S), i_{\text{west}}(S), i_{\text{far west}}(S))$$

and

$$c_0(B1) = (q(B1), i_{\text{far east}}(B1), i_{\text{east}}(B1), i_{\text{west}}(B1), i_{\text{far west}}(B1))$$
$$c_0(B2) = (q(B2), i_{\text{far east}}(B2), i_{\text{east}}(B2), i_{\text{west}}(B2), i_{\text{far west}}(B2))$$
$$c_0(B3) = (q(B3), i_{\text{far east}}(B3), i_{\text{east}}(B3), i_{\text{west}}(B3), i_{\text{far west}}(B3))$$
$$c_0(B4) = (q(B4), i_{\text{far east}}(B4), i_{\text{east}}(B4), i_{\text{west}}(B4), i_{\text{far west}}(B4))$$

where

$$q(1) = ((0, 1, 0, 0, 0, 0, (0, 0, 0, 0))$$
$$q(2) = ((0, 2, 0, 0, 0, 0, (0, 0, 0, 0))$$
$$q(3) = ((0, 3, 0, 0, 0, 0, (0, 0, 0, 0))$$
$$q(4) = ((0, 4, 0, 0, 0, 0, (0, 0, 0, 0))$$
$$q(B1) = ((0, B1, 0, 0, 0, 0, (0, 0, 0, 0))$$
$$q(B2) = ((0, B2, 0, 0, 0, 0, (0, 0, 0, 0))$$
$$q(B3) = ((0, B3, 0, 0, 0, 0, (0, 0, 0, 0))$$
$$q(B4) = ((0, B4, 0, 0, 0, 0, (0, 0, 0, 0))$$

$$i_{\text{far east}}(1) = (0, N, 0)$$
$$i_{\text{far east}}(2) = (0, N, 0)$$
$$i_{\text{far east}}(3) = (0, N, 0)$$
$$i_{\text{far east}}(S) = (0, N, 0)$$
$$i_{\text{far east}}(B1) = (0, N, 0)$$
$$i_{\text{far east}}(B2) = (0, N, 0)$$

$$i_{far\ east}(B3) = (0, N, 0)$$
$$i_{far\ east}(B4) = (0, N, 0)$$

$$i_{east}(1) = (0, 1, 1)$$
$$i_{east}(2) = (0, 2, 1)$$
$$i_{east}(3) = (0, 3, 1)$$
$$i_{east}(S) = (0, S, 1)$$
$$i_{east}(B1) = (0, B1, 1)$$
$$i_{east}(B2) = (0, B2, 1)$$
$$i_{east}(B3) = (0, B3, 1)$$
$$i_{east}(B4) = (0, B4, 1)$$

$$i_{west}(1) = (N, N, 1)$$
$$i_{west}(2) = (N, N, 1)$$
$$i_{west}(3) = (N, N, 1)$$
$$i_{west}(S) = (N, N, 1)$$
$$i_{west}(B1) = (N, N, 1)$$
$$i_{west}(B2) = (N, N, 1)$$
$$i_{west}(B3) = (N, N, 1)$$
$$i_{west}(B4) = (N, N, 1)$$

$$i_{far\ west}(1) = (N, N, 0)$$
$$i_{far\ west}(2) = (N, N, 0)$$
$$i_{far\ west}(3) = (N, N, 0)$$
$$i_{far\ west}(S) = (N, N, 0)$$
$$i_{far\ west}(B1) = (N, N, 0)$$
$$i_{far\ west}(B2) = (N, N, 0)$$
$$i_{far\ west}(B3) = (N, N, 0)$$
$$i_{far\ west}(B4) = (N, N, 0)$$

LEVEL 2

2-1 In this step, all aspects of the algorithm in which a cell changes its outputs to other cells is identified. This step is related to the definition of the next output function, λ. Outputs are changed when a cell first learns that there is a fault to its west. Outputs also change when the computation state of the cell changes.

2-2 In this step, information that each cell needs to know to determine the new outputs are identified. This step is related to the development of the next state function, $\delta$. To determine its new outputs a cell must know the fault indicator for its west and far west neighbors, the computation state of its west neighbor, and the test results for its neighbors.

2-3 The output for each neighbor is identified by a subscript. Specifically, the outputs to the far east, east, west, and far west neighbors are denoted by $o_{FE}$, $o_E$, $o_W$, and $o_{FW}$, respectively. The various parts of each output are identified using a format similar to the C programming language structure notation. The state information of the cell is referred to by the notation "state" using the same format for indicating parts of the state. The results of testing the east and far east neighbors are denoted by $t_E$ and $t_{FE}$. A test result is true if the test fails. The test results are not kept explicitly as part of the state information. Pseudocode for the next output function, $\lambda$ is written as follows:

```
o_W     =     no output
o_FW    =     no output

if (t_E = FALSE) then
        o_E.F   =     state.F
        o_E.s   =     state.s_self
        o_E.a   =     state.a_E
else
        o_E     =     no output
endif
```

```
if ((t_FE = FALSE) & (t_E = TRUE)) then
        o_FE.F  =       state.F
        o_FE.s  =       state.s_self
        o_FE.a  =       state.a_FE
else
        o_FE    =       no output
endif
```

2-4 The input from each neighbor is identified by a subscript. As in Step 2-3, the parts of the inputs and the state are identified using a format similar to the C programming language structure notation. Pseudocode for the next state function, $\delta$ follows:

```
if ((in_FW.F ∧ t_FW′) ∨ (in_W.F ∧ t_W′) ∨ state.F) then
        state.F         =       True;
endif

if (t_W)
        if (t_FW′)
                state.a_FW      =       TRUE
                state.a_W       =       FALSE
        else
                Fatal Failure!
        endif
else
        state.a_FW      =       FALSE
        state.a_W       =       TRUE
endif

if (t_E)
        if (t_FE′)
                state.a_FE      =       TRUE
                state.a_E       =       FALSE
        else
                Fatal Failure!
        endif
else
        state.a_FE      =       FALSE
        state.a_E       =       TRUE
endif
```

LEVEL 3

3-1 Using the pseudocode from the Level 2 model and the general C routines described in Appendix C, C routines were developed for the next state function and the next output function. In addition, routines were developed for the supporting functions. To simplify the development process, an existing Level 3 IFSMM was used as a starting point for the development of the necessary routines. The existing routines were edited to reflect the specific characteristics of the example system. This procedure eliminated mistakes in interface definitions. This example could be used as a template for new model development. Some slight changes were made to the structure of the next state function, and a reconfiguration complete flag was added to the state information. These types of changes are not unusual in the development of Level 3 models. The changes are indicated in the modified pseudocode:

```
if (not s.recon_complete)
        if ((inFW.F ∧ tFW') ∨ (inW.F ∧ tW') ∨ state.tFW ∨ state.tW ∨
        state.F) then
                state.F = True;
        else
                state.F = FALSE;
        endif
```

```
        if (state.t_W = TRUE)
              if (state.t_FW = FALSE)
                    state.a_FW    =    TRUE
                    state.a_W     =    FALSE
              else
                    Fatal Failure!
              endif
        else
              state.a_FW    =    FALSE
              state.a_W     =    TRUE
        endif

  if (in_E.s = FAULT)
        if (in_FE.s ≠ FAULT)
              state.a_FE    =    TRUE
              state.a_E     =    FALSE
        else
              Fatal Failure!
        endif
  else
        state.a_FE    =    FALSE
        state.a_E     =    TRUE
  endif

  if (state.f = TRUE)
        if (in_W.s = NO_VALUE)
              state.s = in_W.s
              state.recon_complete = TRUE
        endif
  endif
```

The C routines are contained in the file "exsim.c". The listing of this file follows:

**exsim.c:**

```
#include <stdio.h>
#include "mystd.h"
#include "model.h"

extern FILE *ofile;
extern FILE *sfile;

/**************************************************************************/
/*  TITLE:         apply_lambda                                           */
/*  PURPOSE:       This procedure determines the output of a cell         */
```

```
/*                  based on its inputs and present state.                  */
/*  PARAMETERS:     STATE *present_state                                    */
/*                  INPUT in[NUM_INPUTS]                                    */
/*                  OUTPUT out[NUM_OUTPUTS]                                 */
/*  RETURN:         none                                                    */
/*  REQUIRES:       none                                                    */
/****************************************************************************/

void apply_lambda (STATE present_state, INPUT in[NUM_INPUTS],
      OUTPUT out[NUM_OUTPUTS])

{
if (!present_state.recon_complete)
   {
   out[WEST].f = NO_VALUE;
   out[WEST].s = NO_VALUE;
   out[WEST].a = present_state.aw;

   out[FARWEST].f = NO_VALUE;
   out[FARWEST].s = NO_VALUE;
   out[FARWEST].a = present_state.afw;

   if (present_state.te == FALSE)
      {
      out[EAST].f = present_state.f;
      out[EAST].s = present_state.s;
      out[EAST].a = present_state.ae;
      }
   else
      {
      out[EAST].f = FALSE;
      out[EAST].s = NO_VALUE;
      out[EAST].a = present_state.ae;
      }

   if ((present_state.tfe == FALSE) && (present_state.te == TRUE))
      {
      out[FAREAST].f = present_state.f;
      out[FAREAST].s = present_state.s;
      out[FAREAST].a = present_state.afe;
      }
   else
      {
      out[FAREAST].f = FALSE;
      out[FAREAST].s = NO_VALUE;
      out[FAREAST].a = present_state.afe;
      }
   }
return;
}

/****************************************************************************/
/*  TITLE:          apply_delta                                             */
/*  PURPOSE:        This procedure determines the next state of a cell      */
/*                  based on its inputs and present state.                  */
/*  PARAMETERS:     STATE *present_state                                    */
/*                  INPUT in[NUM_INPUTS]                                    */
/*                  STATE *next_state                                       */
/*  RETURN:         none                                                    */
/*  REQUIRES:       none                                                    */
/****************************************************************************/
```

```
void apply_delta (STATE present_state, INPUT in[NUM_INPUTS],
                  STATE *next_state)

{
int test;  /* test function return value */

next_state->s = present_state.s; /* assume no change */

next_state->tw = present_state.tw;
next_state->tfw = present_state.tfw;
next_state->te = present_state.te;
next_state->tfe = present_state.tfe;

next_state->f = present_state.f;

next_state->aw = present_state.aw;
next_state->afw = present_state.afw;
next_state->ae = present_state.ae;
next_state->afe = present_state.afe;
next_state->recon_complete = present_state.recon_complete;

if (!present_state.recon_complete)
   {
   if (((in[FARWEST].f == TRUE) && (present_state.tfw == FALSE)) ||
            ((in[WEST].f == TRUE) && (present_state.tw == FALSE)) ||
            (present_state.tfw == TRUE) || (present_state.tw == TRUE) ||
            (present_state.f == TRUE))
      next_state->f = TRUE;
   else
      next_state->f = FALSE;

   if (present_state.tw == TRUE)
      if (present_state.tfw == FALSE)
         {
         next_state->afw = TRUE;
         next_state->aw = FALSE;
         }
      else
         {
         test = printf ("FATAL FAILURE\n");
         if (test == EOF)  exit (ERR_PRINTF);
         test = fprintf (ofile, "FATAL FAILURE\n");
         if (test == EOF)  exit (ERR_FPRINTF);
         exit (FATAL_FAILURE);
         }
   else
      {
      next_state->afw = FALSE;
      next_state->aw = TRUE;
      }

   if (present_state.te == TRUE)
      if (present_state.tfe == FALSE)
         {
         next_state->afe = TRUE;
         next_state->ae = FALSE;
         }
      else
         {
         test = printf ("FATAL FAILURE\n");
         if (test == EOF)  exit (ERR_PRINTF);
```

```
         test = fprintf (ofile, "FATAL FAILURE\n");
         if (test == EOF)  exit (ERR_FPRINTF);
         exit (FATAL_FAILURE);
         }
   else
      {
      next_state->afe = FALSE;
      next_state->ae = TRUE;
      }

   if (next_state->f == TRUE)
      if (in[WEST].s != NO_VALUE)
         {
         next_state->s = in[WEST].s;
         next_state->recon_complete = TRUE;
         }
      else
         next_state->s = present_state.s;
   }
return;
}

/****************************************************/
/*  TITLE:        get_init_state                    */
/*  PURPOSE:      This routine gets the initial     */
/*                state information from the        */
/*                appropriate file and sets         */
/*                other initial state               */
/*                information.                      */
/*  PARAMETERS:   int num_rcells                    */
/*                STATE present_state[MAX_CELLS]    */
/*                STATE next_state[MAX_CELLS]       */
/*  REQUIRES:     printf      (standard library)    */
/*                get_string                        */
/*                get_integer                       */
/****************************************************/

void get_init_state (int num_rcells, STATE present_state[MAX_CELLS],
      STATE next_state[MAX_CELLS])
{
int cell;                            /* cell index */
int gentc;                           /* translated cell name */
int num_states;                      /* number of states in file */
int neighbor;                        /* neighbor index */

/******************************************************/
/*  Make sure present state file matches the graph */
/*  file.                                            */
/******************************************************/

num_states = get_integer (sfile);

if (num_states != num_rcells)
   {
   printf ("ERROR:  STATE FILE DOESN'T MATCH GRAPH FILE!\n");
   printf ("state file says %i cells--graph file says %i cells\n",
         num_states, num_rcells);
   exit (ERR_INVALID_STATEFILE);
   }

/* set all initial states to NO_VALUE, thus boundary cells will have
```

```
  activity indicators of NO_VALUE */
for (cell = 0; cell < MAX_CELLS; cell++)
   {
   present_state[cell].recon_complete = FALSE;
   present_state[cell].s = NO_VALUE;
   present_state[cell].f = FALSE;
   present_state[cell].aw = TRUE;
   present_state[cell].afw = FALSE;
   present_state[cell].ae = TRUE;
   present_state[cell].afe = FALSE;
   }

/***************************************************/
/*  For each real cell, get activity indicator     */
/*  and other state information from file.         */
/***************************************************/

for (cell = 0; cell < num_rcells; cell++)
   {
   gentc = get_integer (sfile);
   present_state[gentc].s = get_integer (sfile);
   present_state[gentc].tfw = get_integer (sfile);
   present_state[gentc].tw = get_integer (sfile);
   present_state[gentc].te = get_integer (sfile);
   present_state[gentc].tfe = get_integer (sfile);
   }
return;
}

/*************************************************************************/
/*  TITLE:       print_cell_state                                        */
/*  PURPOSE:     This routine outputs a state.                           */
/*  PARAMETERS:  STATE state                                             */
/*  REQUIRES:    printf                 (standard library)               */
/*               fprintf                (standard library)               */
/*************************************************************************/

void print_cell_state(STATE state)

{
int test;  /* test function return value */

if (state.aw)
   {
   test = printf ("-");
   if (test == EOF)  exit (ERR_PRINTF);
   test = fprintf (ofile, "-");
   if (test == EOF)  exit (ERR_FPRINTF);
   }
if (state.afw)
   {
   test = printf ("\\");
   if (test == EOF)  exit (ERR_PRINTF);
   test = fprintf (ofile, "\\");
   if (test == EOF)  exit (ERR_FPRINTF);
   }
test = printf ("%d", state.s);
if (test == EOF)  exit (ERR_PRINTF);
test = fprintf (ofile, "%d", state.s);
if (test == EOF)  exit (ERR_FPRINTF);
if (state.ae)
```

```
    {
    test = printf ("-");
    if (test == EOF)  exit (ERR_PRINTF);
    test = fprintf (ofile, "-");
    if (test == EOF)  exit (ERR_FPRINTF);
    }
if (state.afe)
    {
    test = printf ("/");
    if (test == EOF)  exit (ERR_PRINTF);
    test = fprintf (ofile, "/");
    if (test == EOF)  exit (ERR_FPRINTF);
    }
return;
}

/**********************************************************************/
/*  TITLE:        copy_interconnection                               */
/*  PURPOSE:      This routine copies the cell interconnection       */
/*                information.  This routine is different for        */
/*                modeling different algorithms.                     */
/*  PARAMETERS:   STATE to_state                                     */
/*                STATE from_state                                   */
/*  REQUIRES:     printf                  (standard library)         */
/*                fprintf                 (standard library)         */
/**********************************************************************/

void copy_interconnection (INPUT *in, OUTPUT *out)

{
in->f = out->f;
in->s = out->s;
in->a = out->a;
return;
}


/**********************************************************************/
/*  TITLE:        initialize_inputs                                  */
/*  PURPOSE:      This routine sets the initial values for the inputs */
/*                and outputs for the graph.  This routine is        */
/*                different for modeling different algorithms.       */
/*  PARAMETERS:   INPUT in[MAX_CELLS][NUM_INPUTS]                    */
/*  REQUIRES:     none                                               */
/**********************************************************************/

void initialize_inputs (INPUT in[MAX_CELLS][NUM_INPUTS])
{
int cell;    /* cell index */
int index;   /* loop index */
int test;    /* test function return value */

for (cell = 0; cell < MAX_CELLS; cell++)
    {
    for (index = 0; index < NUM_INPUTS; index++)
        {
        in[cell][index].f = NO_VALUE;
        in[cell][index].s = NO_VALUE;
        in[cell][index].a = NO_VALUE;
        }
    }
```

```
return;
}

/**************************************************************************/
/*  TITLE:          copy_cell_state                                       */
/*  PURPOSE:        This routine copies the cell state.  This routine     */
/*                  is different for modeling different algorithms.       */
/*  PARAMETERS:     STATE to_state                                        */
/*                  STATE from_state                                      */
/*  REQUIRES:       printf                    (standard library)          */
/*                  fprintf                   (standard library)          */
/**************************************************************************/

void copy_cell_state (STATE *to_state, STATE from_state)

{
to_state->f = from_state.f;
to_state->s = from_state.s;
to_state->tw = from_state.tw;
to_state->tfw = from_state.tfw;
to_state->te = from_state.te;
to_state->tfe = from_state.tfe;
to_state->aw = from_state.aw;
to_state->afw = from_state.afw;
to_state->ae = from_state.ae;
to_state->afe = from_state.afe;
to_state->recon_complete = from_state.recon_complete;
return;
}
```

The support routines (get_init_state, print_cell_state, copy_interconnection, initialize_inputs, and copy_cell_state) use the interface specified by the general C routines listed in Appendix C. These routines are quite simple and will not be discussed further. A simple makefile was used for compiling and linking these routines.

**makefile:**

```
#  Example System Simulation

CFLAGS = -c

.c.o:
      cc $(CFLAGS) -o $@ $*.c

ex.out: exsim.o get_tg.o get.o simmain.o
      cc -o ex.out exsim.o get_tg.o get.o simmain.o

exsim.o:    mystd.h model.h mod_proto.h
get_tg.o:   mystd.h model.h mod_proto.h
get.o:      mystd.h mod_proto.h
simmain.o:  mystd.h model.h mod_proto.h
```

The header files "model.h" and "mod_proto.h" were developed to reflect the characteristics of the example system.

**model.h:**

```
#define FARWEST      0
#define WEST         1
#define EAST         2
#define FAREAST      3
#define FATAL_FAILURE -3

#define LINE_LENGTH 81
#define OUTPUTS_PER_LINE 4
#define ERR_INVALID_STATEFILE 2

#define MAX_CELLS 97
#define MAX_STEPS 5

#define NUM_INPUTS 10
#define MAX_NEIGHBORS 10

#define NUM_OUTPUTS 10

#define FAULT -1
#define NO_VALUE -2

typedef int BOOLEAN;

typedef struct state {
   BOOLEAN f;     /* known fault to west */
   int s;         /* activity state for cell itself */
   BOOLEAN tw;    /* test west */
   BOOLEAN tfw;   /* test far west */
   BOOLEAN te;    /* test east */
   BOOLEAN tfe;   /* test far east */
   BOOLEAN aw;    /* active connection west */
   BOOLEAN afw;   /* active connection far west */
   BOOLEAN ae;    /* active connection east */
   BOOLEAN afe;   /* active connection far east */
   BOOLEAN recon_complete; /* reconfiguration complete signal */
} STATE;

typedef struct
   {
   int f;     /* known fault to west */
   int s;     /* activity state for cell itself */
   int a;     /* active connection */
   } INPUT, OUTPUT;

typedef struct
   {
   int node;
   int in;
   } GRAPH_NODE;
```

**mod_proto.h:**

```
void apply_delta (STATE pstate, INPUT in[NUM_INPUTS], STATE *nstate);
```

```
void   apply_lambda   (STATE   pstate,   INPUT   in[NUM_INPUTS],   OUTPUT
out[NUM_OUTPUTS]);
void copy_interconnection (INPUT *in, OUTPUT *out);
void copy_cell_state (STATE *tostate, STATE fromstate);
void print_cell_state (STATE state);
void initialize_inputs (INPUT in[MAX_CELLS][NUM_INPUTS]);
void get_init_state (int num_rcells, STATE present_state[MAX_CELLS],
      STATE next_state[MAX_CELLS]);
void get_tgraph (int *num_rcells, int *num_icells, int *num_neighbors,
      GRAPH_NODE graph[MAX_CELLS][MAX_NEIGHBORS]);
void get_string (FILE *infile, char string[LINE_LENGTH]);
int get_integer (FILE *infile);
```

3-2 The general structure main routines described in Appendix C are used with the routines described in Step 3-1.

3-3 Debugging of this model was fairly easy. Total model development time was approximately 3 hours. Simulations were run confirming that the system reconfigures correctly around all single faults. In addition, a non-faulty system is simulated, and no reconfiguration occurs as expected. The graph file used is very simple:

**tgraf.dat:**

```
4 4 4

4 5 1 2
5 0 2 3
0 1 3 6
1 2 6 7

3 2 1 0
3 2 1 0
3 2 1 0
3 2 1 0
```

The first line indicates there are four active cells, four boundary cells, and four neighbors per cell. Cells 0, 1, 2, and 3 are active cells. Cells 4, 5, 6, and 7 are boundary cells. The next four lines define the neighbors of each cell. A cell's position in the sequence given on a

particular line determines whether it is a far west, west, east, or far east neighbor of the corresponding cell. Consider Cell 0. Its far west neighbor is cell 4, its west neighbor is cell 5, its east neighbor is cell 1, and its far east neighbor is cell 2. The next group of lines defines the interconnections for each cell. In this example, all cells are connected in the same way. Output 0 is connected to input 3 of neighbor 0. In other words, the far west output is connected to the far east input of the cell's far west neighbor. Similarly, output 1 is connected to input 2 of neighbor 1. Thus, the west output is connected to the east input of the west neighbor.

For cell 1 faulty, the following state file is used:

**s2.dat:**

```
4

0 1    0 0 1 0
1 2    0 0 0 0
2 3    0 1 0 0
3 0    1 0 0 0
```

The first line indicates that the states of the four active cells are defined in this file. Each subsequent line defines the number of the cell, its activity indicator, and the results of testing its neighbors. The order for the test results is far west, west, east, and far east. If cell 1 is faulty (the node in state 2), then cell 0 determines that its east neighbor is faulty, cell 2 determines that its west neighbor is faulty, and cell 3 determines that its far west neighbor is faulty.

The simulation was run for four time steps. The output shows each cell's activity indicator and active connections. The symbol '-' is used to indicate that the active connection is to the east or west. The symbol '/' indicates that the active connection is to the far east. Similarly, the symbol '\' is used to indicate an active connection to the far west. The output file obtained from this simulation follows:

**o2.dat:**

```
ex.out tgraf.dat s2.dat o2.dat

GLOBAL STATE at time 0
-1--2--3--0-
*************************************************

GLOBAL STATE at time 1
-1/-2-\3--0-
*************************************************

GLOBAL STATE at time 2
-1/-2-\2--3-
*************************************************

GLOBAL STATE at time 3
-1/-2-\2--3-
*************************************************

GLOBAL STATE at time 4
-1/-2-\2--3-
*************************************************

*******************************************************

*******************************************************
FINAL STATE
-1/-2-\2--3-
```

At time one, cells 0 and 2 change their active connections to bypass the faulty cell. At time two, the cell 2 takes over the activity of its faulty west neighbor and cell 3 takes over the activity of its west neighbor. Thus, reconfiguration is complete.

For single faults, the following times were required for reconfiguration:

| Faulty Cell | Time Required |
|---|---|
| 0 | 3 |
| 1 | 2 |
| 2 | 2 |
| 3 | 1 |

## 4.3 Using the Interconnected Finite State Machine Model to Describe Algorithms

This section describes how the Interconnected Finite State Machine Model is used for describing and evaluating fault-tolerant systems. First, its use in modeling the Direct Reconfiguration Algorithm [Sam86] is discussed. Then, the IFSMM for a distributed diagnosis algorithm is described [Hos89]. Then, the White-Gray Local array algorithm will be modeled [Whi88]. The Distributed Recovery Strategy [Yan86] will also be modeled. Next, the model for the Distributed Recovery Strategy-based array algorithm described in Chapter 2 will be discussed. Finally, the IFSMM will be shown to encompass the TAM and thus the LSM.

### 4.3.1 Modeling the Direct Reconfiguration Algorithm using the IFSMM

The Direct Reconfiguration Algorithm presented by Sami [Sam86] is a straight-forward approach to array reconfiguration. A row of spares is added to the south, and a column of spares is added to the east. The basic principle of the algorithm is that reconfiguration around faults goes to the east or to the south. Reconfiguration to the east is called horizontal reconfiguration. Reconfiguration to the south is called vertical reconfiguration. For horizontal

reconfiguration, each cell east of the faulty cell takes on its west neighbor's state. Vertical reconfiguration is similar. The reconfiguration algorithm is implemented in hardware.

The Direct Reconfiguration Structure assumes that each cell tests itself. The result of a cell's self-test is the signal *e(i,j)*. The signal *e(i,j)* is true if the cell is faulty and false if the cell is not faulty. Link failures are considered to be the same as if the cell sending information across that link were faulty.

When a faulty cell is detected, that cell must be marked as either a vertical fault or a horizontal fault. Vertical faults are handled using vertical reconfiguration which involves the spare in the spare row, while horizontal faults are handled using horizontal reconfiguration which involves the spare in the spare column. After faults are marked as horizontal or vertical (determining the direction of reconfiguration), the functions of the cells are mapped to the active cells. No more than one fault in any row can be marked as horizontal, and no more than one fault in any column can be marked as vertical. A further limitation is imposed to simplify the assignment problem: the southern-most fault in a column is marked as a vertical fault. Horizontal reconfiguration, or renaming, is performed first, followed by vertical renaming. In other words, states are reassigned along rows and then along columns.

