

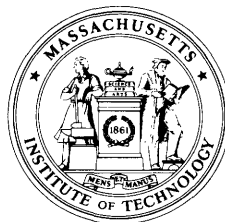
The RESEARCH LABORATORY
of
ELECTRONICS
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS 02139

Coding Approaches to Fault Tolerance in Dynamic Systems

Christoforos N. Hadjicostis

RLE Technical Report No. 628

September 1999



Coding Approaches to Fault Tolerance in Dynamic Systems

by

Christoforos N. Hadjicostis

S.B., Massachusetts Institute of Technology (1993)
M.Eng., Massachusetts Institute of Technology (1995)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

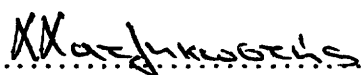
at the


MASSACHUSETTS INSTITUTE OF TECHNOLOGY

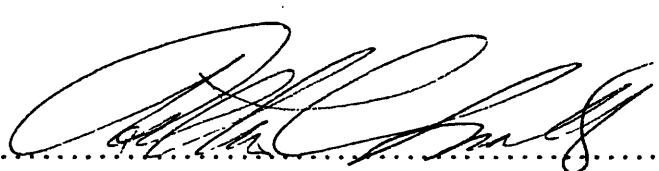
August 1999

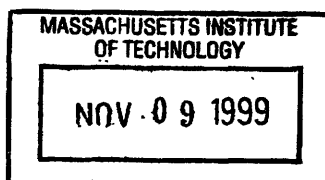
[September 1999]

© Massachusetts Institute of Technology 1999. All rights reserved.

Author.....
Department of Electrical Engineering and Computer Science
August 12, 1999

Certified by.....
George C. Vergheze
Professor of Electrical Engineering
Thesis Supervisor

Accepted by.....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students



ARCHIVES

Coding Approaches to Fault Tolerance in Dynamic Systems

by

Christoforos N. Hadjicostis

Submitted to the Department of Electrical Engineering and Computer Science
on August 12, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

A fault-tolerant system tolerates internal failures while preserving desirable overall behavior. Fault tolerance is necessary in life-critical or inaccessible applications, and also enables the design of reliable systems out of unreliable, less expensive components. This thesis discusses fault tolerance in dynamic systems, such as finite-state controllers or computer simulations, whose internal state influences their future behavior. Modular redundancy (system replication) and other traditional techniques for fault tolerance are expensive, and rely heavily — particularly in the case of dynamic systems operating over extended time horizons — on the assumption that the error-correcting mechanism (e.g., voting) is fault-free.

The thesis develops a systematic methodology for adding structured redundancy to a dynamic system and introducing associated fault tolerance. Our approach exposes a wide range of possibilities between no redundancy and full replication. Assuming that the error-correcting mechanism is fault-free, we parameterize the different possibilities in various settings, including algebraic machines, linear dynamic systems and Petri nets. By adopting specific error models and, in some cases, by making explicit connections with hardware implementations, we demonstrate how the redundant systems can be designed to allow detection/correction of a fixed number of failures. We do not explicitly address optimization criteria that could be used in choosing among different redundant implementations, but our examples illustrate how such criteria can be investigated in future work.

The last part of the thesis relaxes the traditional assumption that error-correction be fault-free. We use unreliable system replicas and unreliable voters to construct redundant dynamic systems that evolve in time with low probability of failure. Our approach generalizes modular redundancy by using distributed voting schemes. Combining these techniques with low-complexity error-correcting coding, we are able to efficiently protect identical unreliable linear finite-state machines that operate in parallel on distinct input sequences. The approach requires only a constant amount of redundant hardware per machine to achieve a probability of failure that remains below any pre-specified bound over any given finite time interval.

Thesis Supervisor: George C. Verghese
Title: Professor of Electrical Engineering

Acknowledgments

First and foremost, I would like to express my most sincere thanks to my thesis supervisor, Professor George Verghese, for his inspiring guidance and unlimited support throughout my graduate studies and research. Without his enthusiasm, encouragement and patience whenever I reached a point of difficulty, this thesis would not have reached its current form.

I am also extremely grateful to Professors Alan Oppenheim and Gregory Wornell for their direction, warm support, and hospitality over these years. Not only did their financial assistance make this thesis possible, but they also gave me the opportunity to work in an excellent academic environment that helped me mature as a researcher and as a person.

I am indebted to many members of the faculty at MIT for their involvement and contribution to my thesis work. My committee members, Professors Sanjoy Mitter and Alex Megretski, were a source for help and inspiration. Their challenging and thought-provoking questions have shaped many aspects of this thesis. Furthermore, the discussions that I had with Professors Bob Gallager, David Forney, Daniel Spielman, and Srinivas Devadas were also extremely encouraging and helpful in defining my research direction; I am very thankful to all of them.

I would also like to thank my many friends and fellow graduate students who made life at MIT both enjoyable and productive. Special thanks go to my “buddy” Carl Livadas who was there every time I needed to ask an opinion, or simply complain. Babis Papadopoulos and John Apostolopoulos were a great source of advice during the formative stages of this work (particularly during late hours!). Thanos Siapas gave me lots of feedback (and laughs) at the later stages of my thesis. Hisham Kassab, Sekhar Tatikonda, Costas Boussios, Chalee Asavathiratham, and Tony Ezzat were good friends and also extremely helpful with comments and suggestions.

I also want to acknowledge all members of the Digital Signal Processing Group for being great colleagues and friends. My time at the DSPG will remain an unforgettable experience. My special thanks to Stark Draper, Richard Barron, Nicholas Laneman, Matt Secor, Charles Sestok, and Jeff Ludwig. Giovanni Aliberti not only offered computer expertise, but was also a great friend; Darla Chupp, Vivian Mizuno, Maggie Beucler, Janice Zaganjori and Sally Bemus made life a lot simpler by meticulously taking care of administrative matters.

Finally, I am grateful to the Defense Advanced Research Projects Agency for support under the RASSP project, EPRI and the Department of Defense for support under the Complex Network Initiation, and the National Semiconductor