The reconfiguration process is performed by combinational logic. The circuit which controls this reconfiguration is shown in Figure 48. The cell undergoes horizontal reconfiguration if the signal $e_x(i,j)$ is true. The cell

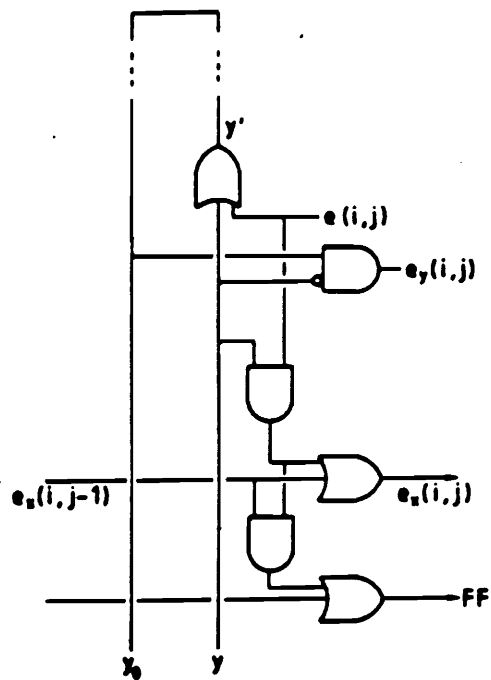
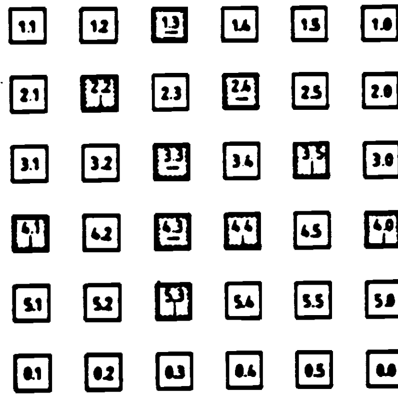
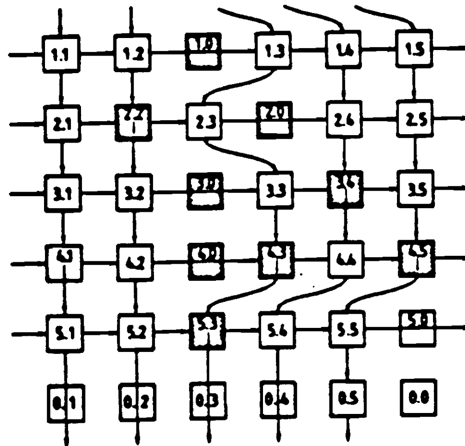


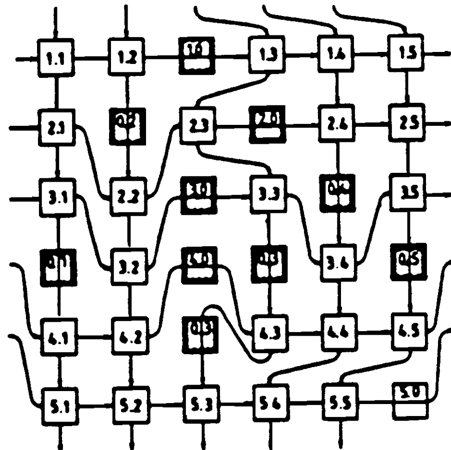

**Figure 48.** Direct Reconfiguration Algorithm Circuit[Sam86]



(a)



(b)



(c)

**Figure 49.** Direct Reconfiguration Algorithm Example[Sam86]

undergoes vertical reconfiguration if the signal $e_y(i,j)$ is true. In certain situations, a cell may undergo both horizontal and vertical reconfiguration. An example is shown in Figure 49. Further details of the Direct Reconfiguration Structure are given in [Sam86].

In this section the use of the Interconnected Finite State Machine Model to describe and evaluate the Direct Reconfiguration Algorithm is shown. As in Chapter 3, a 6 X 6 array in which one column and one row are spares will be used.

LEVEL 1

1-1 The facility graph representing the redundant system is a completely connected graph. This step is the same as Step 1-1 in the development of the TAM (see Chapter 3). Since the extreme cells in the array must have complete knowledge of the rest of the array, a completely connected graph is needed. In the Direct Reconfiguration Algorithm array, combinational logic is used to pass information throughout the array rather than using direct connections between all cells. The combinational logic is hard core. The model represents both direct and indirect communication between the components of the redundant system.

1-2 *A*, the set of active nodes, is the set of vertices in the graph found in Step 1-1. This step is identical to Step 1-2 in the TAM development process.

1-3 As in the TAM, a single boundary cell is used. Edges are added between the active cells and the boundary cell, so each active cell has the same neighborhood structure. Since the graph determined in Step 1-1 is regular, additional edges are not needed to make the degree of each node the same. However, edges are needed to make the description of the algorithm more convenient by making the neighborhood structure consistent. For example, we would like each cell to have a north neighbor since the response of each cell depends on its inputs from its north neighbor. Each active cell has the same neighborhood as described in Chapter 3.

1-4 A cell must send its logical state to its south, east, and southeast neighbors. This requirement is needed because a cell may take on its north, west, or northwest neighbor's state depending on the situation.

1-5 To determine the information to be sent to its neighbors and to determine its next state, a cell needs to know whether each of its neighbors is faulty. In addition, it needs to know its north, west, and northwest neighbors' states.

LEVEL 2

2-1 The output of a cell changes when its logical state changes. The cell's south, east, and southeast neighbors should be sent the new information. In addition, when a cell becomes faulty, the output of the cell should be changed to indicate that the cell is faulty.

2-2 To determine its new outputs, each cell needs to know the logical states of its north, west, and northwest neighbors. Also, each cell must know whether each of its neighbors is faulty.

2-3 The state information for a cell is described using a format similar to the C structure notation. The state information consists of two parts: the logical state and the error signal which is true if the cell itself is faulty. Pseudocode for the output function follows:

$output_{south}$ = state.logical_state
$output_{east}$ = state.logical_state
$output_{southeast}$ = state.logical_state

for (each output i)
    $output_i$.error = state.error

2-4 Pseudocode for next state function is the similar to that of the TAM. The next state function corresponds directly with the system's hardware. As indicated in Step 2-3, each state is composed of the logical state, *logical_state*, and the self-test result, *error*. In addition, a signal *recon_complete* is used to indicate the completion of the reconfiguration

process. The Direct Reconfiguration Algorithm implements this function using combinational logic.

```
nstate.error = pstate.error
if (pstate.recon_complete = FALSE)
        /* determine ex */
        ex = FALSE
        for each neighbor n west of the cell in the same row
                yin = FALSE
                if any of n's neighbors to the south in the same
                                column is faulty
                        yin = TRUE
                if (yin = TRUE and n is faulty)
                        ex = TRUE
        if (ex = FALSE)
                yin = FALSE
                for each neighbor n south of cell in same column
                        if (n is faulty)
                                yin = TRUE
                if (yin = TRUE and cell is faulty)
                        ex = TRUE

        /* determine ey */
        yin = FALSE
        if (any neighbor to south in same column is faulty)
                yin = TRUE
        y0 = FALSE
        if (any neighbor in same column is faulty)
                y0 = TRUE
        if (y0 = TRUE and yin = FALSE)
                ey = TRUE
        else
                ey = FALSE

        /* determine north ex */
        nex = FALSE
        for (each neighbor n to the west of north neighbor and the
                        north neighbor itself)
                yin = FALSE
                if (any neighbor south of n is faulty)
                        yin = TRUE
                if (yin = TRUE and n is faulty)
```

```
                nex = TRUE
if (ey = TRUE)
        if (nex = TRUE)
                nstate.activity = in[NW].activity
        else
                nstate.activity = in[N].activity
else
        if (ex = TRUE)
                nstate.activity = in[W].activity
        else
                nstate.activity = pstate.activity
if (ex = TRUE or ey = TRUE or nex = TRUE)
        recon_complete = TRUE
else
        recon_complete = FALSE
```

LEVEL 3

3-1 The next state function and the next output function routines for the Level 3 IFSM Model follow the pseudocode developed for the Level 2 model. As the Level 3 model was developed, the pseudocode was modified to more accurately reflect the algorithm. The C routines associated with these functions are given in Appendix D. These routines are also discussed in greater detail in the appendix.

3-2 The general purpose C main routine and supporting routines listed in Appendix C were used.

3-3 The Direct Reconfiguration Algorithm performed as expected. All single faults were covered. More complicated fault patterns were also simulated correctly. The details of the simulations are given in Appendix D.

*STRUCT-1:*

1) Group $N$ processors of the linear array into disjoint clusters such that each cluster has $n$ processors. Name these clusters $C_1, C_2, \cdots, C_i, \cdots, C_m$, where $C_1$ and $C_m$ are input and output clusters, respectively.

2) Every cluster is $t$ connected.

3) The $j$th processor of every cluster $C_i$ is connected to the $(n + 1 - j)$th processors of the preceding cluster $C_{i-1}$ (except for the input cluster) and the following cluster $C_{(i+1)}$ (except for the output cluster). If $C_i$ is the input (output) cluster, then $P_j$ is connected to the external input (output) of the array.

**Figure 50.** Procedure for Constructing a Redundant Linear Array [Hos89]

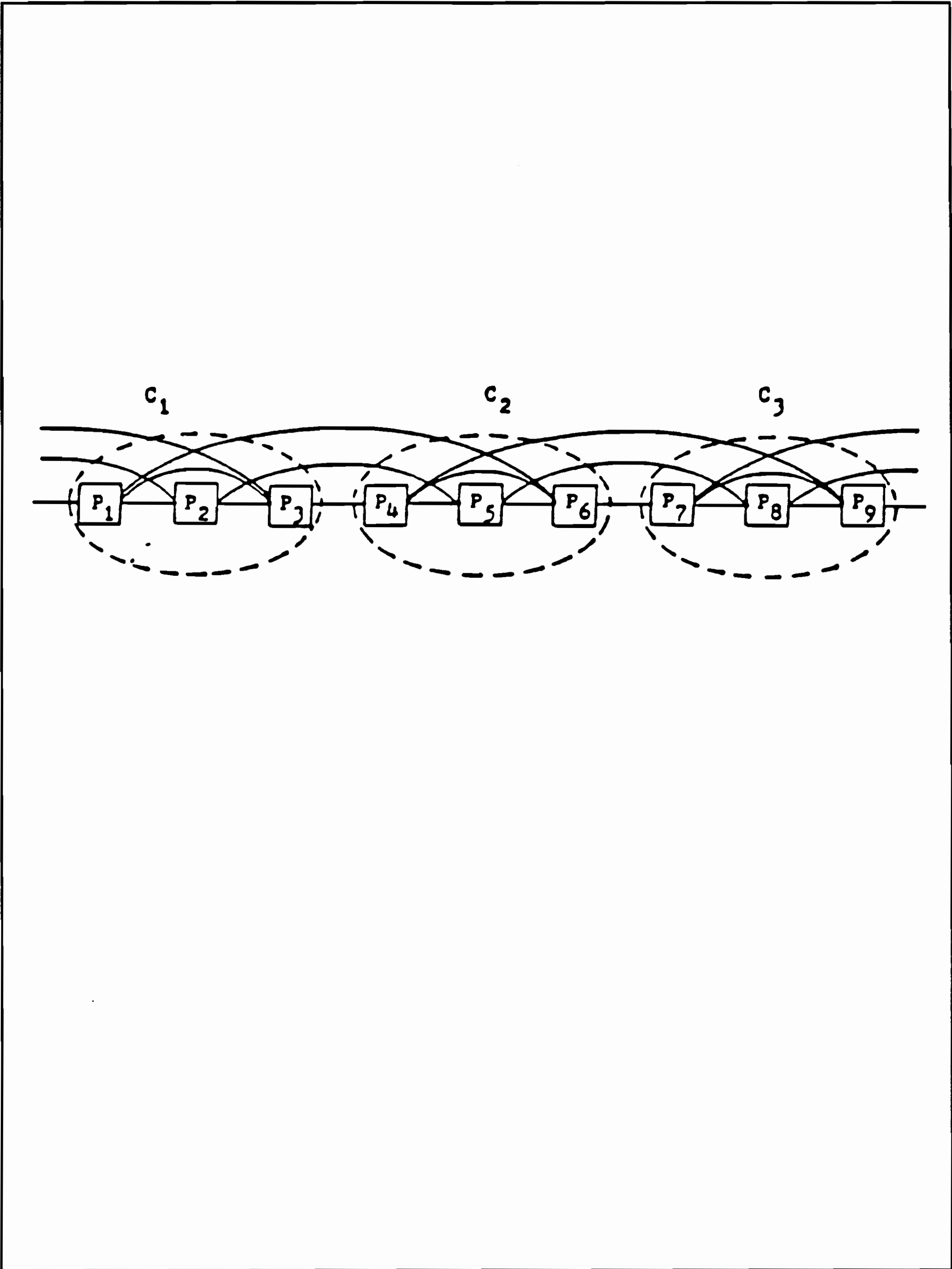



**Figure 51.** Redundant Linear Array with 3 Clusters of 3 Processors in which Clusters are 2-Connected [Hos89]

### 4.3.2 Modeling the DIAG-1 Diagnosis Algorithm Using the Interconnected Finite State Machine Model

A diagnosis algorithm, called DIAG-1, for a linear array is presented in [Hos89]. Using this distributed algorithm, each processor determines which of its neighbors and which of its communication links are faulty. The linear array architecture is constructed in a system of clusters. The procedure for constructing the redundant linear array is given in Figure 50. An example with 9 processor divided into 3 clusters each with size 3 is shown in Figure 51. In this example, each cluster is 2-connected.

DIAG-1 determines fault locations by passing test results between processors. DIAG-1 can identify faulty links, as well as faulty processors for the cell's cluster, its preceding cluster, and its following cluster. A test result is referred to by $a_{i,j}$ where i indicates that Processor $P_i$ tests Processor $P_j$ and $a_{i,j}$ is the result. If $P_i$ is faulty, then the test result is not reliable. It may be either 1 or 0. If $P_i$, $P_j$, and the communication path between $P_i$ and $P_j$ are not faulty, then $a_{i,j} = 0$. If $P_i$ is not faulty, but $P_j$ is faulty or the communication path between $P_i$ and $P_j$ is faulty, then $a_{i,j} = 1$. Test results are passed among processors as specified in the DIAG-1 algorithm. In this way, each processor can determine the fault status of each communication link and each processor in its own cluster and its neighboring clusters. The DIAG-1 Algorithm is given in Figure 52.

*Algorithm DIAG-1:*

1. Every processor $P_i$ should test every one of its neighbors $P_j$ and obtain the test outcome $a_{ij}$, then do

1.1 If $a_{ij} = 0$, then send $a_{ij}$ to every other neighbor $P_k$ which has passed the test by $P_i$ (i.e., $a_{ik} = 0$)

1.2 Else send a FAIL message and $a_{ij}$ to every neighbor $P_k$ which has passed the test by $P_i$ (i.e., $a_{ik} = 0$)

2. Every processor $P_i$ which has difficulty in communicating with an I/O device $P_j$ should send a FAIL message and a link $P_i$–$P_j$ to every neighbor $P_k$ which has passed the test by $P_i$ (i.e., $a_{ik} = 0$)

3. Every processor $P_x$, upon the reception of an $a_{ij}$ or a link $P_i$–$P_j$ from a neighbor $P_y$ which has passed the test by $P_x$ (i.e., $a_{xy} = 0$), should do

3.1 Accept the received message

3.2 If $P_j$ is not in the same cluster, then send the received message to all the neighbors $P_z$ which are in the same cluster and have passed the test by $P_x$ (i.e., $a_{xz} = 0$)

3.3 Else $P_j$ is in the same cluster and send the received message to all the neighbors $P_z$ which have passed the test by $P_x$ (i.e., $a_{xz} = 0$)

4. Every processor $P_x$ considers another processor $P_j$ fault-free if it has received at least a test outcome $a_{ij} = 0$

5. Every processor $P_x$ considers a link $P_i$–$P_j$ faulty if both $P_i$ and $P_j$ are fault-free, but $P_x$ has received either an $a_{ij} = 1$ or an $a_{ji} = 1$

6. Every time a processor $P_x$ receives, for the first time, a FAIL message from a neighbor $P_y$ which has passed the test by $P_x$ (i.e., $a_{xy} = 0$), then it should record this message and send it to every neighbor $P_z$ which has passed the test by $P_x$ (i.e., $a_{xz} = 0$).

**Figure 52.** DIAG-1 Diagnosis Algorithm [Hos89]

This section describes the process for developing an IFSMM of the DIAG-1 algorithm. This algorithm would be quite difficult to model using the TAM since different outputs are sent to different neighbors. Since it is not a reconfiguration algorithm, it cannot be modeled using LSM. This modeling process demonstrates the versatility of the IFSMM.

LEVEL 1

1-1 The facility graph representing the redundant hardware system is given in Figure 53.

1-2 The active nodes are shown in the graph in Figure 53. So, A = {1, 2, 3, 4, 5, 6, 7, 8, 9}.

1-3 Two boundary cells, B1 and B2, are added to the facility graph found in Step 1-1. Thus, B = {B1, B2}. The IFSMM graph created by adding these boundary cells to the facility graph defined above is given in Figure 54.

1-4 The only information passed between cells is the test result information and a failure flag. Each test result indicates the tester, the testee, and the result value. A data ready signal is needed to indicate the availability of valid data. In the actual implementation, a specific data ready flag is not needed if there is some other way to determine that valid data is available. The interconnection values are described by I = {(tester, testee, result $a_{ij}$, data_ready, failure)}.

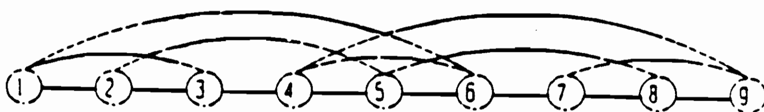



**Figure 53**. The Redundant System for DIAG-1

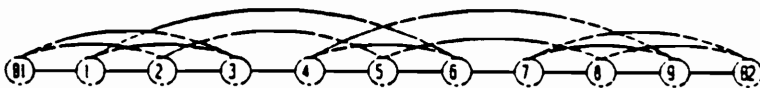



**Figure 54.** The IFSMM Graph for DIAG-1

1-5 At each time step, the cell needs accumulated data to determine the output information. This information is Q = { stored input data, own test results for neighbors, failure flag, array of known results, array of known links, array of known processors}.

LEVEL 2

2-1 A cell changes its outputs to other cells when it receives a failure flag or when it receives a test result.

2-2 To determine its new outputs, each cell needs the original test results (own tests of neighbors), its known test results (from previous communications), cumulative link and processor information, and failure flag.

2-3 Pseudocode for Next Output Function:

```
for each output
    set data_ready = FALSE
    get test result a_i,j
    if (original test result from valid processor)
        for each passing neighbor, cell k, k ≠ j (not boundary cell)
            if (a_i,j = 0) then
                send a_i,j to cell k
            else {a_i,j = 1}
                send FAIL MSG to cell k
                send a_i,j to cell k

for each I/O device j
    if (cell i cannot communicate with I/O device j)
        for each neighbor, cell k
            if (a_i,j = 0) then
                send FAIL MSG to cell k
                send BAD LINK (i,j) MSG to cell k

for each neighbor, cell j
```

```
    if ((data_ready from cell j) ∧ (a_i,j = 0)) then
        if (m is not in same cluster as i) then
            for each neighbor, cell k
                if (a_i,k = 0) then
                    if (k is in same cluster as i) then
                        send a_n,m to cell k
        else {same cluster}
            for each neighbor, cell k
                if (a_i,k = 0) then
                    send a_n,m to cell k

for each neighbor, cell j
    if ((a_i,j = 0) ∧ (no previous fail message)) then
        if (fail message) then
            for each neighbor, cell k
                if (a_i,k = 0) then
                    send fail message to cell k
```

2-4 Pseudocode for Next State Function

```
for each neighbor, cell j
    if ((data_ready from cell j) ∧ (a_i,j = 0)) then
        place new data a_n,m in results array

if (all results are in)
    for each cell i
        for each cell j
            if ((cell i = FAULTFREE) ∧ (cell j = FAULTFREE))
                if ((result (i,j) = 1) ∨ (results (j,i) = 1))
                    link (i,j) = FAULTY

for each neighbor, cell j
    if ((a_i,j = 0) ∧ (no previous fail message)) then
        if (fail message) then
            failure = TRUE
```

<u>LEVEL 3</u>

3-1 The routines which are used to simulate the output function, $\lambda$, and the next state function, $\delta$ were written using the C programming language. These C routines generally follow from the pseudocode developed as part

of the Level 2 IFSMM. These routines are given in Appendix E. The simulation routines interpret the DIAG-1 algorithm in such a way that a cell does not repeatedly send the same results to its neighbors. Because of the need to queue inputs, this simulation model is complicated. Link lists are used to simulate the queuing activity at the inputs for each cell. For these reasons, the C routines do not directly follow the pseudocode given in Level 2. However, the pseudocode describes the key aspects of the algorithm itself.

3-2 This Level 3 IFSMM uses the general structure main routine and supporting routines given in Appendix C.

3-3 The algorithm was simulated for several cases: no faults in the system, a single faulty link, the example given in [Hoss89]. In each case, the algorithm behaved as expected. Additional information is given in Appendix E.

### 4.3.3 Modeling the White-Gray Algorithm Using the Interconnected Finite State Machine Model

The White-Gray Algorithm [Whi88] consists of two parts: the local mode and the global mode. The local mode is used when it can cover the faults. The local mode can correctly reconfigure the array when there is no more than one faulty cell in each row. The global mode is used whenever more than one fault occurs in some row of the active array. The redundant array described in Chapter 2 is used. Each cell in the redundant array has connections to its

north, northeast, east, far east, southeast, south, southwest, west, far west, and northwest neighbors. The array is conceptually divided into two parts: the control plane and the computation plane. The control plane controls the reconfiguration of the system.

Each cell has active connections to four neighbors at any time. In the local mode, each cell passes a ten bit fault register to each of its active neighbors. Then, based on the information it receives from its active neighbors, each node updates its fault register and passes the new value. This process of updating the fault register and passing it to active neighbors is continued until an equilibrium is reached. Based on the contents of its fault register, each cell determines what connections it should activate and whether or not it should take the state of its logical west neighbor.

If a node determines that there are two or more faults in a single row, the global mode is initiated. The details of the global mode are given in [Kum84, Whi88].

A switch-controlled communications scheme is used to isolate faulty cells. The behavior of faulty cells is assumed to be totally unpredictable. Each cell tests its active neighbors (initially NORTH, SOUTH, EAST, and WEST). It is assumed that all active neighbors of a faulty cell will determine that the cell is faulty during the same clock period. [Gra89]

LEVEL 1

1-1 The facility graph representing the redundant system is shown in Figure 55. This graph is the same as is used for the Array Reconfiguration Algorithm.

1-2 The set of active nodes, A, is the set of vertices in the graph found in Step 1-1.

1-3 Boundary cells are added around the array. A single boundary cell could be used. Figure 56 shows the addition of the boundary cells. The graph in Figure 56 is the graph portion of the IFSMM.

1-4 Fault register contents are passed between cells. Thus, the set of interconnection values, I, is the set of all possible fault register contents. A cell's fault register consists of ten bits which indicate the fault status of the previous, the same, and the next rows as shown below

| bit | fault location |
|---|---|
| 1 | west cell faulty |
| 2 | north cell faulty |
| 3 | south cell faulty |
| 4 | any cell west of the west cell faulty |
| 5 | any cell west of the north cell faulty |
| 6 | any cell west of the south cell faulty |
| 7 | east cell faulty |
| 8 | any cell east of the east cell faulty |
| 9 | any cell east of the north cell faulty |
| 10 | any cell east of the south cell faulty |

1-5 To determine its output information (updated fault register) each cell needs to know its current fault register, its active connections, its own computation state, its logical west neighbor's computation state, test

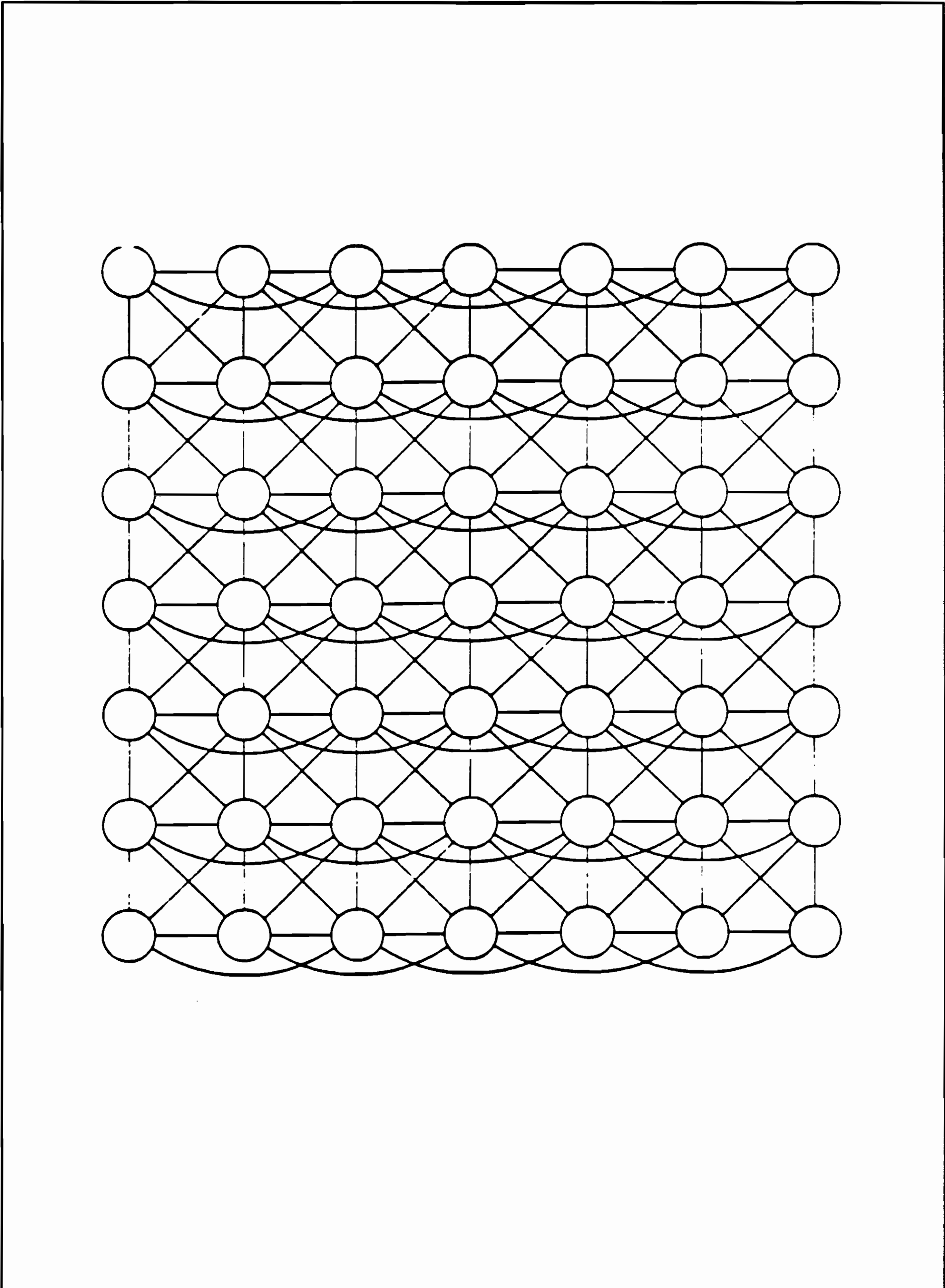


**Figure 55.** The Redundant System for the White-Gray Local Algorithm

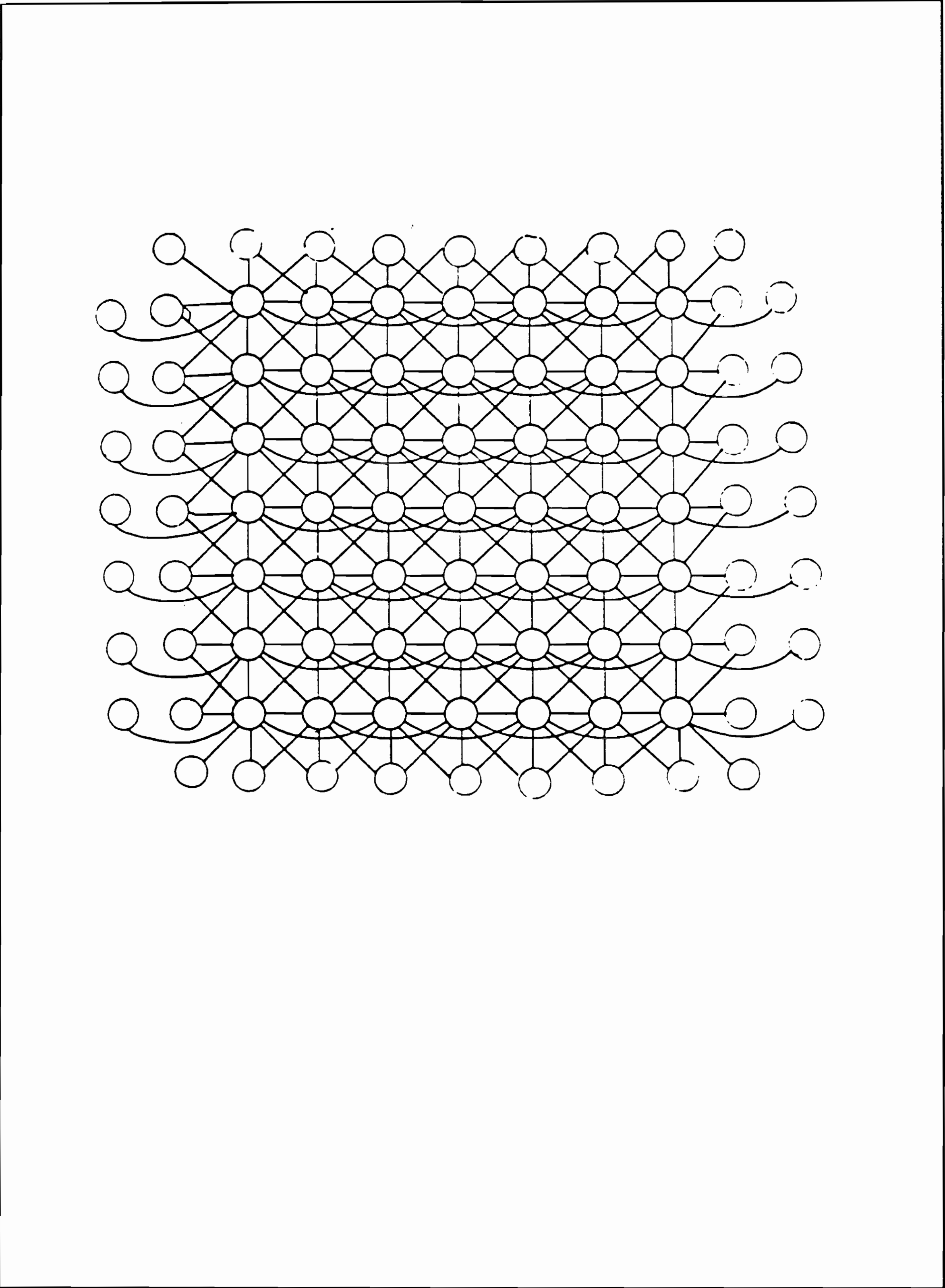


**Figure 56.** The IFSMM Graph for the White-Gray Local Algorithm

results for all its active neighbors, and whether or not it should take its west neighbor's state. This information is the cell's state. The active connections can be determined directly from the fault register contents, but it has been separated in this model for clarity.

LEVEL 2

2-1 Each time step, every cell sends its fault register contents to its active neighbors. For a communication path to be established, both of the communicating cells must successfully attempt communication.

2-2 To determine the new outputs, the current fault register contents are needed and the fault register contents of all of the active neighbors are needed.

2-3 Pseudocode for the Output Function

Output fault register and computation state (activity indicator) to active neighbors
No output to inactive neighbors

More formally,

Let r(Q) denote the fault register and computation state information for state Q.
Let c(Q) denote the active connections for state Q.
Then,

$$\lambda\,(Q, i_n, i_{ne}, i_{fe}, i_e, i_{se}, i_s, i_{sw}, i_w, i_{fw}, i_{nw}) = (o_n, o_{ne}, o_{fe}, o_e, o_{se}, o_s, o_{sw}, o_w, o_{fw}, o_{nw})$$

where $o_d$ = r(Q) if and only if the cell is active else $o_d$ = NO OUTPUT

2-4 Pseudocode for Next State Function

Let t denote a test result. A test result of "1" means FAULTY.
Let i denote an input to the cell.

$a_i$ = 1 if connection i is activated
$f_j$ is the j bit of the fault register
$t_k$ is the test result for neighbor k

Determine test values for active neighbors (if a connection is not active, assume that the test result is NOT FAULTY)

Determine fault register content

| | | | |
|---|---|---|---|
| 1 | $f_W$ | = | $t_W$ |
| 2 | $f_N$ | = | $t_N + i_{NE,W} + i_{NW,E}$ |
| 3 | $f_S$ | = | $t_S + i_{SE,W} + i_{SW,E}$ |
| 4 | $f_{FW}$ | = | $t_{FW} + i_{W,W} + i_{W,FW}$ |
| 5 | $f_{NW}$ | = | $t_{NW} + i_{N,W} + i_{N,FW} + i_{NW,W} + i_{NW,FW} + i_{W,N} + i_{W,SW}$ |
| 6 | $f_{SW}$ | = | $t_{SW} + i_{S,W} + i_{S,FW} + i_{SW,W} + i_{SW,FW} + i_{W,S} + i_{W,SW}$ |
| 7 | $f_E$ | = | $t_E$ |
| 8 | $f_{FE}$ | = | $t_{FE} + i_{E,E} + i_{E,FE} + i_{FE,E} + i_{FE,FE}$ |
| 9 | $f_{NE}$ | = | $t_{NE} + i_{NW,FE} + i_{N,E} + i_{N,FE} + i_{NE,E} + i_{NE,FE} + i_{E,N} + i_{E,NE} + i_{FE,N} + i_{FE,NE}$ |
| 10 | $f_{SE}$ | = | $t_{SE} + i_{SW,FE} + i_{S,E} + i_{S,FE} + i_{SE,E} + i_{SE,FE} + i_{E,S} + i_{E,SE} + i_{FE,S} + i_{FE,SE}$ |

Activate connections in computation plane

$$a_{FW} = t_W$$

$$a_{FE} = t_E$$

$$a_N = f_N' \wedge ((f_{NE} \wedge (f_E \vee f_{FE}) \wedge (f_W' \wedge f_{NW}' \wedge f_{FW}') \vee (f_W' \wedge f_{NW}' \wedge f_{FW}') \wedge ((f_{NW}' \wedge f_{NE}') \wedge (f_{FE} \vee f_E)) \vee (f_E' \wedge f_{NW} \wedge f_{NE}' \wedge f_{FE}' \wedge (f_W \vee f_{FW})))$$

$$a_S = (f_{FW} \wedge f_{SE} \wedge f_{SW}' \wedge f_W' \wedge (f_E \vee f_{FE})) \vee (f_{FE}' \wedge f_{SE}' \wedge f_{SW} \wedge f_E' \wedge f_S' \wedge (f_W \vee f_{FW})) \vee (f_S \wedge ((f_E \wedge (f_{SW}' \vee f_{SE})) \vee (f_{FE} \wedge (f_{SE} \vee f_{SW}')) \vee (f_W' \wedge f_{SE} \wedge f_{FW}')))$$

$$a_E = f_E'$$

$$a_W = f_W'$$

$$a_{NE} = (f_N' \wedge f_{NW}' \wedge f_W \wedge f_{FW} \wedge f_{NE} \wedge (f_E \vee f_{FE})) \vee (f_W' \wedge f_{NE}' \wedge f_{FW} \wedge (f_N \vee f_{NW})) \vee (f_{NE}' \wedge ((f_E \wedge (f_N \vee f_{NW})) \vee (f_{FE} \wedge (f_N \vee f_{NW}))))$$

$$a_{SE} = (f_S \vee f_{SW}) \wedge (f_{FW}' \wedge f_{SE}' \wedge f_W') \vee (f_E \vee f_{FE}) \wedge (f_{SE}' \wedge (f_S' \vee f_{SW}))$$

$$a_{NW} = (f_E' \wedge f_{FE}') \wedge ((f_N \wedge (f_W \vee f_{FW})) \vee (f_{NE} \wedge (f_W \vee f_{FW})) \vee (f_W \wedge f_{NW}') \vee (f_{FW} \wedge f_{NW}' \wedge f_E'))$$

$$a_{SW} = (f_S \wedge f_E' \wedge f_{SE}' \wedge f_{FE}' \wedge (f_W \vee f_{FW})) \vee (f_S' \wedge f_E' \wedge f_{SE} \wedge f_{FE}' \wedge (f_W \vee f_{FW})) \vee (f_E' \wedge f_W \wedge f_{FE}' \wedge (f_{FW} \vee f_{SE} \vee f_{SW})) \vee (f_E' \wedge f_{FE}' \wedge f_{SW}' \vee f_{FW})$$

Development of this pseudocode was hindered by the inaccuracies in documentation. Both [Whi88] and [Gra89] contained descriptions which specified the way in which the fault register contents should be determined which were in conflict. In addition, a personal communication [Whi90] contained a third description for determining fault register contents. After much study, the definition in [Whi88] appears to be the most correct. However, it has several omissions and a slight error. Two omissions involve the use of test results to set the Northeast and Southeast bits of the fault register. The Northeast bit should be set if the cell is connected to a faulty northeastern cell. Similarly, the Southeast bit should be set if the cell is connected to a faulty southeastern cell. The small error refers to one way in which the Northeast bit is set. The Northeast bit should be set if the fault information from the northwestern cell contains bit 8 (Far East) <u>rather than</u> bit 4 (Far West). In addition, the definition for the Northwest bit of the fault register should be set if the input from the west neighbor indicates that its north or northwest neighbor is faulty. Similarly, the Southwest bit of the fault register should be set if the input from the west neighbor indicates that its south or southwest neighbor is faulty.

The IFSMM for the White-Gray Local Algorithm is very useful for identifying errors in the description. However, through this work the need for a graphical interface became more evident than it had been previously. Identifying the problems with the algorithm was tedious because the output from the simulation routines had to be interpreted each time. If a graphical representation were available, interpretation of the simulation results would be much simpler.

Since detailed information about determining active connections is not readily available in published information, this information was obtained through a personal communication [Whi90]. This information could have been obtained from the ADAS simulation programs supplied in [Whi88] however, it would have been quite cumbersome. This problem is one that might arise in simulating other people's designs. If their documentation does not clearly define the behavior of the algorithm at every level, then additional information must be obtained or suitable assumptions must be made. Alternatively, some details of the algorithm could be omitted from the IFSMM.

One error was identified in the description of how the active connections are determined. In determining whether the south connection is active, four cases were omitted. These cases are when no bits of the fault

register are set, when only the northwest bit of the fault register is set, when only the north bit of the fault register is set, and when only the northeast bit of the fault register is set.

LEVEL 3

3-1 The pseudocode developed as part of the Level 2 IFSM Model was used to develop C routines which implement the output function, $\lambda$, and the next state function, $\delta$. These routines are given in Appendix F with additional discussion.

3-2 The general structure main routine and supporting routines written in C are used (see Appendix C).

3-3 As described above, through simulation errors were identified in the descriptions of the White-Gray local algorithm which caused reconfiguration to take place incorrectly. These errors were corrected so that reconfiguration around faults is correct. These corrections are described in detail in Appendix F with the detailed description of the Level 3 model. After a correctly reconfiguring model was developed, simulations were run for all single faults. In addition, an example involving multiple faults was simulated. In each case correct reconfiguration occurs. These simulations are described in detail in Appendix F.

### 4.3.4 Modeling the Distributed Recovery Strategy Using the Interconnected Finite State machine Model

The Distributed Recovery Strategy [Yan86] was discussed in detail in Chapter 2. This section describes the Interconnected Finite State Machine Model for the Distributed Recovery Strategy.

LEVEL 1

1-1 The redundant graph for the Distributed Recovery Strategy, $G_r$, is the facility graph for the redundant hardware system.

1-2 The set of active cells, A, is the set of the nodes in the graph found in Step 1-1.

1-3 Since all boundary conditions are the same, add a single boundary cell. Add edges between the boundary cell and active nodes so that the degree of each active node is the same and neighborhood structure is consistent.

1-4 The information passed between nodes is the node's logical state and the node's neighbors' logical states. Takeover signals are also needed to indicate to spare nodes what states they should take and when.

1-5 To determine its output information, a cell needs to know its current logical state, its neighbors' logical states, and its neighbors' neighbors' logical states. In addition, it must know the errors to which it responds and required adjacencies for those error states.

LEVEL 2

2-1 A cell changes its outputs when its logical state changes or when one of its neighbor's logical state changes. It also sends a takeover signal to a spare to tell that spare to take over an error state.

2-2 To determine its new outputs, a cell needs to know its current logical state, its neighbors' logical states, and its neighbors' neighbors' logical states, as well as, the errors to which it responds and the required adjacencies for those error states.

2-3 Pseudocode for the Next Output Function

```
CASE (cell's activity indicator)
        faulty:
                do nothing;
        normal computation state:
                send activity indicators for self and neighbors
                if ((not (for each error response assignment, some
                        cell in the neighborhood of the cell has that
                        computation state)) and (spare in neighborhood with
                        necessary connections to takeover))
                                then tell spare to takeover;
        spare:
                do nothing;
```

2-4 Pseudocode for the Next State Function

```
CASE (cell's activity indicator)
        faulty:
                do nothing;
        normal computation state:
                if (no current values for neighbors' states) then
                        get current values from inputs;
                if (for each error response assignment, some cell
                        in the neighborhood of the cell has that computation
                        state) then do nothing;
                else (error condition exists)
```

```
                if (no spare in neighborhood)
                        if (cell can takeover) then takeover
                                error state;
                        else (cell can't takeover)
                                RECONFIGURATION FAILS;
                else (spare in neighborhood)
                        if (spare can't takeover)
                                takeover error state;
                        else (cell can't takeover)
                                RECONFIGURATION FAILS;
        spare:
                if (input from cell that says takeover state) then
                        takeover error state;
                else (no takeover input)
                        do nothing;
```

LEVEL 3

3-1 The pseudocode developed for the Level 2 IFSM Model was used to develop C routines which implement the output function, $\lambda$, and the next state function, $\delta$. The ADAS CSIM routines described in [Gra88] were also used in the development of these C routines. In fact, parts of the ADAS CSIM routines are identical to the C routines for the Level 3 IFSMM. The C routines are given in Appendix G. Additional information about the Level 3 modeling process is also provided in Appendix G.

3-2 The general structure main routine and supporting routines written in C are used (see Appendix C).

3-3 A specific system was chosen to illustrate the use of the Distributed Recovery Strategy. Since the IFSMM can only be used to model

reconfiguration for regular architectures, not all cases in which the Distributed Recovery Strategy can be used is represented by the model. However, the behavior of the Distributed Recovery Strategy for regular architectures is fully modeled. The system used to illustrate the Distributed Recovery Strategy is the redundant cyclic system described in [Yan86]. The system reconfigured correctly for all single faults as expected. Details of this system and the IFSMM are furnished in Appendix G.

### 4.3.5 Modeling of the Array Reconfiguration Algorithm Using the Interconnected finite State Machine Model

The Array Reconfiguration Algorithm is described in detail in Chapter 2. This section describes the Interconnected Finite State Machine Model developed for this algorithm. This IFSM model is the same as for the Distributed Recovery Strategy because of the close relationship between the Distributed Recovery Strategy and the Local Supervisor Model. Only the input files for the graph and initial state are different.

LEVEL 1

1-1 The facility graph representing the redundant system is the same as that developed for the TAM in Chapter 3. It is also the same as the facility graph developed for the IFSMM model of the White-Gray Local algorithm (see Figure 55).

1-2 A, the set of active nodes, is the set of vertices in the facility graph in Step 1-1.

1-3 Boundary cells are added around the edges of the array as in the IFSM model of the White-Gray Local algorithm (see Figure 56). Once again, a single boundary cell could have been used.

1-4 To determine its next outputs, a cell needs to know its neighbors' logical states and its neighbors' neighbors' logical states. In addition, if it is a spare it needs indicators from its neighbors that it should take on a specified state (if appropriate).

1-5 To determine its next state, a cell needs to know the error conditions to which it responds, its own logical state, its neighbors' logical states, and its neighbors' neighbors' logical states.

LEVEL 2

2-1 At each time step, neighbors need to know a cell's logical state and that cell's neighbors' logical states. When a spare cell needs to take on a new state, the responding node must send it an appropriate message.

2-2 A cell's neighbors' logical states need to be maintained in its state information.

2-3 Pseudocode for Output Function (This pseudocode is identical to the pseudocode in the Level 2 IFSMM of the Distributed Recovery Strategy.)

```
CASE (cell's activity indicator)
      faulty:
            do nothing;
```

```
normal computation state:
        send activity indicators for self and neighbors
        if ((not (for each error response assignment, some
                cell in the neighborhood of the cell has that
                computation state)) and (spare in neighborhood with
                necessary connections to takeover))
                        then tell spare to takeover;
spare:
        do nothing;
```

2-4 Pseudocode for Next State Function (This pseudocode is identical to the pseudocode in the Level 2 IFSMM of the Distributed Recovery Strategy.)

```
CASE (cell's activity indicator)
    faulty:
        do nothing;
    normal computation state:
        if (no current values for neighbors' states) then
                get current values from inputs;
        if (for each error response assignment, some cell
                in the neighborhood of the cell has that computation
                state) then do nothing;
        else (error condition exists)
                if (no spare in neighborhood)
                        if (cell can takeover) then takeover
                                error state;
                        else (cell can't takeover) then
                                RECONFIGURATION FAILS;
                else (spare in neighborhood)
                        if (spare can't takeover)
                                takeover error state;
                        else (cell can't takeover)
                                RECONFIGURATION FAILS;
    spare:
        if (input from cell that says takeover state) then
                takeover error state;
        else (no takeover input)
                do nothing;
```

LEVEL 3

3-1 Routines written in C which implement the output function and the next state function are given in Appendix G. These routines are the same as for the Distributed Recovery Strategy.

3-2 The general structure main routine and related supporting routines is given in Appendix C.

3-3 The algorithm was simulated for all single and all double sequential faults. A double sequential fault is defined to be "two single faults separated in time so that reconfiguration in response to the first fault has completed before the second fault occurs." [Gra88] As expected, the simulations show that 100% of single faults are covered, and 83% of double sequential faults are covered. The time requirements for reconfiguration around single faults are given in Figure 57. Time requirements for reconfiguration around double sequential faults are given in Figure 58. These results agree with the results presented in [Gra88]. Simulation results are presented in more detail in Appendix H.

### 4.3.6 Using the Interconnected Finite State machine Model to Describe the Tessellation Automata Model

The Tessellation Automata Model and the Interconnected Finite State Machine Model are closely related. They both use the same graph structure. The difference between the two models is the communications between cells.

| Error Condition | Faulty Cell | ARA Steps |
|---|---|---|
| E(1) | 16 | 1 |
| E(2) | 17 | 2 |
| E(3) | 18 | 1 |
| E(4) | 23 | 1 |
| E(5) | 24 | 2 |
| E(6) | 25 | 1 |
| E(7) | 30 | 1 |
| E(8) | 31 | 2 |
| E(9) | 32 | 1 |

**Figure 57.** Time Requirements of the Array Reconfiguration Algorithm for Single Faults

| Faults | ARA Steps | Faults | ARA Steps |
|---|---|---|---|
| E(1)E(1) | 4 | E(6)E(1) | 2 |
| E(1)E(2) | Fails | E(6)E(2) | 3 |
| E(1)E(3) | 2 | E(6)E(3) | 2 |
| E(1)E(4) | 2 | E(6)E(4) | 2 |
| E(1)E(5) | 3 | E(6)E(5) | Fails |
| E(1)E(6) | 2 | E(6)E(6) | Fails |
| E(1)E(7) | 2 | E(6)E(7) | 2 |
| E(1)E(8) | 3 | E(6)E(8) | 3 |
| E(1)E(9) | 2 | E(6)E(9) | 2 |
| E(2)E(1) | Fails | E(7)E(1) | 2 |
| E(2)E(2) | Fails | E(7)E(2) | 3 |
| E(2)E(3) | 3 | E(7)E(3) | 2 |
| E(2)E(4) | 3 | E(7)E(4) | 2 |
| E(2)E(5) | 4 | E(7)E(5) | 3 |
| E(2)E(6) | 3 | E(7)E(6) | 2 |
| E(2)E(7) | 3 | E(7)E(7) | 4 |
| E(2)E(8) | 4 | E(7)E(8) | Fails |
| E(2)E(9) | 3 | E(7)E(9) | 2 |
| E(3)E(1) | 2 | E(8)E(1) | 3 |
| E(3)E(2) | Fails | E(8)E(2) | 4 |
| E(3)E(3) | 3 | E(8)E(3) | 3 |
| E(3)E(4) | 2 | E(8)E(4) | 3 |
| E(3)E(5) | 3 | E(8)E(5) | 4 |
| E(3)E(6) | 2 | E(8)E(6) | 4 |
| E(3)E(7) | 2 | E(8)E(7) | Fails |
| E(3)E(8) | 3 | E(8)E(8) | Fails |
| E(3)E(9) | 2 | E(8)E(9) | 3 |
| E(4)E(1) | 2 | E(9)E(1) | 2 |
| E(4)E(2) | 3 | E(9)E(2) | 3 |
| E(4)E(3) | 2 | E(9)E(3) | 2 |
| E(4)E(4) | 4 | E(9)E(4) | 2 |
| E(4)E(5) | Fails | E(9)E(5) | 3 |
| E(4)E(6) | 2 | E(9)E(6) | 2 |
| E(4)E(7) | 2 | E(9)E(7) | 2 |
| E(4)E(8) | 3 | E(9)E(8) | Fails |
| E(4)E(9) | 2 | E(9)E(9) | Loop |
| E(5)E(1) | 3 | | |
| E(5)E(2) | 4 | | |
| E(5)E(3) | 3 | | |
| E(5)E(4) | Fails | | |
| E(5)E(5) | Fails | | |
| E(5)E(6) | Fails | | |
| E(5)E(7) | 3 | | |
| E(5)E(8) | 4 | | |
| E(5)E(9) | 3 | | |

**Figure 58.** Time Requirements of the Array Reconfiguration Algorithm for Double Sequential Faults

In the TAM, all state information is sent to each neighbor. In the IFSMM, the output to each neighbor is defined explicitly through the next output function.

Any algorithm which can be modeled using the TAM can be modeled using the IFSMM. To directly convert a TAM model to an IFSMM, the next output function should define the output to each neighbor as part of the current state.

## 4.4 Summary

The Interconnected Finite State Machine Model (IFSMM) was developed to overcome the TAM's awkward handling of sending different information to neighbors. The IFSMM differs from the TAM in that a cell's outputs, as well as its next state, is determined by the cell's state and its neighbors' states. In addition to better management of the communications between cells, the IFSMM clearly distinguishes between output information and state information. The separate definition of outputs is a significant improvement over the TAM.

The IFSMM is a hierarchical model with the same basic structure as the TAM. The specifics of the hierarchical modeling process differ from the TAM process because of the specifics of the IFSMM; however, the key aspects are the same. High level models are developed rapidly. Lower level models are developed only when additional detail is required.

The IFSMM can be used to describe algorithms, evaluate and compare algorithms, and to design new algorithms. The IFSMM has been used to

describe a wide variety of distributed algorithms and approaches: the Direct Reconfiguration Algorithm [Sam86], a distributed diagnosis algorithm [Hos89], the White-Gray Local Algorithm [Whi88], the Distributed Recovery Strategy [Yan86], and the LSM Array Reconfiguration Algorithm. The IFSMM was used to identify description errors in the White-Gray Local algorithm. In addition, the IFSMM is shown to encompass the TAM. The use of the IFSMM in comparing systems, improving algorithms, and designing an algorithm for a specific application is discussed in later chapters.

# 5.0 Using the Interconnected Finite State Machine Model to Improve Distributed Algorithms

This chapter addresses the problem of improving existing algorithms. The Interconnected Finite State Machine Model is used to model an algorithm and identify its weaknesses. Then, the model is used to identify improvements to the algorithm and assess the improved algorithm. First, this chapter describes how the IFSMM is used to improve algorithmic aspects of the Local Supervisor Model. Then, the use of the IFSMM to improve the Direct Reconfiguration Algorithm is described.

## 5.1 Improving the Local Supervisor Model

In the development of the Local Supervisor Model, several areas were identified as needing further consideration:

1. The Use of Spares to Respond to Errors
2. The Problem of Predicting an Algorithm's Coverage
3. The Handling of Multiple and Near-Coincident Faults

4. The Implementation of a Local Supervisor Model Algorithm

These areas were identified during the attempt to apply the Distributed Recovery Strategy to a different architecture. In general, using the IFSMM to model an existing algorithm would identify the algorithm's weaknesses more effectively because the systematic method of attack considers all aspects of the algorithm in turn. Although these areas of improvement were identified before the IFSMM was developed, the IFSMM can be used to address the needed improvements. The specific use of the IFSMM to identify areas for improvement is illustrated in Section 5.2 in which the Direct Reconfiguration Algorithm is considered.

### 5.1.1 The Use of Spares to Respond to Errors

In various systems, using spares to respond to error conditions can produce a redundant system that has a more consistent reconfiguration response. However, when spares are used to respond to errors, the question arises "What happens if the spare which responds to the error becomes faulty?" Since spare cells do not have an operating computation state, they are not required for the system to operate properly. Because spares are not needed for system operation, when a spare becomes faulty no active cell will "miss" the spare cell's non-existent active state. Thus, reconfiguration is not triggered.

The problem of spares is identified by considering how reconfiguration is carried out in general. Although this problem was identified before the IFSMM was devised, the IFSMM could help in problem identification through

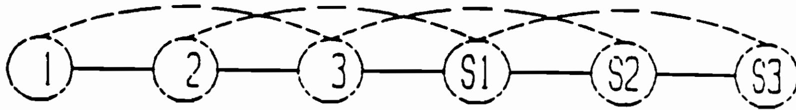
consideration of the next state function. The problem would be readily identified through simulation with the IFSMM.

First, consider the reconfiguration behavior associated with the Local Supervisor Model and what happens if a spare with error response responsibilities becomes faulty. If a spare that is assigned to respond to an error condition E(i) becomes faulty, then no fault-free node in the system will be assigned to respond to error condition E(i). Consequently, multiple errors involving a spare with error response assignment E(i) and the node in State i cannot be covered by the reconfiguration algorithm.

A related issue involves a "chain" of spares where each one in turn takes over error states. This problem is not as readily apparent. However, in developing the description of the next state function, the modeler would confront the issue of how the other spares in the system are affected when one spare takes over for a faulty cell. The problem of using a "chain" of spares to respond to error conditions is related to the problem of spares with error response responsibilities becoming faulty. Suppose a faulty cell creates an error condition to which a spare responds by taking over the faulty cell's activity. Now, the question arises "What happens if the cell which took over the faulty cell's activity becomes faulty?" There is no cell with that error response assignment! This issue is best illustrated by an example.

Consider the system shown in Figure 59. The basic graph is a linear array. There are three spare processors in the redundant graph and additional

(a) The Basic Graph




(b) The Redundant Graph

**Figure 59.** Linear Array Example Illustrating the Use of Spares to Respond to Errors

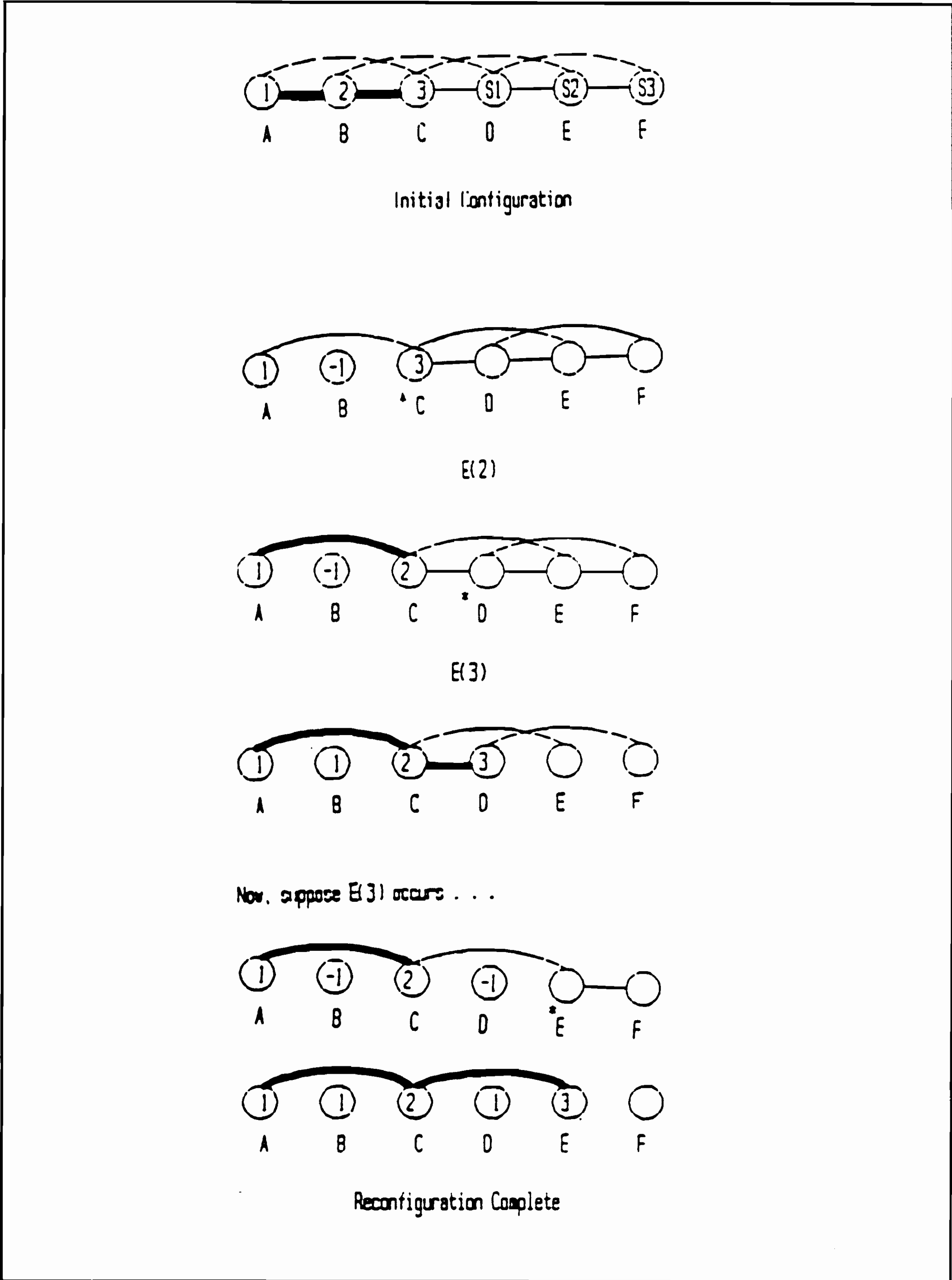



Figure 60. E(2)E(3) for Linear Array

connections between processors as shown. This simple example allows the description of the process of reconfiguration as it should take place without explaining how it is known what the "best" local actions are. If a node with one of the normal computation states 1, 2, or 3 becomes faulty, then each node to the east of the faulty node takes on the state of its closest, fault-free western neighbor. Consider the reconfiguration example shown in Figure 60. If node B becomes faulty, error E(2) exists in the system. In response, node C takes state 2 which in turn creates error E(3). To handle error E(3), node D, a spare, takes on state 3, and reconfiguration is complete. Then, if node D becomes faulty, error E(3) exists, and node E, a spare, responds by taking on state 3. Reconfiguration is complete.

In this manner, a "chain" of spare nodes is used to respond to errors. The Local Supervisor Model provides guidelines for determining error response assignments, unlike the Distributed Recovery Strategy. The need for a method of making error response assignments is even more evident in the special behavior that spares with error response responsibilities must have. The IFSMM identifies the behavior of cells explicitly in its next state function.

The issues involving spare cells must be addressed to allow the general use of spares in error response assignments. The suggested improvement for this problem is illustrated for the previous example in Figure 61. First, for each spare that has error response duties, a different state should be added to the error response digraph. States S1, S2, and S3 are added. For the spare

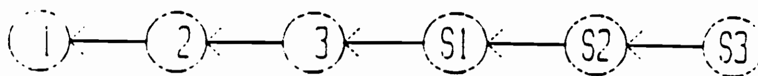



**Figure 61.** Error Response Digraph for Linear Array Example

which responds to a normal computation error condition (i.e. E(3) for the previous example), an arc from that spare state to the state it responds to will be added. So, an arc is added from S1 to 3.

When a "chain" of spares is involved and a spare responds to a normal computation error condition, the next spare in the chain then takes on that first spare's responsibilities. In the previous reconfiguration example (shown in Figure 60, when error E(3) occurred, Node D, a spare, took over state 3. At this point, the spare to the east of Node D, Node E, responded to a faulty node D which was state 3. In other words, for the initial configuration, the spare to the east of the node with state 3, responds to error E(3). Then, after reconfiguration around E(2), the spare to the east of the node with state 3, also responds to error E(3). So, we can see that the spares' responsibilities are passed to the east in the same way that the normal computation states are. Consequently, arrows are also added to the digraph from S2 to S1 and from S3 to S2. This part of the digraph indicates that spares maintain the same function relative to the basic graph.

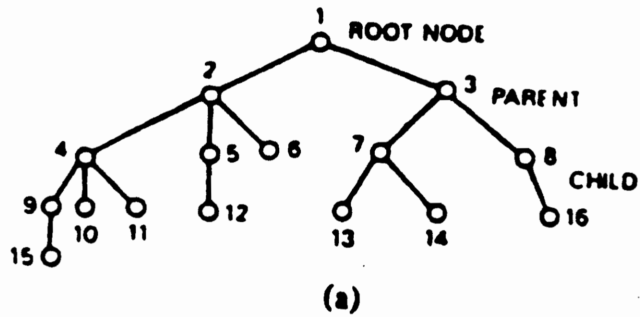
Notice that in this example, S3 does not have any arcs incident on it (its in-degree is zero). The constraints for error response assignments require that the in-degree for each vertex in the digraph be exactly 1. These constraints must be modified to incorporate the use of spares. We will still require that each vertex in the digraph corresponding to a normal computation state must have an in-degree of exactly 1. However, vertices in the digraph which

correspond to spare states will be allowed to have an in-degree of 1 or 0. These guidelines should be used in error response assignments to spares. The effects of these changes can be seen by considering the model's next state function.

This modification to the Local Supervisor Model allows more variety in the behavior of spare cells in the reconfiguration process. With this modification, the LSM can be used to model algorithms in which spares take an active role in the reconfiguration process.

Consider the 1-FT tree example in [Yan86]. The basic graph and the redundant graph from the paper is shown in Figure 62. The associated error response graph that would be used for this example is shown in Figure 63. Notice that the spare S responds to E(1). The assignment of the spare S to respond to E(1) was chosen in [Yan86]. This choice is a natural extension of the tree structure of the system. In the error response assignment in [Yan86], each error condition is responded to by the parent state in the basic graph. Having the spare respond to an error in the root state in the basic graph maintains the consistent upward flow of error response assignments.

Now, consider the 2-FT tree produced using the same procedure that is given in [Yan86]. The basic graph and the redundant graph are given in Figure 64. The associated error response digraph shown in Figure 65 is developed using the procedure discussed above. S2 takes the function of S1 when error E(S1) occurs. E(S1) can occur when the node with state S1



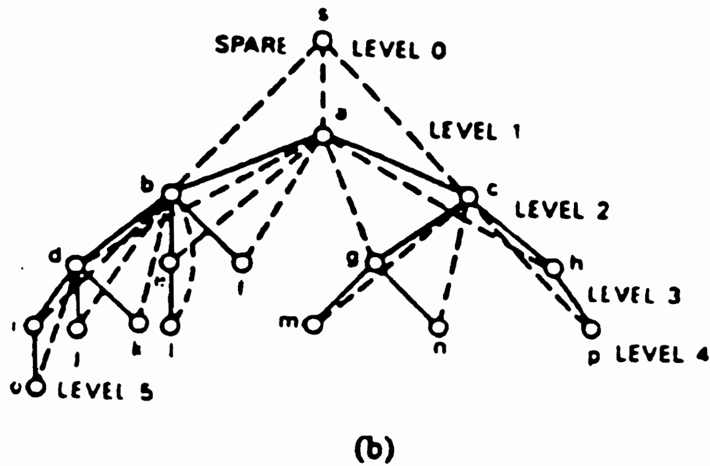




**Figure 62.** Graphs for 1-FT Tree System [Yan86]

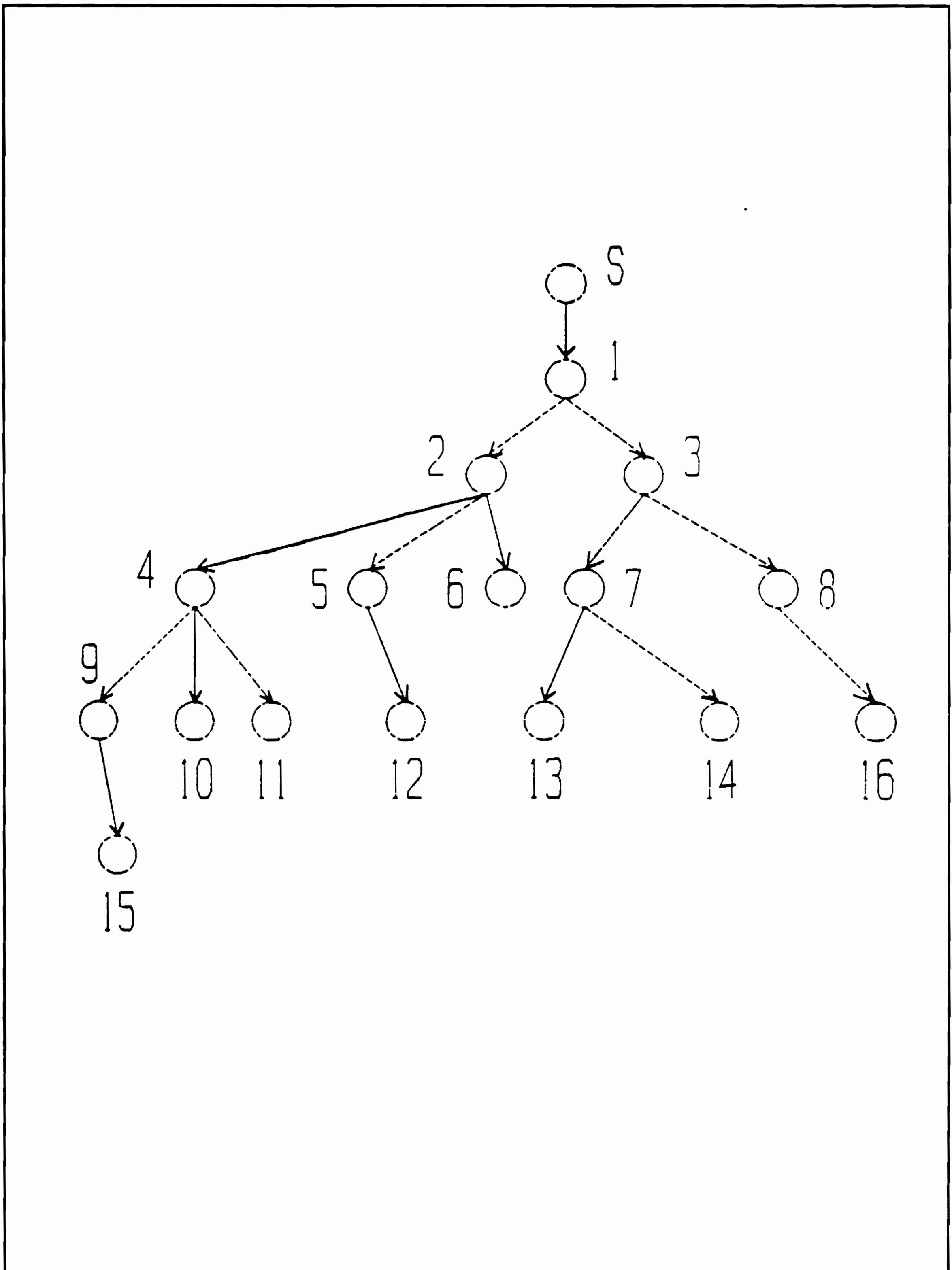
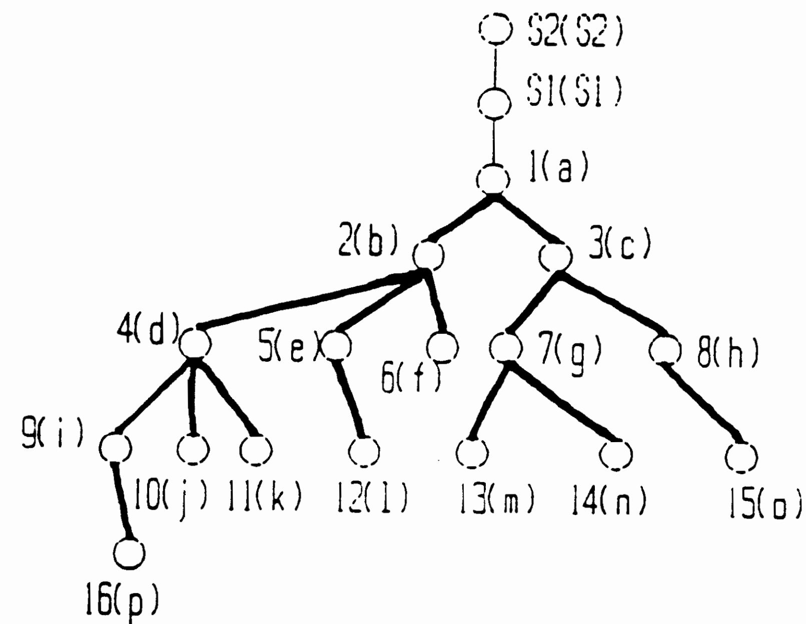



**Figure 63.** Error Response Digraph for 1-FT Tree System




Redundant Connections from each node at Level i to its descendents at level j ($j \leq i + k + 1$) for $0 \leq i \leq k + 4$.

Figure 64. Graphs for 2-FT Tree System

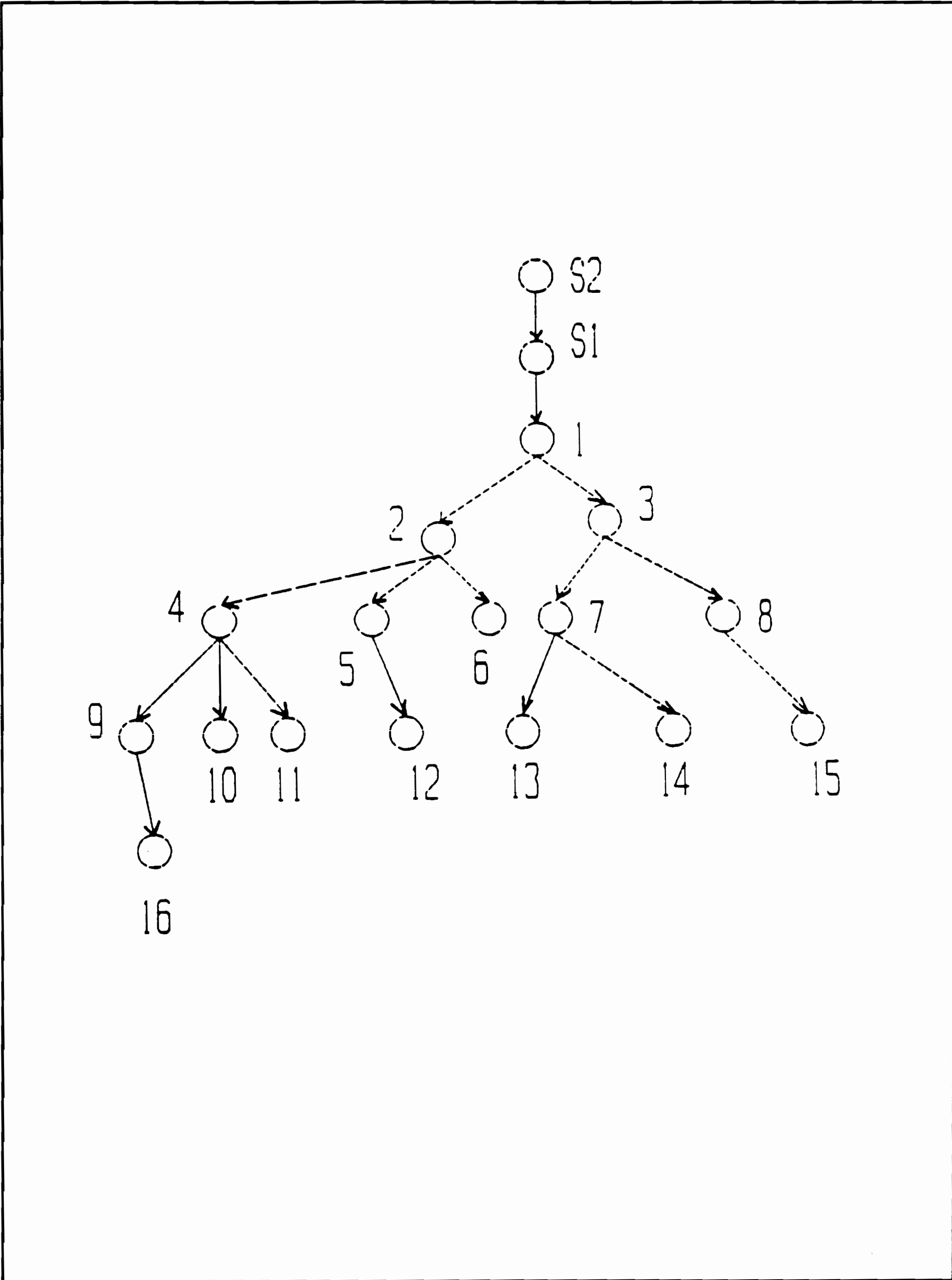


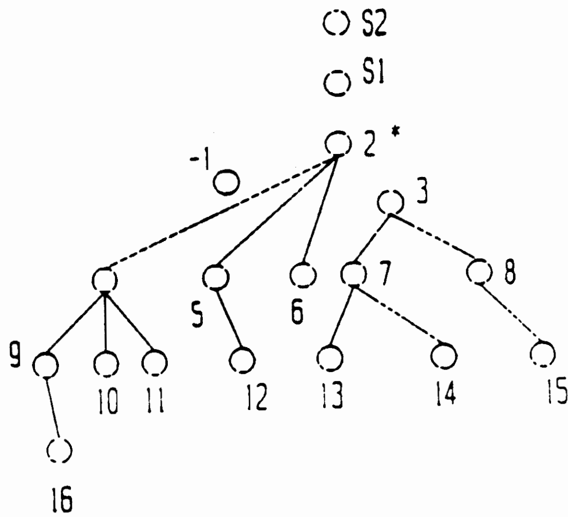

**Figure 65.** Error Response Digraph for 2-FT Tree System

becomes faulty or when the node with state S1 takes over state 1.

No node responds to E(S2). It is appropriate that error E(S2) does not require system reconfiguration, since the node with state S2 is the last spare to ever be used. Thus, if error E(S2) occurs, then the system is not affected. Also, after E(S2) occurs no node responds to E(S1). Similarly, if states S1 and S2 are both absent from the system, no node tests for error E(1). If no nodes have states S1 and S2, then there are not any spares in the system, so this action is appropriate.

Consider an example. The recovery process for node b (defined in Figure 64) being faulty is shown in Figure 66. A faulty cell is indicated by a state of -1. Error condition E(2) exists. Node a (defined in Figure 64) in state 1 responds by taking on state 2. This response creates error E(1). The node in state S1 responds and takes over state 1. This action removes state S1 from the system which creates error E(S1). The node in state S2 responds to E(S1) and takes on state S1. At this point, S2 is no longer present in the system and no action is taken to respond to error E(S2) at any future time.

In summary, the Distributed Recovery Strategy allows spares to be used to respond to error conditions, but it doesn't give any guidelines for how spares should be assigned to avoid difficulties. The Local Supervisor Model described in Chapter 2 was not conducive to the use of spares to respond to errors. This section has described how the Local Supervisor Model can be improved to use spares to respond to errors, has given guidelines for making error response



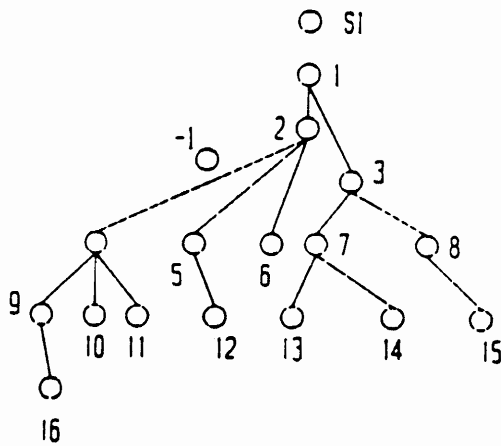




**Figure 66.** Recovery Process for 2-FT Tree System to Node b Faulty (E(1) exists)

assignments, and has identified some of the pitfalls inherent in the use of spares to respond to error conditions.

### 5.1.2 The Problem of Predicting an Algorithm's Coverage

For special types of graphs, such as complete graphs and trees, the coverage of an algorithm can be predicted using graph theory. If, however, the redundant and basic graphs are not of such a type, then how can the coverage be predicted? The IFSMM does not solve the problem of predicting coverage, although it can be a useful tool for simulations which can be used to help predict coverage. The problem of predicting coverage is not any worse for algorithms created using the Local Supervisor Model design approach than for any other algorithm. To minimize the problem of predicting coverage, the structure of the graph must be considered and used to help predict coverage without considering all error conditions.

The cycle and tree examples described in [Yan86] employ graphs in which graph theory can be used to help predict the algorithm's coverage. For the loop systems, the graph theoretical concepts of power graphs and Hamiltonian cycles are used in the analysis. Similarly, for the tree system, the properties of tree graphs are used.

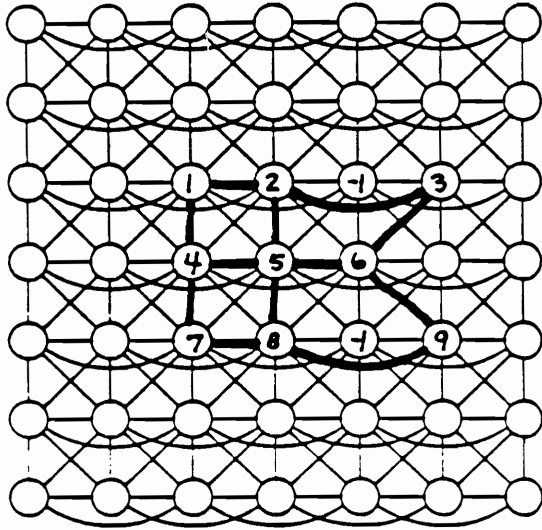
To predict the coverage of the White-Gray Local algorithm [Whi88], properties of the system's structure are used even though graph theory is not useful in the usual sense. In general, the structure of the system can be used in the analysis of coverage. Simulation results and proofs should then be used

to support the analysis. Coverage can also be determined through exhaustive simulation using the Level 3 IFSMM.

### 5.1.3 The Handling of Multiple and Near-Coincident Faults

To make the problems involved in reconfiguration tractable, the Distributed Recovery Strategy only allows one error to be present in the system at a given time. Algorithms developed using the Local Supervisor Model may or may not reconfigure correctly for multiple faults in the system because of the locality of knowledge in distributed systems. This section describes some attempts to solve the problem of handling multiple and near-coincident faults. The modifications presented do not adequately solve all aspects of the problems although they do present solutions which may improve the algorithms associated with the LSM. In addition, these modifications do allow the LSM reconfiguration behavior to be defined for multiple faults. This problem will be considered using the Array Reconfiguration Algorithm to demonstrate the results.

For the Array Reconfiguration Algorithm, some coincident and near-coincident faults are known to be able to be handled as "side-effects." In the example shown in Figure 67, reconfiguration around the two faults takes place with the same results as if they occurred in sequence. In other words, reconfiguration activities which do not affect any of the same nodes cannot interfere with each other. So, the distributed nature of the algorithm prevents the individual processors from knowing that more than one error is present in



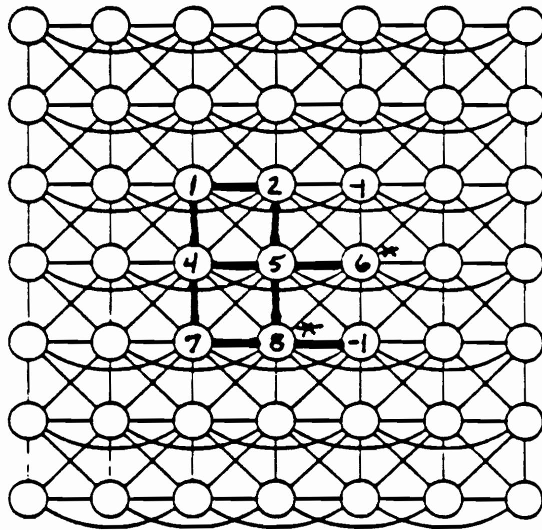




**Figure 67**. Array Reconfiguration Algorithm for E(3,9)

the system at the same time. Consequently, the reconfiguration process continues as if there were two separate errors. The results of the simultaneous reconfiguration in response to the two errors would then be the same as if one error occurred and then the other. The order of occurrence would not affect the final outcome either. However, the problem of what happens if two (or more) interacting faults occur should be addressed.

Consider interacting error conditions. Suppose E(1,4) occurs in the Array Reconfiguration Algorithm. This situation is shown in Figure 68. After errors E(1, 4) are detected, the nodes in states 2 and 5 respond. The node in state 2 responds to E(1) by looking for a spare that is adjacent to states 2 and 4. No spare in the neighborhood of the node in state 2 has all of these required adjacencies. The node in state 2 then checks to see if it has the required adjacencies (except state 2, its own state). However, the node in state 2 is not adjacent to a cell in state 4, so the node in state 2 takes no action in response to E(1). Thus, reconfiguration fails for E(1) when E(4) is also present.

Simultaneously, the node in state 5 responds to E(4). It looks for a spare that is adjacent to states 1, 5, and 7. But, since E(1) exists, no spare node can be adjacent to state 1. Similarly, the responding node itself cannot be adjacent to state 1. Thus, reconfiguration fails for E(4) when E(1) is also present in the system. In other words, since state 1 is absent from the system and since state 4 must be adjacent to state 1, no spare can be found to take over state 4. As described above, no spare can be found to take over state 1.

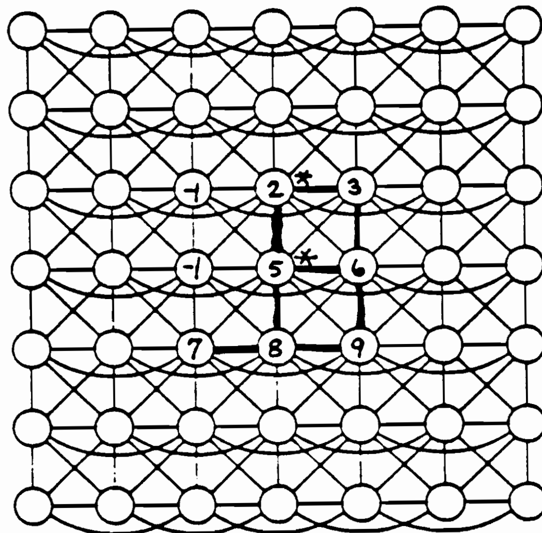
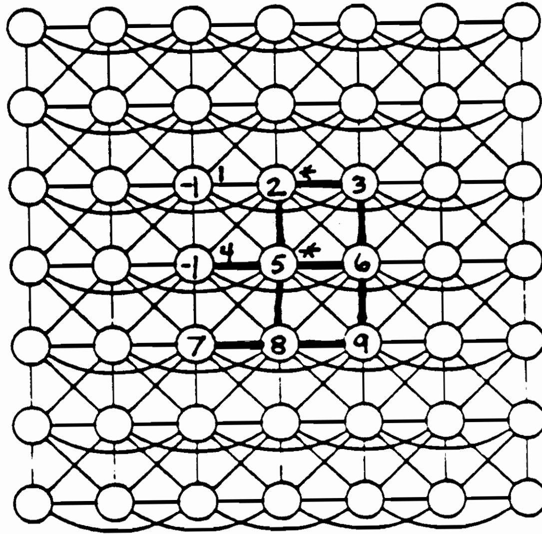



**Figure 68.** Array Reconfiguration Algorithm for E(1,4)--FAILS!

In each of the above cases, no appropriate spare was available and the responding node cannot take over either. So, a deadlock situation exists. Two proposed solutions which address this problem through modifications to the LSM were developed. The first modification has a shortcoming which the second remedies. Neither modification is guaranteed to improve the coverage of multiple faults; however, for specific cases there can be improvement.

The first modification to the problem requires some previous state information to be kept about neighbors. Consider the following modification to the LSM so that multiple error conditions can be handled. In addition to the neighbors' state information, the neighbor's previous state should also be kept. If the cell has only had one state (it has not changed states) then keep that information. This previous state information is indicated in the example by writing the previous state beside the node as shown in Figure 69. Only the state immediately previous to the present state is kept. Now, if reconfiguration fails, then consider each cell's previous state. Specifically, if a cell is lacking an adjacency, see if that adjacency was present during a previous state. If it was, then the cell should be allowed to takeover the error state. This modification may be applied to spares only or to current cell takeovers as well. These two instances of this proposed modification are described in Appendix I.

Consider the example shown in Figure 69 for the modification described above. The node in state 2 responds to E(1). No spare can be found using



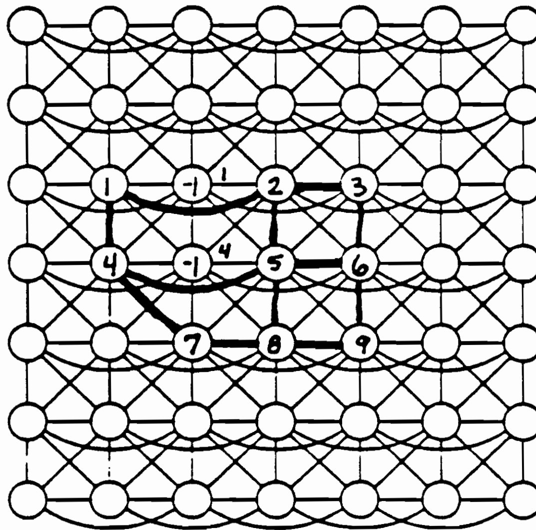




**Figure 69.** Reconfiguration for E(1,4) Showing Previous State Information

original approach as explained previously. The node in state 2 is also incapable of taking over state 1 under the original approach. Using the modified approach, state 2 then checks to see if any spare is available which had the required adjacencies (2 and 4) previously. A spare that meets these conditions is found to take over, and reconfiguration is complete.

The node in state 5 responds to E(4) at the same time that the node in state 2 responds to E(1). No spare is available with the required adjacencies, and the responding node doesn't have the required adjacencies either. Thus, the node in state 5 considers its neighbors' previous states. It finds that using this approach, there is a spare that is adjacent to states 5 and 7 and which was previously adjacent to a node in state 1. The spare takes over and reconfiguration is complete.

This technique worked well for this example, but in general it may not. It's principle shortcoming is that it may lead to disorderly behavior. In the general case more than one cell may be assigned the same state. This problem is illustrated in Figure 70. The basic graph is shown embedded in the redundant graph. The Error Response Digraph is also shown. Suppose, first a fault occurs and E(1) is created. The node in state 2 responds to E(1) and finds a spare with the required adjacencies. When the spare takes over, a valid configuration exists. Now, suppose a double fault occurs so that E(2,3) exists. The node in state 4 responds to E(3), and no node responds to E(2). The node in state 4 does not find any spare with the required adjacency to

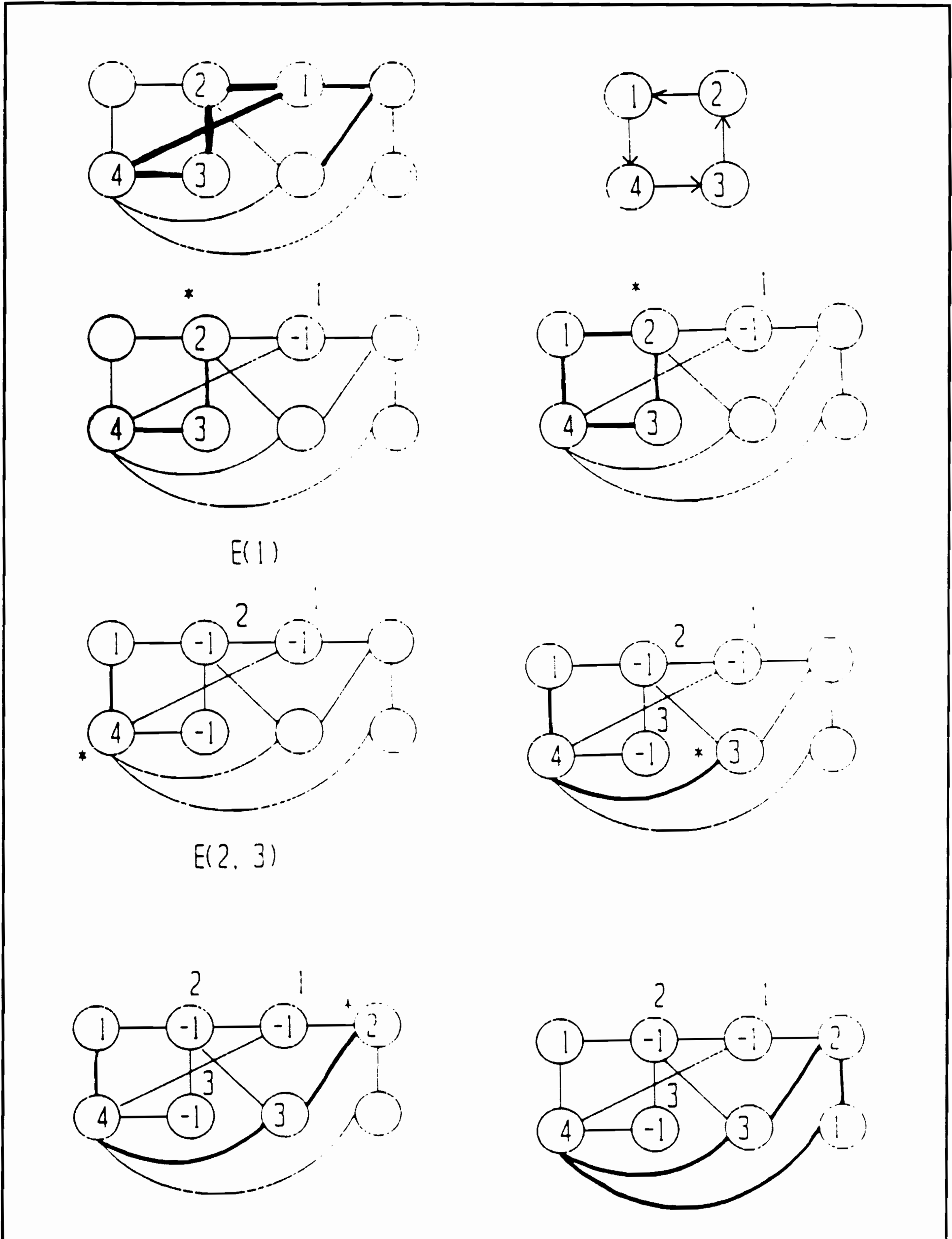



**Figure 70.** Example showing Problem where More than One Cell has the Same Normal Computation State

state 2 and the responding node does not have that adjacency either since state 2 is not present in the system. Then, following the approach outlined previously, the node in state 4 finds a spare that is adjacent to state 4 and is adjacent to a node that was previously in state 2. The node in state 4 assigns state 3 to this spare. The node in state 3 responds to E(2). Since there are no spares with the required adjacency to state 1 and since the responding node itself is not adjacent to state 1, the node in state 3 searches for a spare that meets the modified adjacency requirements. It finds a spare that is adjacent to state 3 and is adjacent to a cell that previously had state 1. Thus, the node in state 3 assigns state 2 to this spare. A valid configuration does not exist at this point even though all states are present in the system. The node in state 2 responds to E(1) since no node in its neighborhood has state 1. It finds a spare that does have all required adjacencies and assigns that spare state 1. At this point a valid configuration does exist in the system. And, by passing information between the cells, each cell can determine which connections should be activated. However, there are two cells with state 1. If at this point the node state 4 became faulty, both of the nodes in state 1 would respond and the behavior of the system becomes disorderly.

The problem of cells having the same state is a shortcoming in an algorithm which we would like to be well-behaved. However, by carefully assigning error response assignments for a particular system, this problem could be minimized and perhaps eliminated.

The second proposed solution to this problem uses a slightly different modification than that used in the first modification. Instead of allowing a spare with neighbors that previously had required adjacency states to take over when the algorithm fails, a spare can take over if it is missing an adjacency that is not an error response assignment and some neighbor previously had that required adjacency. In this way we can limit the disorderly behavior of the algorithm since no node can respond to an error if the error state is present in the system anywhere. This orderly behavior of my new modified approach is proven below.

THEOREM: Consider the modified LSM in which a cell may take on an error state if it is only missing adjacencies which are not in its error response assignments and which were immediately previously present in its neighborhood. If no two cells have the same computation state in the initial state of the system (nodes in $G_b$ are assigned distinct states), then no two cells will ever have the same computation state.

PROOF (By Induction):

**Base Case:** Time step = 0.

In the initial system, each computation state appears exactly once each. Thus, no two cells have the same computation state.

**Induction Hypothesis:** At time step n, no two cells have the same computation state.

**Induction Case:** Show that at time step n+1, no two cells have the same computation state.

Assume cell i and cell j both have state s. Let r denote the state assigned to respond to E(s). Then, one of the following cases must have occurred.

Case 1: One cell had state s at time step n and another cell took it time step n+1.

Let i denote the cell that had state s during time step n, and let j denote the cell that took state s this step. Cell j was either (a) a spare assigned to take over state s by a cell k with state r adjacent to cell j, or (b) cell j previously had state r and took over state s itself when it could not find an appropriate spare.

Case (a). Since cell k assigned state s to cell j, no cell in cell k's neighborhood had state s. Therefore, cell i must not be in cell k's neighborhood. However, cell i must be in the neighborhood of some cell in state r by restrictions on state assignment during reconfiguration. Specifically, either cell i has a cell in its neighborhood with state r or cell i had state r immediately before taking over state s. Thus, at time step n, there must have been two cells with state r. This contradicts the induction hypothesis that no two cells had the same state at time step n.

Case (b). Since cell j took over state s when it did not find an appropriate spare, no cell in cell j's neighborhood has state s. Therefore, cell i must not be in cell j's neighborhood, and by the same argument as above, there must have been two cells in the system with state r when cell j took over state s. This argument leads to the same contradiction as in Case (a).

Case 2: Two cells not previously in state s take state s during the current step (time step n+1).

If both cells were assigned state s at the same time, there must have been two cells with state r at that time which contradicts the initial assumption.

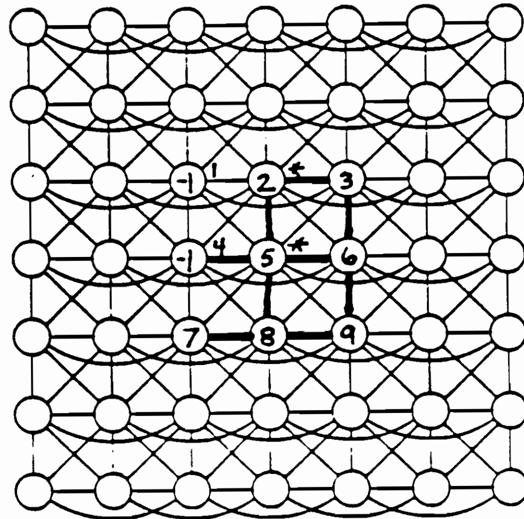
The important point in which the second suggested modification differs from the first modification is that a cell may take over an error state when that error state is currently in the system. Since the first modification allows a cell (responding node or spare node) to take over a missing state when that state is not present in the neighborhood of the responding node if the state was previously present. This important difference is indicated in the proof by the

statement that "cell i must be in the neighborhood of some cell in state r by restrictions on state assignment during reconfiguration."

The above proof has shown that the second suggested modification to the Local Supervisor Model is well-behaved and does not allow more than one cell in the system to have the same state.

Consider the previous example of reconfiguration around E(1,4). See Figure 71. The nodes in states 2 and 5 respond to the error conditions. The node in state 2 looks for a spare that is adjacent to states 2 and 7. No spares have these required adjacencies and the responding node does not either. So the responding node looks to see if there is a spare which has the adjacencies required by the modification. The modification requires that to take over an error state a node must be adjacent to nodes in all of the error state's error response set and to be adjacent to nodes which either have the other adjacencies or had the other adjacencies before their current states. In this case, the adjacencies required by the modification are state 2 or a node previously in state 2, and node 7 or a node previously in state 7. State 1 has no associated error response assignments. As before, a spare with these adjacencies is found.

At the same time, node 5 responds to E(4). No spares are available with the required adjacencies (states 1, 5, and 7), and the responding node does not have the required adjacencies either. The responding node then looks for a spare that meets the modified adjacency requirements. The modified adjacency



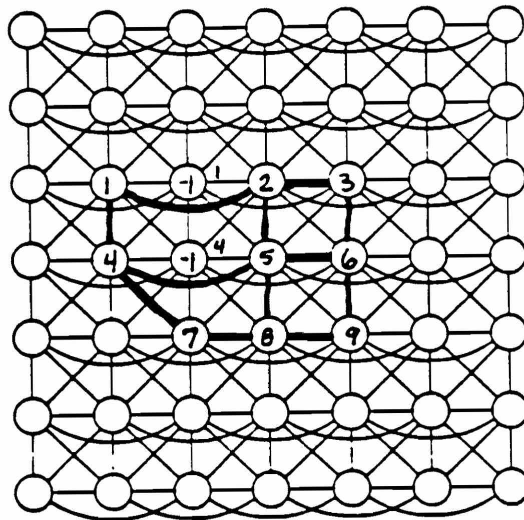




**Figure 71.** Example Showing Reconfiguration for E(1, 4) Using Second Modified Technique

requirements are state 1 or a node previously in state 1, state 5 or a node previously in state 5, and state 7 or a node previously in state 7. As before, a spare is found with these requirements. The spare takes over, and reconfiguration is complete.

For this example, this approach behaves as the previous modification. However, the disorderly behavior which results for some situations in which reconfiguration fails does not occur.

In summary, four modifications to the Array Reconfiguration Algorithm were considered. For the first modification, if a spare is not found with the required adjacencies and if the responding node cannot take over, then a spare which has the required adjacencies or had any missing adjacency as the state before the current state can take over. The second modification is like the first modification except that if a spare is not available with the relaxed adjacency requirements, then if the responding node has the relaxed adjacency requirements then it takes over the error state. The third modification to the original system provides that if no spare is found with the required adjacencies and if the current cell does not have the required adjacencies, then a spare which is adjacent to cells which have the states in the error response assignment for the missing state and if it has other required adjacencies or has neighbors that previously had those adjacencies can take over. The fourth modification is like the third modification except that if a spare with the modified adjacency requirements is not found, then the current cell takes over

| Double Fault | Original ARA | Modified ARA |
|---|---|---|
| 1,2 | fails | fails |
| 1,3 | 1 | 1 |
| 1,4 | fails | 1 |
| 1,5 | 2 | 2 |
| 1,6 | 1 | 1 |
| 1,7 | 1 | 1 |
| 1,8 | 2 | 2 |
| 1,9 | 1 | 1 |
| 2,3 | fails | fails |
| 2,4 | 2 | 2 |
| 2,5 | fails | fails |
| 2,6 | 4 | 4 |
| 2,7 | 2 | 2 |
| 2,8 | 2 | 2 |
| 2,9 | 2 | 2 |
| 3,4 | 1 | 1 |
| 3,5 | 3 | 3 |
| 3,6 | fails | fails |
| 3,7 | 1 | 1 |
| 3,8 | 2 | 2 |
| 3,9 | 1 | 1 |
| 4,5 | fails | fails |
| 4,6 | 1 | 1 |
| 4,7 | fails | 1 |
| 4,8 | 2 | 2 |
| 4,9 | 1 | 1 |
| 5,6 | fails | fails |
| 5,7 | 2 | 2 |
| 5,8 | fails | fails |
| 5,9 | fails | 2 |
| 6,7 | 1 | 1 |
| 6,8 | 3 | 3 |
| 6,9 | fails | fails |
| 7,8 | fails | fails |
| 7,9 | 2 | 2 |
| 8,9 | fails | fails |

**Figure 72.** Double Fault Coverage for Original and Modified Array Reconfiguration Algorithms

if it meets those modified requirements. Each of these different LSM modifications allows multiple faults to be in the system which is not allowed by the LSM. In addition, slightly better coverage of double faults occurs for the modified Array Reconfiguration Algorithm corresponding to each of the LSM modifications. The coverage of double faults is shown in Figure 72. All single faults are covered in each case. Additional information is provided in Appendix I.

### 5.1.4 The Implementation of a Local Supervisor Model Algorithm

The algorithm generated using the Local Supervisor Model may be implemented exactly as described, the way it is also simulated in the Level 3 IFSMM of the LSM Array algorithm. However, the algorithm generated by the LSM may be optimized by streamlining it, which will mean that it does not strictly follow the approach in the LSM which is the same as the Distributed Recovery Strategy. For example, the responding node can consider only the node it knows is in one of its error response assignment states.

In summary, after the desired behavior is identified, the implementation can be optimized. Using the Interconnected Finite State Machine Model will help in the identification of modifications which optimize the algorithm's implementation.

## 5.2 Improving the Direct Reconfiguration Algorithm

Through the use of the Interconnected Finite State Machine Model to describe the Direct Reconfiguration Algorithm, an important weakness in the algorithm was identified. To operate correctly, the combinational logic used to implement the algorithm must work correctly. Thus, this combinational logic is hard core. Through the use of the IFSMM, an improvement can be developed which eliminates the problem of hard core.

The major problem identified by developing the TAM and IFSMM for the Direct Reconfiguration Algorithm is the amount of hard core components in the system. To work properly, all of the added combinational logic must be fault-free. Although the added logic is simple, it spans the entire array. To solve the problem of hard core, the Level 3 IFSMM was used to isolate each cell's reconfiguration logic and make it specific to that cell. The Level 3 model was used to iteratively make changes, observe the improved algorithm's behavior, and determine new changes that need to be made. The result of using the IFSMM in this way is the algorithm described in this section. The final Level 3 IFSM model is given in Appendix J.

The Improved Direct Reconfiguration Algorithm uses the same basic principles of reconfiguration as the original Direct Reconfiguration Algorithm. Information is propagated through the array about the locations of faulty cells. In the Direct Reconfiguration Algorithm, the propagation takes place through

the combinational logic. In the Improved Reconfiguration Algorithm, the propagation occurs in a step-wise manner with each cell collecting information and then passing it to its neighbors. In the improved algorithm, several steps will be needed to pass the information that ripples asynchronously through the combinational logic in the Direct Reconfiguration Algorithm. However, the operations which occur at each time step are simple. Thus, the time steps can be short.

To develop the improvements to the Direct Reconfiguration Algorithm, simple equations describing the signals used in the original algorithm were developed. For example, the $y$ signal is set if either the $y$ output from the south neighbor or the $y$ signal from the far south neighbor is set. The definition for each of these signals is given in the Level 2 model description below. The equations are much simpler than the descriptions used in the IFSMM for the Direct Reconfiguration Algorithm. The hard core nature of the combinational logic is described in the IFSMM as a neighborhood which encompasses all of the cell in the array. Thus, the reduced neighborhood to be used in the Improved algorithm must be chosen. The neighborhood selected is the North, Far North, East, Far East, South, Far South, West, and Far West neighbors. This neighborhood includes cells that are at most two cells away. Thus, the neighborhood is reasonably small. However, the far neighbor connections reduces the number of steps required to transmit information through the array.

The iterative process of improving the algorithm using the Level 3 IFSMM required a few steps to determine the most important characteristics of the improved algorithm which have been described above. The model is so convenient for making changes and immediately observing their effects that it is troublesome to document the design process in detail. After a few iterations were used to define the major components of the Improved Direct Reconfiguration Algorithm, several more simple iterations were used to refine the algorithm and to correct logical errors.

The Improved Direct Reconfiguration Algorithm corresponds directly to the original Direct Reconfiguration Algorithm. The two IFSM models indicate the precise differences in the two algorithms. This section will describe the improved algorithm's IFSMM and identify the differences between the improved algorithm and the original algorithm.

Level 1 IFSMM for the Improved Direct Reconfiguration Algorithm

1-1 The facility graph for the improved algorithm is different from that of the original algorithm. Since the hard core is reduced, the neighborhood of each cell in the facility graph is smaller. The new facility graph is shown in Figure 73.

1-2 The set of active cells for the improved algorithm is identical to that for the original algorithm.

1-3 The set of boundary cells for the improved algorithm is identical to that for the original algorithm.

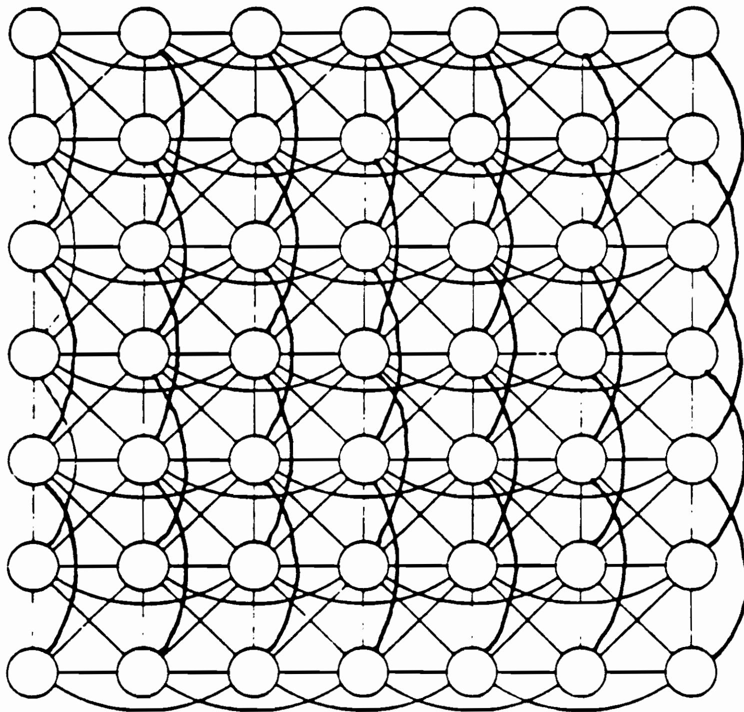


**Figure 73.** The Facility Graph for the Improved Direct Reconfiguration Algorithm

1-4 The information passed between nodes is different for the original algorithm and the improved algorithm. The improved algorithm includes the $y$ signal, the $y_0$ signal, the $e_x$ signal, and the $e_y$ signal in the information passed between cells, as well as, the logical state. The $y$ signal, the $y_0$ signal, the $e_x$ signal, and the $e_y$ signal are defined in the combinational logic for each cell as shown in Figure 48. The $y$ signal is true if there is a known fault in the cell or in the cell's column to the south. The $y_0$ signal is true if there is a known fault anywhere in the cell's column. The $e_x$ signal is true if the cell should undergo horizontal reconfiguration. The $e_y$ signal is true if the cell should undergo vertical reconfiguration. The $y$ signal is passed from south to north. The $y_0$ signal is passed both north to south and south to north. The $e_x$ signal is passed from west to east. The $e_y$ signal is passed from north to south. Since these signals propagate through the array, the values will change appropriately as additional information is transmitted throughout the array.

1-5 The information needed by the cell at each time step to determine the output information is also different than for the original algorithm. A cell not only needs to know whether its neighbors are faulty and its north, west, and northwest neighbors' states, but also needs to know the $y$ signal, the $y_0$ signal, the $e_x$ signal, and the $e_y$ signal.

Level 2 IFSMM for the Improved Direct Reconfiguration Algorithm

2-1 A cell changes its outputs to other cells whenever its logical state changes, as in the original algorithm's IFSMM. However, the neighborhood of the graph for the improved algorithm is much smaller than that of the original algorithm. In addition, it changes its outputs whenever the $y$ signal, the $y_0$ signal, the $e_x$ signal, or the $e_y$ signal changes.

2-2 To determine what its new outputs will be, a cell needs to know its north, west, and northwest neighbors' logical states and whether or not each of its neighbors are faulty, as in the original algorithm's IFSMM. Remember however, the neighborhood of the graph for the improved algorithm is much smaller than that of the original algorithm. In addition, a cell needs to know its neighbors' $y$ signal, $y_0$ signal, $e_x$ signal, and $e_y$ signal.

2-3 Pseudocode for output function:

```
/* Determine y */
if ((in[South].y = True) or (in[Far South].y = True) or (present_state.e =
True) or (present_state.y = True))
      y = True
else y = False

/* Determine y0 */
if ((in[South].y0 = True) or (in[Far South].y0 = True) or (in[North].y0 =
True) or (in[Far North].y0 = True) or (present_state.e = True) or
(present_state.y0 = True))
      y0 = True
else y0 = False
```

```
/* Determine ex */
if ((in[West].ex = True) or (in[Far West].ex = True) or ((present_state.e
= True) and (in[South].y = True)) or ((present_state.e = True) and
(in[Far South].y = True))
        ex = True
else ex = False

/* Determine ey */
if ((y0 = True) and (in[South].y = False))
        ey = True
else ey = False

for each output out[i]
        out[i].y = y
        out[i].y0 = y0
        out[i].ex = ex
        out[i].ey = ey
        out[i].state = present_state
```

2-4 Pseudocode for the next state function:

```
increment reconfiguration time counter
if (time to perform horizontal renaming)
        if (west neighbor sends ex = True)
                next_state = west neighbor's state
        else next_state = present_state
if (time to perform vertical renaming)
        if (north neighbor sends ey = True)
                next_state = north neighbor's state
        else next_state = present_state
if (not time for vertical renaming and not time for horizontal renaming)
        next_state = present_state
```

Level 3 IFSM Model for the Improved Direct Reconfiguration Algorithm

The Level 3 IFSMM is given in Appendix J. The Improved Direct Reconfiguration Algorithm was simulated for all single faults and selected multiple faults (including the example given in [Sam86]). In all cases, the final

configurations after reconfiguration are the same as for the Original Direct Reconfiguration Algorithm. Additional information is given in Appendix J.

The elimination of hard core is shown in the IFSMM by the reduction of the neighborhood of each cell to far north, north, far east, east, far south, south, far west, and west. In the improved algorithm, combinational logic spanning the system is not used. To eliminate the requirement of the original Direct Reconfiguration Algorithm that faulty cells must be able to receive, retain, and pass information, simple logic and multiplexors should be associated with cells so that information does not need to be passed through a faulty cell. Thus, this problem is solved through the implementation of the algorithm.

## 5.3 Summary

The Interconnected Finite State Machine Model is a valuable tool for improving existing reconfiguration algorithms and strategies. By modeling an algorithm or strategy using the IFSMM, its weaknesses and strengths become readily apparent. The model can then be used to overcome the weakness without losing the algorithm's advantages. The Level 3 model in particular is quite useful since it allows the designer to make a modification and quickly see its effects. The Local Supervisor Model has an associated reconfiguration strategy which is used to describe existing algorithms or design new algorithms. The IFSMM was used to simulate improvements to the Local

Supervisor Model's reconfiguration strategy. In addition, the IFSMM was used to improve the Direct Reconfiguration Algorithm.

During the development of the Local Supervisor Model, the need to further investigate several areas became apparent. The use of spares to respond to errors, the problem of predicting an algorithm's coverage, the handling of multiple and near-coincident faults, and the implementation of an algorithm designed using LSM were investigated. Although these problems were identified during the development process for the LSM itself, these problems could also be identified by the IFSMM modeling process in a similar way.

The IFSMM was useful in contemplating these problems; however, some of the problems only used the IFSMM to help understand the nature of the problem. The Local Supervisor Model was modified to include the use of spares in responding to errors. The use of spares is an expansion of the LSM reconfiguration strategy. The IFSMM does not help to solve the problem of predicting an algorithm's coverage; however, the problem is no more difficult for algorithms designed using the Local Supervisor Model than for algorithms in general. The Local Supervisor Model was also modified to allow multiple faults to be present in the system at a given time. The IFSMM was very useful in modifying the LSM to accommodate multiple faults. By using the Level 3 IFSMM for the Array Reconfiguration Algorithm presented in Chapter 2, various approaches could rapidly and easily be investigated. An Improved

Array Reconfiguration Algorithm is presented. The problem of determining a streamlined implementation of an algorithm designed using LSM cannot be easily solved. The implementation of each algorithm must be optimized on an individual basis. The IFSMM is independent of an algorithm's implementation.

By modeling the Direct Reconfiguration Algorithm with the IFSMM, a major defect was identified: the Direct Reconfiguration Algorithm requires that the combinational reconfiguration logic throughout the array be fault-free for correct reconfiguration to be guaranteed. This problem was identified by the Level 2 model. By using a Level 3 model to iteratively improve the Direct Reconfiguration Algorithm, the structure was modified to remove this hard core component. This hard core was eliminated by adding interconnection information corresponding to the signals generated using combinational logic in the original Direct Reconfiguration Algorithm. The Level 3 IFSMM was extremely useful in considering potential improvements.

# 6.0 Using the Interconnected Finite State Machine Model to Compare Distributed Algorithms

A structured comparison method is needed to systematically and completely compare distributed systems. The method should be hierarchical so that each comparison can be made using as little detail as possible. The three models presented, the LSM, the TAM, and the IFSMM, can all be used. This chapter describes the methodology for comparing distributed systems using the most general of the models, the IFSMM. Then, an example comparison is made between the White-Gray Local algorithm and the Array Reconfiguration Algorithm to illustrate the technique.

## 6.1 A Methodology for Comparing Distributed Systems

This section focuses on the methodology for comparing distributed systems using the IFSMM. The parameters of the model are directly used to identify the differences and similarities of distributed algorithms. Of course, a centralized algorithm can be considered as a special case. This section first

identifies important features that differentiate algorithms and then describes the procedure for comparing two systems.

### 6.1.1 Important Features that Differentiate Algorithms

This section discusses the parameters of the Interconnected Finite State Machine Model which are used in the comparison of systems. The parameters of the IFSMM are Q, A, B, I, G, $\delta$, $\lambda$, and $c_0$. Q is the set of states. The parameter A is the set of active cells, and B is the set of boundary cells. I is the set of interconnection values. G is the graph representing the system. $\delta$ is the next state function, and $\lambda$ is the output function. The initial state of the system is $c_0$.

The parameters A, B, and G define the topology of the redundant system. In addition, B and G also define boundary conditions. The information that is passed between nodes is defined by I, and the information that each node must have to determine outputs is defined by Q. This information associated with Q is related to the requirements for each node. The initial state of the entire system is given by $c_0$. The next state function, $\delta$, and the output function, $\lambda$, identify <u>how</u> the algorithm works. Specifically, $\delta$ identifies <u>how</u> each node moves from one state to another. Similarly, $\lambda$ identifies <u>how</u> the outputs of a node are determined. Each of these parameters identifies an aspect of distributed systems which should be considered in making comparisons.

One potential problem associated with using this model in the comparison of systems is the fact that coverage information is not readily apparent. Also, whether the algorithm will terminate (or loop infinitely) is not readily apparent. This information is buried within $\delta$ and $\lambda$. Coverage and termination would typically be compared using the Level 3 IFSMM.

The IFSMM of an algorithm identifies the essence of that algorithm. The boundary conditions of the redundant system are clearly identified and removed from the analysis of the system. Thus, special boundary cases do not need to be considered. The IFSMM also clearly specifies the redundant system's topology. All information passed between nodes is identified, and the communication complexity of the interconnections is determined. All information which must be stored locally in a cell is also identified, and the local memory requirements for each node are thus determined.

The next state function, in conjunction with the set of states, describes the behavior of the algorithm from time step to time step. The complexity of the next state function, along with the complexity of the output function, determines the computing power required at each node. It also indicates the complexity of the computations that need to be made--which is related to the time required to complete the computations.

The output function, in conjunction with the interconnection information, I, specifies how the algorithm determines a node's outputs to its neighbors. The complexity of the output function, along with the complexity of the next

state function, determines the computing power required at each node. Further, it indicates the complexity of the computations that need to be made--which is related to the time required to complete the computations.

The reason for separating the output and next state functions is evident in the comparison of algorithms. The separation forces the modeler to distinguish between output information and state information. This distinction leads to a better definition of the interconnection communication requirements and the local state information requirements. In addition, it identifies more concretely how the algorithms work. By having a better grasp of these aspects, the comparison of different algorithms is facilitated.

Algorithms may affect the same behavior from the system with only the implementation of the algorithms differing. The separation of the next state function and the output function also makes model development easier. It is difficult to conceptually place information about the outputs of a function in the state information. In addition, it more readily allows the node to output different information to each of its neighbors.

### 6.1.2 Procedure for Comparing Systems

The procedure for comparing systems, like the procedure for modeling systems, is hierarchical. After comparing the systems at a high level, if additional comparison information is needed, the comparison process is performed at a lower level.

To compare two systems at Level 1, a Level 1 model for each of the systems should be developed. The facility graphs developed in Step 1-1 represent the redundant hardware systems for the two algorithms being considered. The set of active cells (Step 1-2) for each algorithm should also be compared. If the facility graphs and the active cells are the same for each algorithm, then the two algorithms are appropriate for the same hardware structure. The differences in active cells and the facility graphs indicate differences in the hardware systems appropriate for each algorithm. The set of boundary cells (Step 1-3) defines the boundary information needed by a system. Since there is no single choice of boundary cells for a given algorithm, this comparison must be qualitative. The graph encompassing the active cells, the boundary cells, the redundant hardware system, and the boundary cell connections is the graph G in the IFSMM. By comparing G for the algorithms, the differences in required hardware for each algorithm is identified. By comparing the set of interconnection values, I, for the algorithms (Step 1-4), the differences in communication requirements for each algorithm are determined. The set of cell states, Q, consists of the information needed by a cell at each time step to determine the output information (Step 1-5). The state information and the inputs to a cell are used to determine the cell's outputs. Thus, the set of cell states for an algorithm is related to the local storage required by each cell.

To compare the algorithms at Level 2, a Level 2 model for each of the systems should be developed. At Step 2-1, all aspects of the algorithm in which a cell changes its outputs to other cells are identified. This step is the preliminary step to defining the output function, $\lambda$. In Step 2-3, pseudocode for the output function is developed. Similarly, Steps 2-2 and 2-4 define the next state function, $\delta$. The information identified in these steps defines the output function. By comparing the output functions and the next state functions at this level, the hearts of the algorithms are identified, and essential differences are found.

To compare the algorithms at Level 3, a Level 3 model for each of the systems should be developed. Level 3 comparison is the most comprehensive of the comparison processes. The simulation models developed can be used to evaluate various aspects of the algorithms' performances. Simulations can be used to show in detail how each algorithm accomplishes its goals. In addition, coverage and timing requirements can be compared. Finally, the actual outcomes of invoking the algorithm for different situations can be compared to determine whether or not the algorithms are more alike than expected.

## 6.2 Comparison of the White-Gray Algorithm and the Array Reconfiguration Algorithm

This section uses the comparison of the White-Gray Local algorithm and the Array Reconfiguration Algorithm to illustrate the comparison process using

the Interconnected Finite State Machine Model. First, the Level 1 comparison is made. Then, the Level 2 comparison is made. Finally, the detailed Level 3 comparison is performed. The Level 3 comparison shows how the model can be used to relate one algorithm to another.

### 6.2.1 Level 1 Comparison

The Level 1 IFSM Models for the White-Gray Local algorithm and the Array Reconfiguration Algorithm are given in Chapter 4. The key information is summarized here for convenience. First, the IFSMM for the White-Gray Local algorithm is described. Then, the model for the Array Reconfiguration Algorithm is presented.

The Interconnected Finite State Machine Model for the White-Gray Local algorithm is outlined as follows. Steps 1-1 through 1-3 define the graph for the IFSMM. Figure 56 shows the graph for the model of the White-Gray Local algorithm. Step 1-4 identifies the information passed between cells as the fault register contents (10 bits). Step 1-5 identifies a cell's state information to be its current fault register, its active connections, its own computation state, its logical west neighbor's computation state, test results for all its active neighbors, and a signal indicating whether it should take its west neighbor's state.

The Interconnected Finite State Machine Model for the Array Reconfiguration Algorithm can also be outlined. Steps 1-1 through 1-3 define the graph for the IFSMM. This graph is the same as for the White-Gray Local

algorithm (see Figure 56). The information passed between cells (Step 1-4) is the cell's logical state, the cell's neighbors' logical states, and a signal indicating whether a spare should take on a specified state. Step 1-5 identifies the state information for a cell to be a set of error conditions to which it responds, its own logical state, its neighbors' logical states, and its neighbors' neighbors' logical states.

After Level 1 IFSM Models have been developed for each algorithm, some comparisons should be made. First consider A, B, and G, then consider I and Q. Since the same redundant system is used for each of the algorithms, A is the same for each. For each algorithm, all boundary cells behave in the same way, so only one boundary cell is needed in each case. Thus, B is identical for both algorithms. Since the structures of the redundant systems were chosen to be the same for both algorithms, and since the boundary structure is also the same, G is the same for both algorithms.

The comparison of I for the two algorithms should also be performed in detail. For the White-Gray Local algorithm, the interconnection information consists of 10-bits (the fault register). In addition, the cell must know the logical state information required for rollback and the take over requirements; however, this information will be essentially the same for both algorithms and can thus be neglected. For the Array Reconfiguration Algorithm, more information must be passed between cells. For the Array Reconfiguration Algorithm, the logical state of a cell also indicates whether self-testing reveals

the cell as faulty. If $a$ bits are required to encode the logical state and each cell has $p$ neighbors (not counting itself), then this requires $a(p+1)$ bits. Since each cell has 10 neighbors, $p$ = 10. Since there are 9 normal computation states, a spare state, and a faulty state, 4 bits are needed to encode a cell's logical state. Thus, $a$ = 4. Thus,

$$a(p+1) = 4(10+1) = 44 \text{ bits}$$

are needed to pass the required information. Only 1 bit, along with appropriate rollback information, is needed to indicate to a spare that it should take over a particular state. This single bit tells the spare that valid rollback information is available. Rollback information is also needed if a cell that is not a spare is required take over a state.

The comparison of Q for the two algorithms can also be performed in detail. The state information required by the White-Gray Local algorithm is

- the fault register--10 bits
- the active connections--no bits are required since this can be decoded from the fault register
- test results for all active neighbors--pass/fail; 4 bits
- computation state and west neighbor's computation state (required for rollback information)

Thus, 14 bits of storage are required, plus rollback information.

For the Array Reconfiguration Algorithm, more storage is required. The error conditions to which the cell responds requires

(NUMBER OF CONDITIONS) * (NUMBER OF BITS REQUIRED FOR LOGICAL STATE)

= (NUMBER OF CONDITIONS) * $a$ = 2 * 4 = 8 bits

Information about the required adjacencies for the computation states in the graph must also be maintained. There are 9 computation states. The maximum number of bits required is 9 * 4 = 36, since each computation state can have at most 4 required adjacencies. However, because of boundary conditions, state $S_{2,2}$ requires 4, states $S_{1,2}$, $S_{2,1}$, $S_{2,3}$, and $S_{3,2}$ require 3, and states $S_{1,1}$, $S_{1,3}$, $S_{3,1}$, and $S_{3,3}$ require 2. Thus, for this implementation, 27 bits are required. To be more general, the maximum number of bits required would be the number of computation states multiplied by four. In addition, a cell must know its own logical state ($a$ bits = 4 bits) and its neighbors' logical states ($a$ * $p$ bits = 4 * 10 bits = 40 bits). Thus, the total number of bits required for the state information of the Array Reconfiguration Algorithm is 8 + 4 + 40 bits = 52 bits. As mentioned previously, additional information is needed for rollback.

In summary, the Level 1 comparison indicates that the Array Reconfiguration Algorithm is more complex than the White-Gray Local algorithm. The two algorithms use the same hardware structure. However, the White-Gray Local algorithm requires 10 bits to be passed between cells, while the Array Reconfiguration Algorithm requires 44 bits to be passed. The state information is also more complex for the Array Reconfiguration Algorithm. The White-Gray Local algorithm requires 14 bits of state information, while the Array Reconfiguration Algorithm requires 52 bits.

### 6.2.2 Level 2 Comparison

If more comparison information is needed after the Level 1 comparison is complete, Level 2 IFSM Models of the two algorithms should be developed. Level 2 models for the White-Gray Local Algorithm and the Array Reconfiguration Algorithm are given in Chapter 4. Studying the pseudocode developed for the Level 2 models will lead to the understanding that the Array Reconfiguration Algorithm is more complicated than the White-Gray Local algorithm since it has a more complex decision process to determine how reconfiguration will take place. However, it is also evident that the Array Reconfiguration Algorithm is more flexible than the White-Gray Local algorithm since the Array Reconfiguration Algorithm decision process can be easily changed by modifying the error response assignments. This part of the comparison process is best suited to drawing general conclusions about the relative complexities of the algorithms and to a general understanding of how the algorithms work.

### 6.2.3 Level 3 Comparison

After Level 1 and Level 2 comparisons have been completed, additional areas for further comparison may become evident. Level 3 IFSM Models for each algorithm can be developed. These Level 3 models allow simulation of the algorithms. By simulating each algorithm, measures such as coverage and time requirements for reconfiguration can be determined and compared. Level 3 IFSM Models for each of the algorithms are discussed in Chapter 4. Timing

and coverage information for each of the algorithms is given in Chapter 2 where the Array Reconfiguration Algorithm is first discussed.

In comparing the White-Gray Local algorithm and the Array Reconfiguration Algorithm at Levels 1 and 2, significant similarities became evident. Most noticeably, the graphs for each of these algorithms are identical. The strong similarity of the algorithms leads to the question of how closely the two algorithms are related. The graphs were chosen to be the same to illustrate the use of the LSM to develop an algorithm; perhaps the application of the LSM yielded an algorithm that is essentially the same as the White-Gray algorithm. To consider this question, a slightly different approach in comparing the two systems was adopted. The Level 3 IFSMM for the Array Reconfiguration Algorithm was used to determine the similarities and differences of the two algorithms by making modifications to the model so that it exactly mimics the behavior of the White-Gray Local algorithm. To duplicate the behavior of the White-Gray Local algorithm, the fourth modification to the Array Reconfiguration Algorithm presented in Chapter 4 is used so that (coincident) multiple faults can be handled.

Since the White-Gray Local algorithm only allows reconfiguration to take place through a shifting to the east, an error response assignment for the Array Reconfiguration Algorithm in which reconfiguration causes a shift to the east if possible was chosen. The error response digraph associated with this assignment is given in Figure 74. This error response assignment includes the

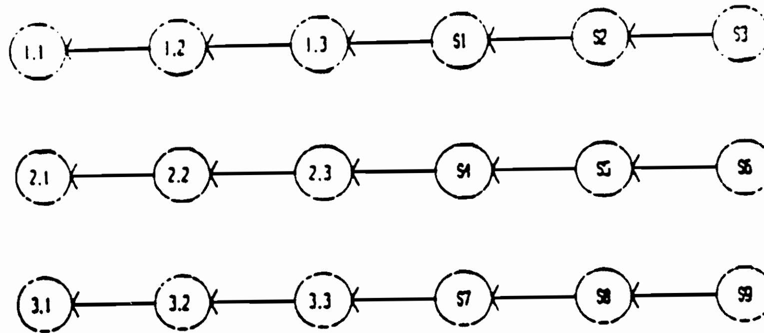



**Figure 74.** The Error Response Digraph for the Array Reconfiguration Algorithm which Mimics the White-Gray Local Algorithm

use of spares as described in Chapter 5. This error response digraph uses a modified version of the LSM (modification 4) which is developed in Chapter 5. In addition, the basic graph for the Array Reconfiguration Algorithm with the new error response assignment is embedded in the northwest-most corner of the redundant graph. Details of the modified Level 3 IFSMM are given in Appendix K. The only required modification to the C routines is a limitation that only spares to the east and far east are considered.

The result of the simulation shows that the behavior of the Array Reconfiguration Algorithm with the new error response assignment exactly mimics the White-Gray Local algorithm. The coverage and time requirements for the new Array Reconfiguration Algorithm are summarized in Figure 75. This information is identical to that stated for the White-Gray Local algorithm in [Gra88]. This modification of the IFSMM for the Array Reconfiguration Algorithm shows that the White-Gray Local algorithm and the Array Reconfiguration Algorithm are very similar structurally. Only very simple changes were required to the IFSMM for the original Array Reconfiguration Algorithm (see Appendix K).

The Level 3 IFSM Models for the original Array Reconfiguration Algorithm and the White-Gray Local algorithm indicate that although the original Array Reconfiguration Algorithm requires fewer time steps to reconfigure than the White-Gray Local algorithm in many cases, each step requires more actual time. Thus, the Array Reconfiguration Algorithm will

| Double Fault | Time Steps |
|---|---|
| 1,2 | fails |
| 1,3 | fails |
| 1,4 | 3 |
| 1,5 | 3 |
| 1,6 | 3 |
| 1,7 | 3 |
| 1,8 | 3 |
| 1,9 | 3 |
| 2,3 | fails |
| 2,4 | 3 |
| 2,5 | 2 |
| 2,6 | 2 |
| 2,7 | 3 |
| 2,8 | 2 |
| 2,9 | 2 |
| 3,4 | 3 |
| 3,5 | 2 |
| 3,6 | 1 |
| 3,7 | 3 |
| 3,8 | 2 |
| 3,9 | 1 |
| 4,5 | fails |
| 4,6 | fails |
| 4,7 | 3 |
| 4,8 | 3 |
| 4,9 | 3 |
| 5,6 | fails |
| 5,7 | 3 |
| 5,8 | 2 |
| 5,9 | 2 |
| 6,7 | 3 |
| 6,8 | 2 |
| 6,9 | 1 |
| 7,8 | fails |
| 7,9 | fails |
| 8,9 | fails |

**Figure 75.** Coverage and Time Requirements for the Array Reconfiguration Algorithm Modified to Mimic the White-Gray Local Algorithm

require more time to reconfigure.

## 6.3 Summary

The Interconnected Finite State Machine Model is a tool which provides a structured, systematic method for completely comparing distributed systems. The IFSM Modeling process has three levels, allowing the hierarchical comparison of systems. Thus, only as much detail as necessary is used in the comparison process. The key differences and similarities between algorithms can be quickly identified at a high level with more specific comparisons being performed (if desired) at lower levels. The IFSMM comparison process specifies the procedure for identifying similarities and differences.

The IFSMM was used to compare the White-Gray Local algorithm and the Array Reconfiguration Algorithm. This comparison illustrates an important capability of the IFSMM comparison process to determine how algorithms are similar and to identify their essential differences.

The Level 1 comparison process indicates that the Array Reconfiguration Algorithm is more complex than the White-Gray algorithm. Both algorithms use the same redundant graph structure.

The Level 2 comparison confirms the Level 1 analysis that the Array Reconfiguration Algorithm is more complex than the White-Gray algorithm; however, it also determines that the Array Reconfiguration Algorithm is more

flexible than the White-Gray Local algorithm. The process of performing a Level 2 analysis also helps to clarify how the algorithms work.

In general, a Level 3 comparison would consist of developing Level 3 models for each algorithm and analyzing performance measures, such as coverage. However, since the two algorithms are based on the same redundant structure, the Level 3 comparison was also used to show that the White-Gray Local algorithm behaves in a way very similar to the Array Reconfiguration Algorithm. By using the fourth modification to the Local Supervisor Model described in Chapter 5, an array algorithm is developed that exactly mimics the White-Gray Local Algorithm. The Level 3 comparison not only illustrates the use of the IFSMM in comparing algorithms, but also shows how the modified Local Supervisor Model can be used to model existing algorithms through the use of the Array Reconfiguration Algorithm to mimic the White-Gray Local algorithm.

# 7.0 Using the Interconnected Finite State Machine Model to Design a Distributed Algorithm for a Particular Application

This chapter addresses the use of the Interconnected Finite State Machine Model in the design process. The model and its associated design procedure provide a useful tool for designing reconfigurable systems. Without a consistent, systematic plan for developing a design, the process would be haphazard and might not yield good results. This section describes the IFSMM design process and describes is application to a selected problem.

## 7.1 The Design Procedure

The design procedure used with the Interconnected Finite State Machine Model is a procedure for developing a reconfigurable architecture for a given application. The IFSMM model is used in several different ways throughout the design process. The design procedure consists of the following steps: determining the redundant architecture, defining the reconfiguration algorithm behavior, and defining implementation details of the reconfiguration algorithm.

Determining the redundant architecture is a process in which alternatives are generated and then the Interconnected Finite State Machine Model is used to narrow the list of alternatives to a single design. At each step in the design process, items are added to the list of alternative architectures and other items are pruned from the list. To determine the redundant architecture, basic architecture candidates are chosen, ways of adding redundant processors and links are generated, and the IFSMM is used to analyze the alternative architectures.

First, the candidates for the basic architecture must be chosen. Often, designers choose a final basic architecture with total disregard for fault tolerance. Fault tolerance should be considered in making the choice of basic architecture. The designer should identify all appropriate candidate architectures and order them by preference. In this way, fault tolerant capabilities can be considered along with other system requirements.

After the basic architecture candidates are identified, the process of generating potential redundant architectures begins. The architecture candidates should be ordered from most promising to least promising based on general knowledge of the application and the architectures. As new advantages and disadvantages of the architectures become apparent, the list of basic architecture candidates should be reordered appropriately. Starting with the most promising architecture, redundant processors and links should be added. Then, the designer should consider the next most promising

architecture. The procedure should be applied to each candidate on the list. To determine how redundant elements should be added to an architecture, the designer should brainstorm for ideas. It is important not to evaluate ideas when brainstorming, since it tends to inhibit creativity. This process should be applied to each candidate architecture. If at any point an architecture ceases to look promising, the designer should proceed to the next architecture. In this way, time spent considering basic architectures which will not be used can be minimized. In addition, if an architecture is clearly the best choice, then consideration of other architectures should end. This step will yield a host of potential redundant architectures. During the addition of redundant elements the designer should always consider the basic architecture's underlying characteristics. Also, the designer should consider known redundant architectures and reconfiguration techniques for the basic architecture. The review of known techniques will help generate new ideas. In addition, a known technique might be adequate or can be used with small modifications.

After generating candidate redundant architectures, the Interconnected Finite State Machine Model should be used to analyze them. To analyze architectures, design restrictions and goals must be stated clearly. Many design restrictions and goals can be stated directly in terms of the IFSMM. Then, the IFSMM should be used to evaluate candidate architectures and eliminate any which do not meet the goals and restrictions. The evaluation

process should be used to yield a single redundant architecture for the reconfigurable system that is appropriate to the problem.

During the process of evaluating and eliminating redundant architecture alternatives, the basic structure of reconfiguration algorithms will be developed. After the best redundant architecture is chosen, the details of the reconfiguration algorithm's behavior must be specified. The Interconnected Finite State Machine Model should be used to specify the algorithm's behavior clearly. In addition, several example reconfiguration processes should be performed by hand to identify any difficulties with the algorithm. A Level 3 IFSMM can be developed at this time. Coverage and reconfiguration time requirements can be determined through simulation with the Level 3 IFSMM.

After the basic reconfiguration behavior is defined, the implementation details of the reconfiguration algorithm should be specified. For example, restrictions on spare availability may be set to simplify the reconfiguration process. By applying this technique to the Array Reconfiguration Algorithm discussed in previous chapters, an algorithm similar to the White-Gray Local algorithm could result. This streamlining of the implementation is not a well-specified process.

## 7.2 The Design Procedure Applied to a Specific Problem

The design procedure described above is suitable for use with any architecture which can be modeled using the IFSMM. This section describes

the development of a reconfigurable system for a particular application. This example illustrates the most important aspects of the IFSMM design process.

This example will assume that the application requires a hypercube architecture. In addition, each node must be a reasonably powerful processor; so nodes are relatively expensive. High availability, as well as high reliability, is desired. The degree of each node should be small, and the links should be short. Many of these requirements conflict with one another, so they must be balanced. A more specific set of requirements must be developed. The list of design constraints and goals will be developed in the next section based on these outlined requirements.

The design process specified previously for developing a redundant architecture will be applied to the example application in this section. Candidate architectures will be developed as described above. Then, the candidate architecture list will be pruned to yield a selected redundant structure. Finally, the details of the reconfiguration algorithm will be specified.

### 7.2.1 Development of the Candidate Architecture List

This example assumes that the hypercube is the most promising by far of all candidate basic architectures. Thus, this analysis will consider only the hypercube. If there is no practical way to design a reconfigurable hypercube, other candidate architectures must be considered.

Brainstorming for methods of adding redundant elements and links yields the following possibilities:

1. spare(s) per subcube of some size $2^i$ for $i = 0, 1, 2, \ldots$ with connections to each cell in the subcube
2. spare(s) per subcube of some size with connections to some subset of processors in the subcube
3. subcubes as spares instead of single processor spares
4. some larger general-type architecture in which the cube is embedded (e.g. ternary cubes)
5. extra column(s) with far east/far west connections (extra row(s) is similar)

Approaches in this list will be considered in detail.

### 7.2.2 Design Constraints and Goals for the Chosen Application

Before the candidate redundant architectures can be evaluated, design goals and limits must be specified. Some aspects to consider are the level of redundancy to be used, the maximum acceptable link length, and the maximum acceptable degree of each node. The level of redundancy is important because of cost; processor complexity must be considered when assigning a maximum level of redundancy.

The following design constraints and goals are set. The constraints and goals are based on the architecture requirements stated previously. Many of

these goals and limitations relate directly to the IFSMM. The design constraints and goals must be determined for each application.

- Assume nodes are relatively expensive. So, choose a maximum processor redundancy of 25%.
- The basic graph is the hypercube. For this example, the $2^4$ cube will be considered.
- Highly available system is desired.
- Highly reliable system is desired. Would like to cover all single and double faults.
- Want to keep link lengths as short as possible.
- Want to keep degree of each node at a reasonable value (less than 12). Avoid complex switching mechanisms because they lower reliability and increase expense.
- Need a simple algorithm to reduce the reconfiguration time required. This reduction in reconfiguration time increases the availability of the system.

These design constraints and goals relate directly to the model. The level of redundancy is the number of spare nodes divided by the number of computation nodes. The set of active nodes in the model consists of all spare nodes and computation nodes. The degree of each node is p in the model's terminology. The simplicity of the algorithm can be determined by considering

the simplicity of the functions $\lambda$ and $\delta$ of the model. Many of these constraints and goals are difficult to quantify with actual values.

### 7.2.3 The Choice of a Redundant Structure

After design constraints and goals are specified, the candidate redundant architectures should be evaluated with respect to them. The following discussion considers the redundant configurations listed previously. The list of potential redundant architectures was developed without evaluation. Now, the ideas should be compared and evaluated. The order of analysis of the approaches should be chosen to minimize work; related approaches should be considered together. After further consideration, it becomes clear that almost all ideas for adding redundant links can be grouped under a single approach (Approach 2):

spare(s) per subcube of some size with connections to some subset of processors in the subcube

This approach is quite general. Many of the other ideas are special cases of this approach as described below.

The relationships between the approaches should be considered further before beginning the evaluations. If the size of the subcube is the size of the whole cube, then this approach is equivalent to adding general spares to the cube. At the other extreme, if the size of the subcube is $2^0 = 1$, then this approach is equivalent to having multiple processors at each node. If connections are added to all processors in the subcube, then this approach is

equivalent to having global spares for the subcube. If connections are added to only one processor in the subcube, then this approach is equivalent to having a spare for a single processor in the subcube or requiring that processor to pass all information to the spare. So, this general approach covers many situations. Its generality reduces its value in developing detailed algorithms. However, it is useful in approaching the problem, because it increases the understanding of the relationships between ideas.

With this relationship between the ideas in mind, as many approaches as possible should be pruned from the list using the design constraints and goals. Consider the approaches listed above.

First, consider Approach 3:

use subcubes as spares instead of single processors

Assume subcube size is not $2^0 = 1$ (the trivial case). The approach is costly in terms of processor redundancy, since a single faulty processor requires that an entire subcube be replaced. This approach is simple and could be implemented using natural redundancy with larger dimension cubes. This approach appears to be a good choice for situations in which processors are relatively inexpensive, which is not the case in this example. Thus, Approach 3 should be removed from the list.

Next, consider Approach 4:

some larger general-type architecture in which cube is embedded (e.g. ternary cubes)

Again, this approach appears to be expensive. For the ternary cube, many extra processors are added to a binary cube of the same dimension, d. Specifically, there are $3^d - 2^d$ extra nodes. Some of the processors in the ternary cube could be omitted so that the cube is incomplete. This approach has little value for cases in which processors are expensive, so it will not be considered further. Other larger general-type architectures could also be used, such as the Star Graph [Nig90].

Consider the first idea in the list:

spare(s) per subcube of some size $2^i$ for i = 0, 1, 2, . . . with connections to each cell in the subcube

This approach looks quite promising. It appears to be simple and not too expensive to implement. It should be retained in the list of candidates and considered further.

Next, consider Approach 2:

spare(s) per subcube of some size with connections to some subset of processors in the subcube

This approach also seems promising, but how to choose the subset of processors in the subcube to which to connect the spare(s) is unclear. This approach should be investigated more fully and should remain on the list.

Finally, consider Approach 5:

add extra column(s) with far east/far west connections (extra row(s) is similar)

This approach also seems promising. It exploits the Euclidean space properties of the cube, rather than being restricted to the cube structure. It is like adding general spares to the cube with connections to some set of processors in the cube. Leave Approach 5 on the list.

Considering the approaches on the list will often lead to new ideas. Another option not listed above is to just add redundant connections (without adding spare processors) to enhance the natural redundancy properties of the cube. Adding connections to processors other than spares is often appropriate. This approach should be added to the list.

In summary, two ideas have been pruned from the list and the following idea (Approach 6) was added:

just add redundant connections (no spares added)

The first analysis of the approaches on the list removed the least promising alternatives. However, there are several remaining approaches which require further consideration. The pruned list is

1. spare(s) per subcube of some size $2^i$ for i = 0, 1, 2, . . . with connections to each cell in the subcube
2. spare(s) per subcube of some size with connections to some subset of processors in the subcube
5. extra column(s) with far east/far west connections (extra row(s) is similar)
6. just add redundant connections (no spares added)

Consider Approach 6 further. Approach 6 should be dismissed because, like natural redundancy, it doesn't maintain the cube structure for the basic graph after reconfiguration since no additional processors are added. As in natural redundancy methods, redundancy in time can be used to maintain the cube structure. However, time redundancy decreases the speed of operation significantly. So, Approach 6 should be pruned from the list.

So, the list of candidates still has three approaches to consider (1, 2, and 5). But, as discussed previously, Approaches 1 and 5 can be considered special cases of Approach 2. Each of these options corresponds to a different choice for the parameter G in the IFSMM.

At this point, I will consider Approach 2:

spare(s) per subcube of some size with connections to some subset of processors in the subcube

with Approaches 1 and 5 considered as special cases. First, consider the basic graph--$2^4$ cube (See Figure 76). Since one design goal is no more than 25% redundancy, a maximum of

$$0.25(16) = 4 \text{ spares}$$

can be added. The redundancy level is somewhat arbitrary, but some guideline is necessary. The designer must decide to which processors these spares should be connected. In considering this problem, since all processors are assumed to be equally likely to fail, a regular structure in the redundant graph G should be maintained. This regular structure requirement should be added

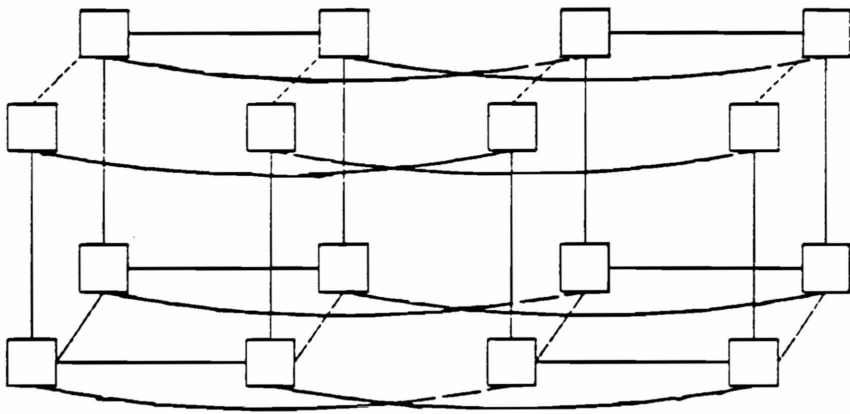


**Figure 76.** Basic Graph -- $2^4$ Cube

as a design goal.

Approach 2 can be subdivided as follows:

A. 4 subcubes of size $2^2$--can add 1 spare per subcube

B. 2 subcubes of size $2^3$--can add up to 2 spares per subcube

C. 1 cube of size $2^4$--can add up to 4 general spares

A definition is needed for clarity. For a spare to be assigned to a subcube means that the spare can only take on the state of the nodes in that subcube. Redundant connections can be added within subcubes and can be added between subcubes, as well. When spares are only used within subcubes, their utility is limited. By limiting spares to subcubes, reliability is lower than it could otherwise be, although complexity may be reduced. Because high reliability is desired, the maximum number of spares will be considered in each case. The Ideas A, B, and C given above should be considered further.

In Idea A, the basic architecture is divided into four subcubes of size $2^2$. This assignment means that one spare can be added per subcube. If connections are only added to some subset of nodes in the subcube, then some faults could not be handled without using other nodes to forward information. In the IFSMM, any communication between nodes, even forwarded information, appears as an edge in G. So, if all single faults in the subcube are to be covered, then edges must exist between each spare and all of the nodes in its associated subcube.

One particularly useful aspect of the IFSMM is that it does not distinguish between actual connections and forwarded information in G. This property simplifies analysis of the redundant graph. The forwarding of information is described by $\delta$ and $\lambda$. The added connections are shown in Figure 77. The edges in the basic graph are omitted for clarity. With respect to reliability, it does not matter how subcubes are chosen, as long as each node in the basic graph is assigned to some subcube.

Additional connections between subcubes are also needed. These are shown for one spare in Figure 78. All other spares have similar added connections. These connections are needed since for a spare to be able to take over for each node in the subcube, it must have the connections to the other subcubes that each of the nodes in the subcube has. Rather than having direct connections, a crossbar switch could be used. A 12 X 4 crossbar would be needed for each spare. One potential problem is that some communication paths might be too long. However, the maximum added path should not be significantly longer than other regular communication paths.

This choice of redundant graph is capable of handling all single faults since it was designed so a spare can take over for any node in its subcube. It cannot handle any double fault situations within a subcube since there is only one spare for each subcube. The amount of information passed between nodes (described by the set of interconnection values, I), is small. The algorithm is quite simple as can be seen in the description of $\lambda$ and $\delta$ below. The output

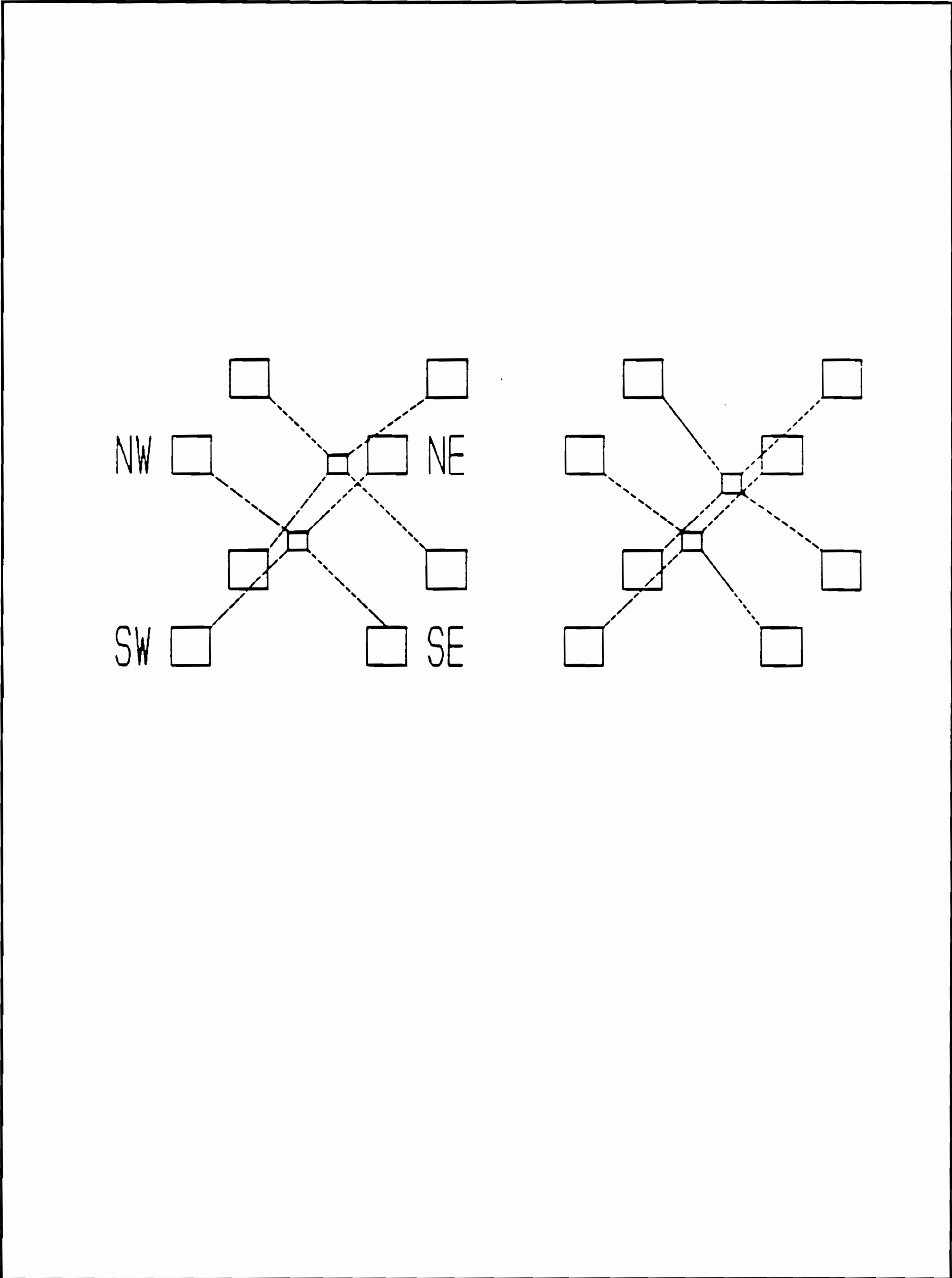



**Figure 77.** Added Connections

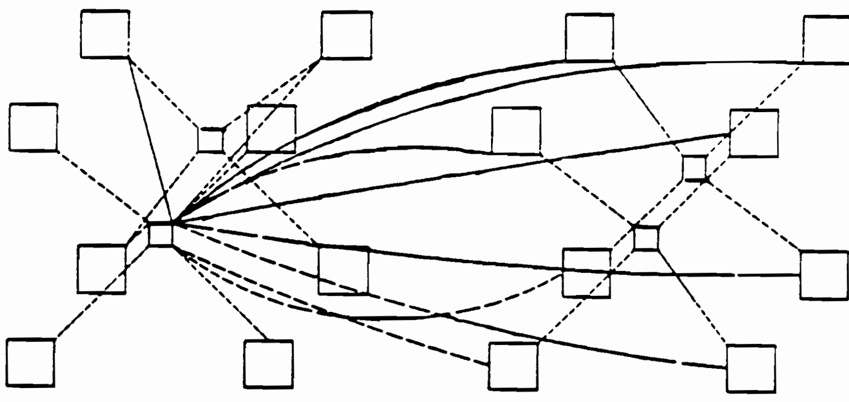


**Figure 78.** Added Connections Between Subcubes

function and the next state function assume that each cell can determine whether each of its neighbors is faulty. Dimension 1 will be considered the North-South dimension, and Dimension 2 will be considered the East-West dimension.

δ: (next state function)

```
/*** determine state ***/
if (own_state = SPARE) then
        if (NE_neighbor_in_same_subcube = FAULTY) then
                rollback to NE neighbor's previous state
        if (NW_neighbor_in_same_subcube = FAULTY) then
                rollback to NW neighbor's previous state
        if (SE_neighbor_in_same_subcube = FAULTY) then
                rollback to SE neighbor's previous state
        if (SW_neighbor_in_same_subcube = FAULTY) then
                rollback to SW neighbor's previous state
else {own state is not spare}
        keep own state;

/*** activate connections ***/
if (own_state = SPARE) then
        if (NE_neighbor_in_same_subcube = FAULTY) then
                activate connections to NE neighbor's neighbors
        if (NW_neighbor_in_same_subcube = FAULTY) then
                activate connections to NW neighbor's neighbors
        if (SE_neighbor_in_same_subcube = FAULTY) then
                activate connections to SE neighbor's neighbors
        if (SW_neighbor_in_same_subcube = FAULTY) then
                activate connections to SW neighbor's neighbors
else {own state is not spare}
        if (neighbor n is FAULTY) then
                activate connection to neighbor n's associated spare
```

λ: (output function)

No reconfiguration information is passed. Only testing information is passed between nodes. Each spare must maintain rollback information for the nodes in its associated subcube.

Idea A will not be considered any further since it cannot handle all double faults.

Now, consider Idea B in detail. In Idea B, the basic architecture is divided into two subcubes of size $2^3$. Thus, up to two spares can be added per subcube. For the high reliability design goal, I will only consider adding two spares per subcube. Communication can be chosen to occur between each spare and each node in the subcube or to occur between each spare and some subset of nodes in the subcube. Since a regular structure is desired and since dividing subcubes into two parts (in a regular manner) is the same as considering smaller subcubes, only the full communication case will be considered.

The extra connections required are shown in Figure 79 for one spare. Each spare must be able to communicate with every node in the entire cube. A crossbar switch seems to be the most appropriate choice for making the connections. A 16 X 4 crossbar is required for each spare. Four additional connections are also added to all regular nodes (one for each spare).

Idea B covers all double faults. In addition, all triple faults, except the case of three in one subcube, are also covered. The set of interconnection values, I, is small. The reconfiguration algorithm is simple. $\delta$ and $\lambda$ are similar to the definitions for Idea A. As will be seen in the discussion of Idea C, this approach has no significant advantage over general spares with full connections. Since Idea B requires each spare to be able to be connected to

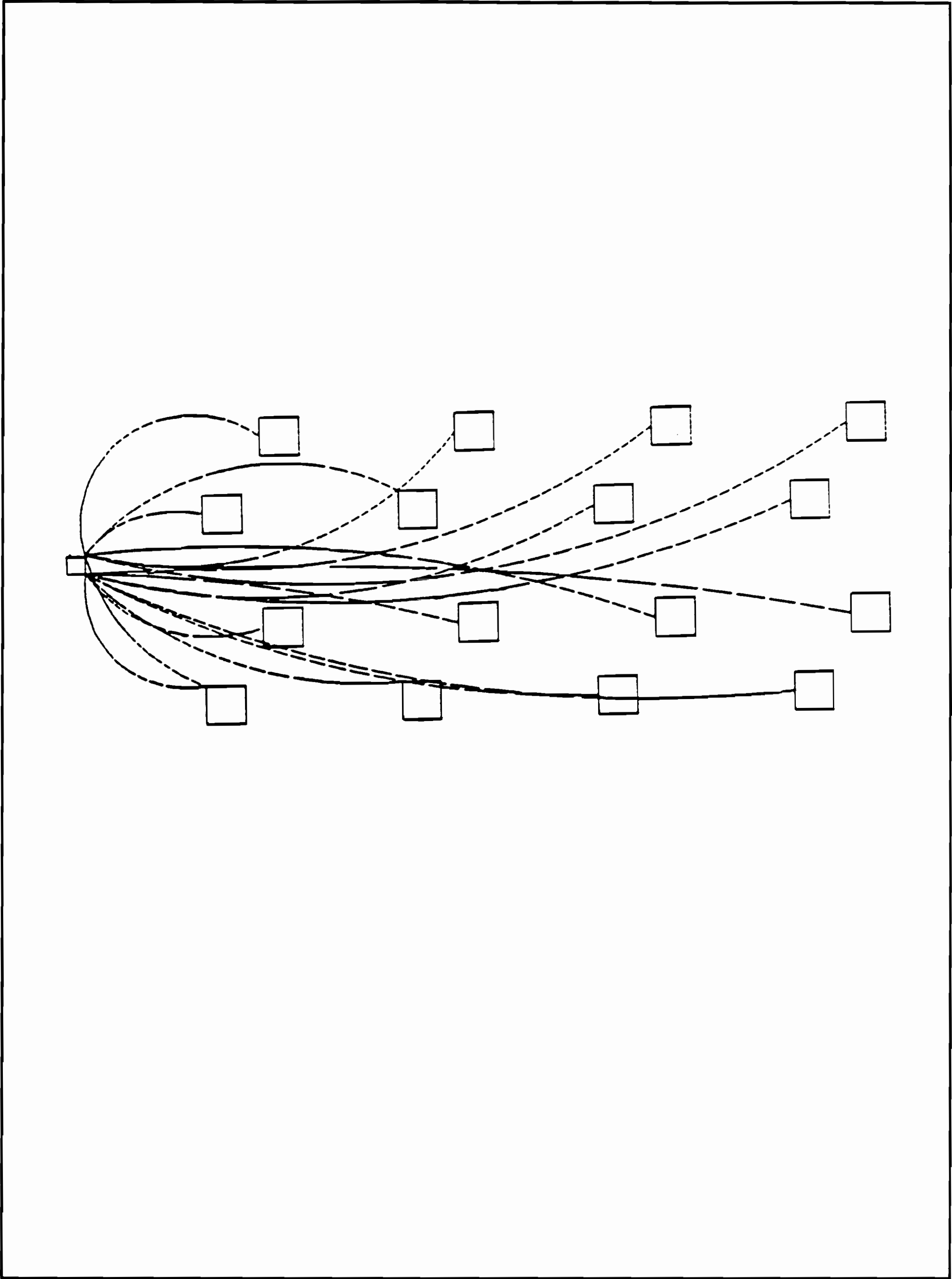


**Figure 79.** Added Connections for Idea B

every node in the cube, I will not consider this approach further.

Finally, consider Idea C in detail. In Idea C, the basic architecture is not divided; so, there is a single cube of size $2^4$. Thus, up to four general spares can be added to the cube. Since a highly reliable system is desired, only the case of four added spares will be considered.

Each spare could be connected to each node in the $2^4$ cube. This approach would require a 16 X 4 crossbar for each spare just as Idea B required. This approach handles all cases of up to 4 simultaneous faults, since each spare can replace any node. (Idea B provides less coverage at the same cost). Connecting each spare to each node in the cube is more complex than desired.

To consider connecting spares to some subset of nodes, a logical way to choose a subset of nodes to which the spare will be connected is needed. The structure of the basic graph should be used to develop the approach for determining connections. Spares could be added to the graph using the subcube idea, but dividing up the subcubes differently (instead of regularly). This approach would still require every spare to be connected to every node and would not really add any other advantages.

Another idea is to add spares and treat the hypercube like a 3-space with extra connections. One choice of additional connections is shown in Figure 80. The maximum node degree of this graph is ten. However, the design choice could be made to activate only 4 of the 10 connections at any

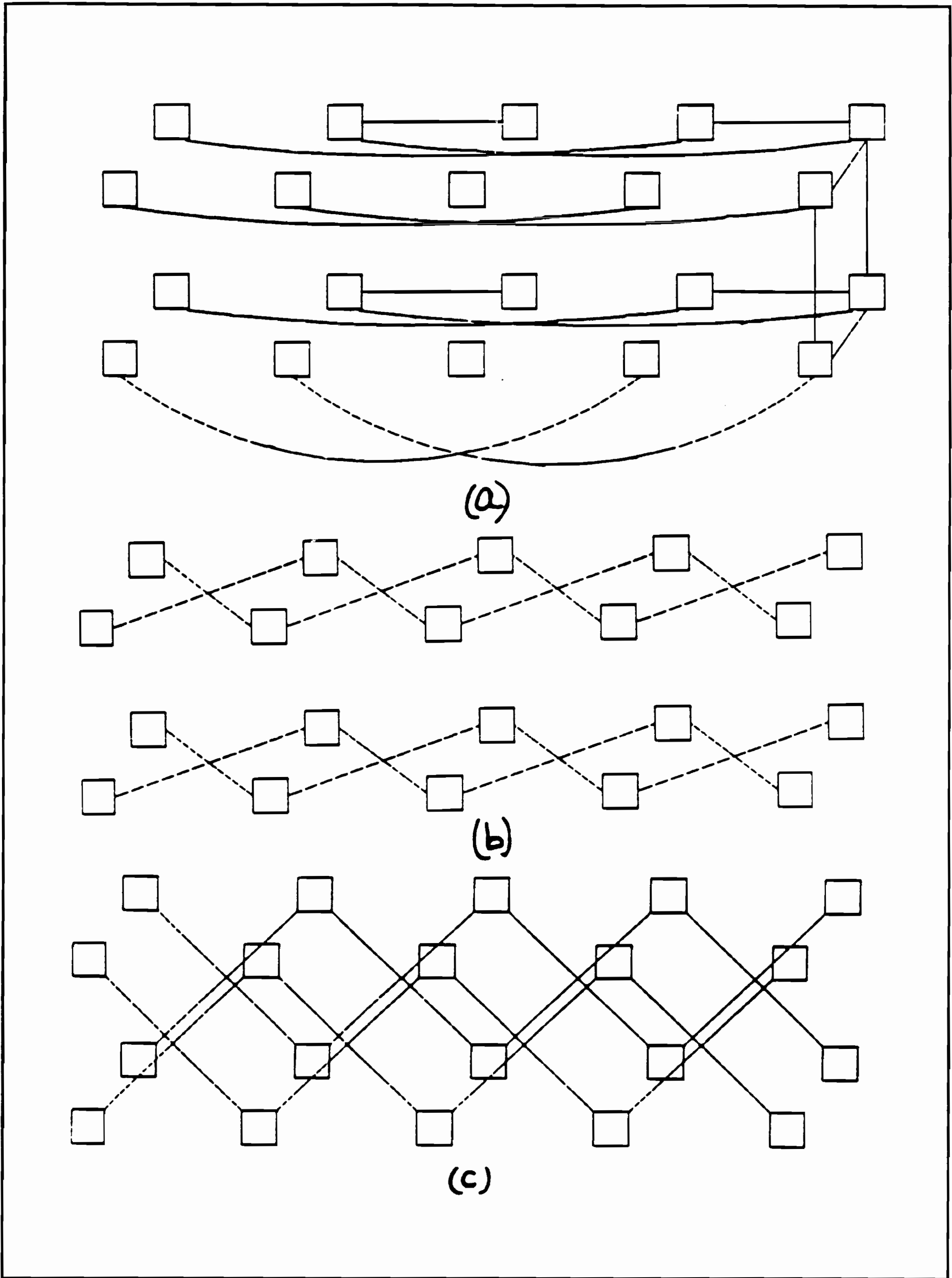



**Figure 80.** Added Connections for 3-Space

given time. The reconfiguration algorithm following this approach is quite simple. The simplicity is shown in the descriptions of $\delta$ and $\lambda$ below. The output function and the next state function assume that each cell can determine if each of its neighbors is faulty. Dimension 1 will be considered to be North-South, while Dimension 2 will be considered to be East-West.

```
δ (next state function):
        if (fault to west) then
                take on west neighbor's state
        activate necessary connections (if desired)
        update fault register

λ (output function):
        pass neighbors' fault information needed to determine new fault
        register contents
```

This final approach seems very promising and will be the design approach chosen. The details of the reconfiguration algorithm are given in the next section.

### 7.2.4 Design Reconfiguration Algorithm

To develop a detailed design for the reconfiguration algorithm, an IFSM model will be used. The approach is based on treating the hypercube like a 3-space with extra connections and added spares. Dimension 1 will be referred to as North-South; Dimension 2 will be called East-West. Dimension 3 will be called Back-Front.

<u>Level 1:</u>

1-1 The facility graph for the redundant system is a $2^4$ cube with the redundant connections shown in Figure 80.

1-2 The set of active nodes, A, is the set of vertices in the facility graph defined in Step 1-1. A is the set of vertices in a $2^4$ cube plus the added spares.

1-3 Some choice of boundary cells is required for each cell to have the same degree. A single boundary cell can be used since all boundary conditions are identical.

1-4 Fault register contents are passed between cells. A fault register consists of 9 bits:

| bit | fault location |
|---|---|
| 1 | west neighbor faulty |
| 2 | east neighbor faulty |
| 3 | north/south neighbor faulty |
| 4 | back/front neighbor faulty |
| 5 | fault to far west |
| 6 | fault to north/south east |
| 7 | fault to north/south west |
| 8 | fault to back/front east |
| 9 | fault to back/front west |
| 10 | fault to far east |

1-5 To determine its output information, each cell needs to know its current fault register, its active connections (for this model all connections will be considered to be active), its own computation state, its west neighbor's computation state, test results for all its active neighbors, and whether it should take its west neighbor's state.

Level 2:

2-1 At each time step, each cell sends its fault register to all of its active neighbors.

2-2 To determine its output, a cell must know its active neighbors' fault registers, as well as its current fault register.

2-3 Pseudocode for the output function is given in the previous section.

2-4 Pseudocode for the next state function is given in the previous section.

Hand performed reconfiguration analyses were performed. All single faults are covered, and 80% of double faults are covered.

Level 3:

The Level 3 IFSMM for the Hypercube Reconfiguration Algorithm is described in detail in Appendix L. Simulation results indicate that all single faults are covered, and 80% of double faults are covered. The reconfiguration algorithm fails in cases in which there is more than one fault along an edge in the x-dimension. This is because essentially one spare is assigned to each x-dimension edge of the redundant architecture.

The hypercube with error E(1) is shown in Figure 81. The reconfiguration algorithm responds to this fault condition by each cell in the x-dimension which is east of the faulty cell taking the state of its west neighbor. The final configuration is shown in Figure 82. The simulation for this example is described in detail in Appendix L.

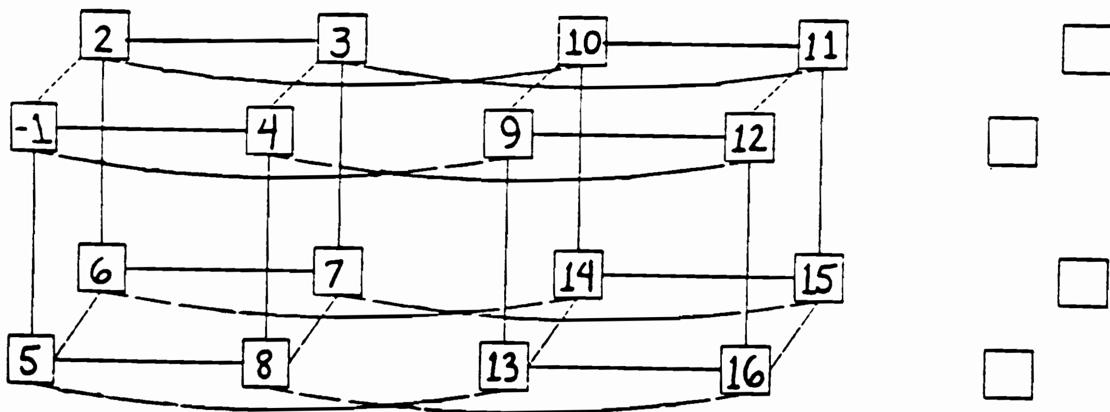



**Figure 81.** Hypercube with E(1)

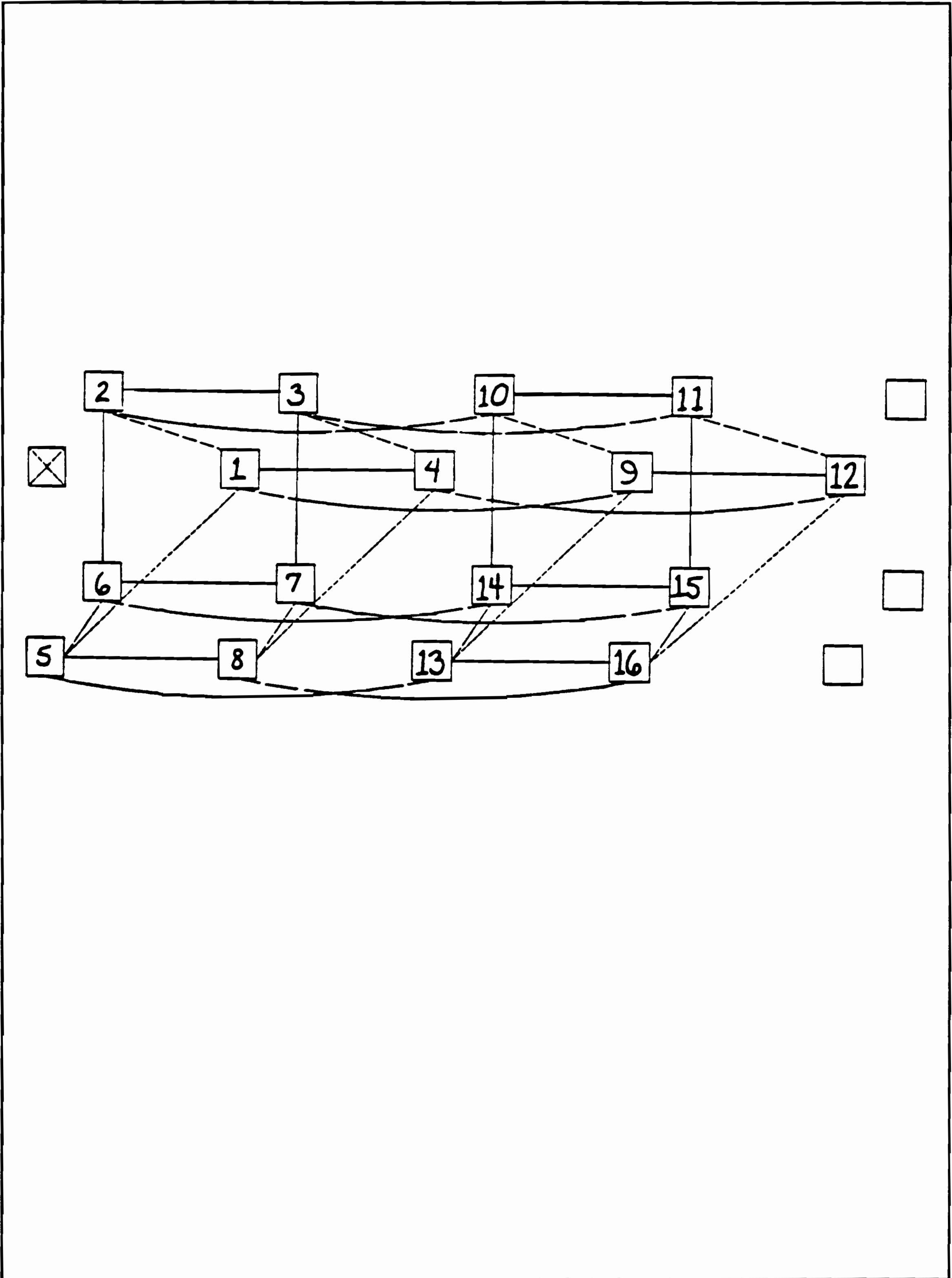



**Figure 82.** Final State after Reconfiguration around E(1)

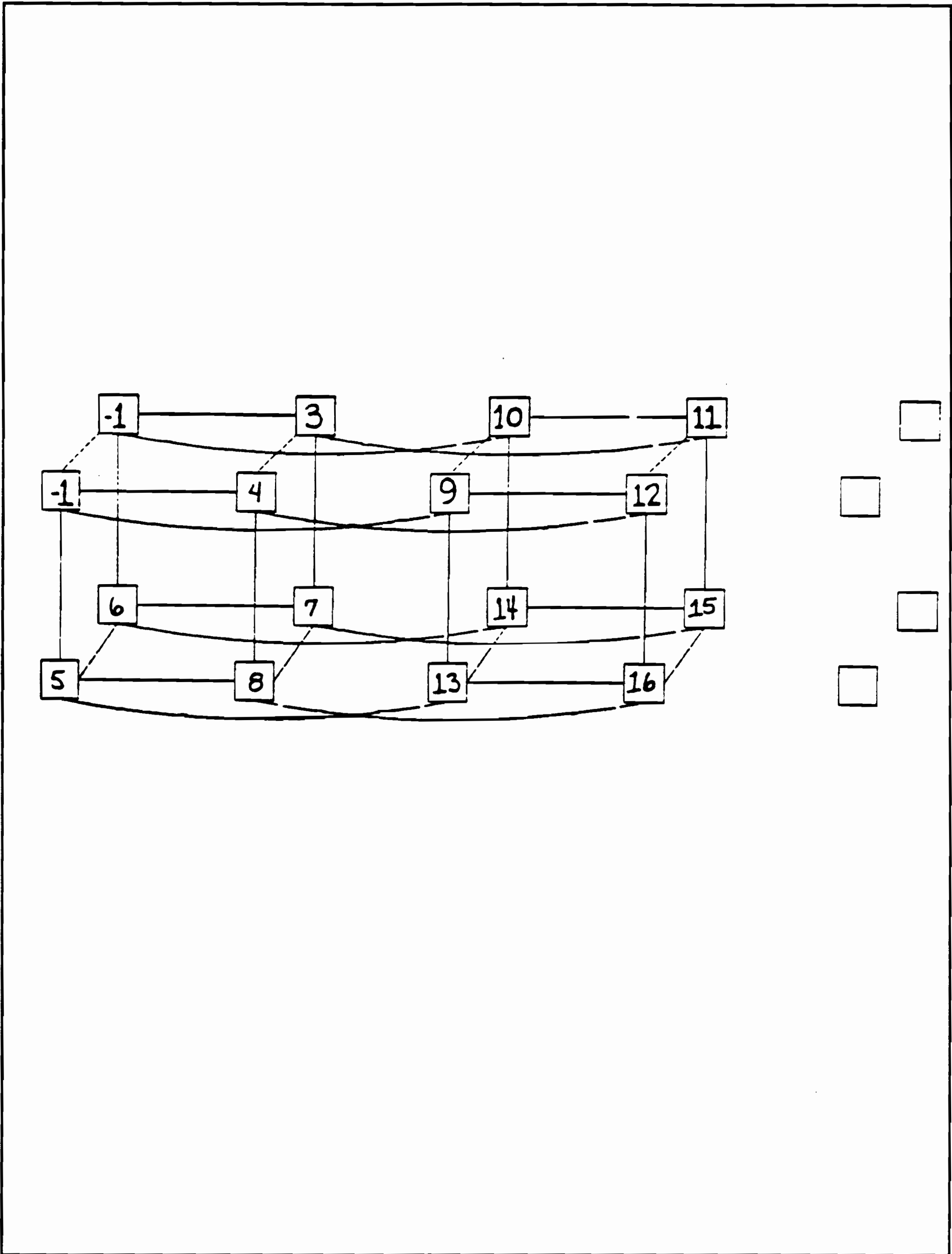



**Figure 83.** System with Fault Combination E(1, 2)

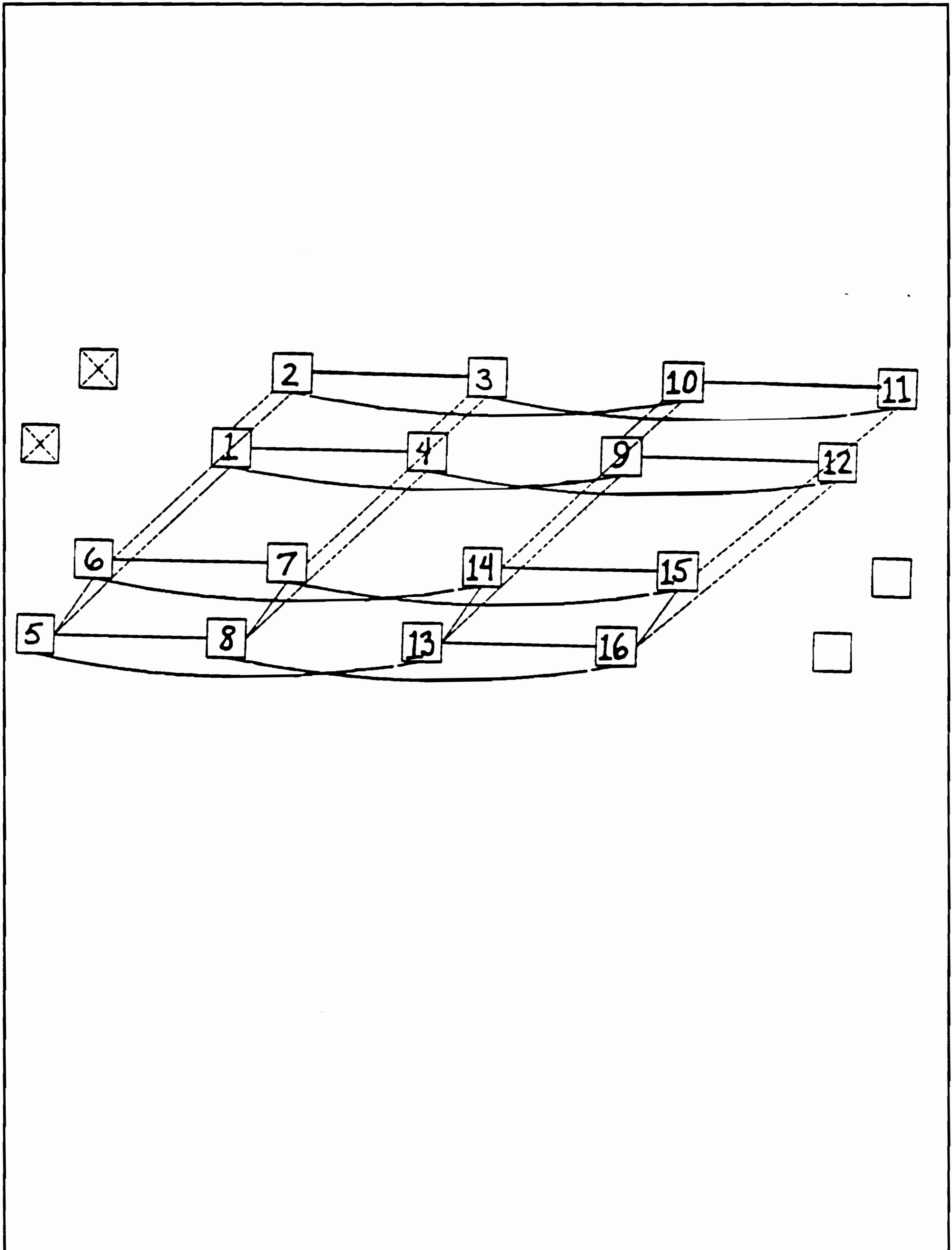



**Figure 84.** Final State after E(1, 2) Reconfiguration

The faulty configuration E(1, 2) will be used as an example of a double fault. This fault combination is shown in Figure 83. In response to this fault combination, the reconfiguration algorithm shifts the states in each faulty state's x-dimension to the east. Thus, each cell directly east of a faulty cell takes on the state of its west neighbor. The final configuration is given in Figure 84.

A case in which the reconfiguration algorithm fails is shown in Figure 85. For this fault combination, reconfiguration fails since there is more than one fault in the x-dimension along one edge of the redundant architecture.

## 7.3 Summary

The Interconnected Finite State Machine Model is a useful tool for designing a reconfigurable system for a particular application. The IFSMM design procedure provides a systematic, consistent process for completely designing a reconfigurable system. The IFSMM design process is a procedure in which a list of alternative approaches is developed. Then, the IFSMM is used to prune the list to a single design. The IFSMM design approach takes advantage of the hierarchical properties of the IFSMM. High level models are used to quickly eliminate the least suitable approaches. Then, as the list is narrowed, more detail is added. Level 3 models should only be developed for one of two approaches which are the most suitable. The Level 3 model is used

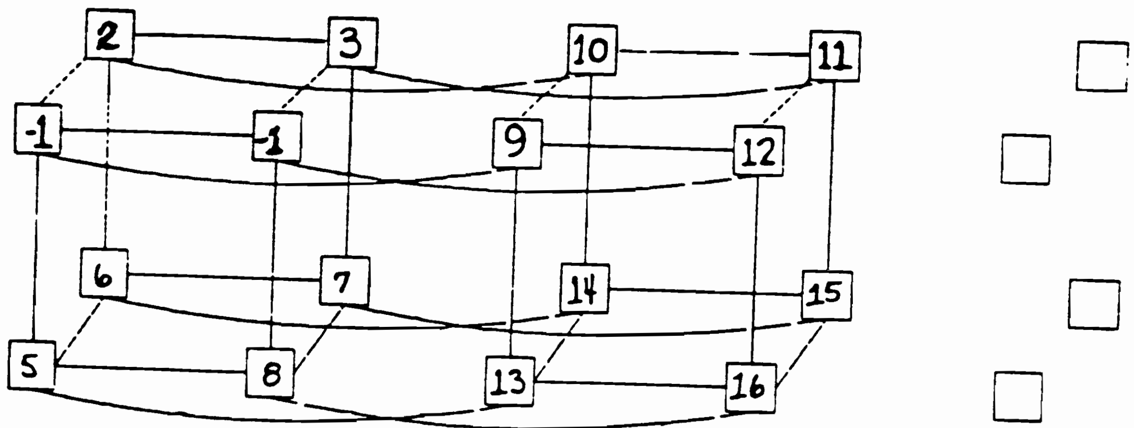



**Figure 85.** System with Fault Combination E(1, 4) -- Reconfiguration Fails!

to complete a design by specifying all details. This design approach considers fault tolerance at the earliest stages of the design process, by allowing various basic architectures to be considered.

A reconfigurable system was developed for a particular application using the IFSMM design method. To make the problem tractable, this example assumes that the hypercube architecture will be the basic architecture which is used. A novel design for a reconfigurable hypercube was developed. All single faults are covered, and 80% of double faults are covered. The design of the reconfigurable hypercube illustrates the use of the IFSMM as a tool for designing reconfigurable systems for a particular application.

# 8.0 Conclusions

Three models were proposed for the evaluation and design of distributed reconfigurable systems: the Local Supervisor Model, the Tessellation Automata Model, and the Interconnected Finite State Machine Model. Each model is based on the concept of cells working together in parallel to achieve a valid system configuration through a series of separate actions. In these models, cells communicate only with their neighbors.

In the Local Supervisor Model, the designated local supervisor cells guide the process of reconfiguration. The Local Supervisor Model was used in the design of the Array Reconfiguration Algorithm, a new algorithm for array reconfiguration.

In the Tessellation Automata Model, each cell in the system determines its next state based only on its present state and its neighbors' present states. This state information is then communicated to the neighbors of the cell. The Tessellation Automata model is hierarchical to allow investigation of algorithms to be performed at as high a level as possible. The Tessellation Automata Model was used to describe several existing algorithms including the

Array Reconfiguration Algorithm which was designed using the Local Supervisor Model.

In the Interconnected Finite State Machine Model, each cell determines its next state from its current state and inputs from its neighbors. In addition, each cell determines its next outputs from its current state and its inputs. Like the Tessellation Automata Model, the Interconnected Finite State Machine Model is hierarchical to support effective investigation of algorithms. The Interconnected Finite State Machine Model was used to describe several existing algorithms including those algorithms described using the Tessellation Automata Model. Because of the versatility of the Interconnected Finite State Machine Model, it was the focus of the rest of this research.

The Interconnected Finite State Machine Model is useful for the improvement of algorithms. It was used to improve the algorithm associated with the Local Supervisor Model so that multiple faults can be handled better and so that spares can be used as local supervisors. In addition, it was used to modify the Direct Reconfiguration Algorithm so that hard core components are not needed.

The Interconnected Finite State Machine Model is also useful for comparing reconfiguration algorithms. Comparisons can be performed at various levels depending on the amount of detail required. The Array Reconfiguration Algorithm was compared to the White-Gray Local Algorithm

using the Interconnected Finite State Machine Model and important differences and similarities were identified.

Finally, the Interconnected Finite State Machine Model was used to develop a novel design for a reconfigurable hypercube-based architecture. This fault tolerant system covers all single faults and 80% of double faults. The design process was described in detail to illustrate how the Interconnected Finite State Machine Model is used.

Several areas for future work were also identified. Specifically, the process of running simulations using the most detailed of the Interconnected Finite State Machine Models was tedious. The collection of the result data was painstaking. A graphical interface to the standard simulation model would greatly simplify the process of collecting data. In addition, it would make the design and development of algorithms much more pleasant. Also, the Hypercube Reconfiguration Algorithm should be studied in greater detail and developed into a complete reconfigurable hypercube.

In summary, the Local Supervisor Model, the Tessellation Automata Model, and the Interconnected Finite State Machine Model appear to be useful for the design and evaluation of distributed, reconfigurable systems for fault tolerant, highly reliable applications. The Interconnected Finite State Machine Model is the most versatile of the three models and has been shown to be useful for design, as well as the analysis of systems.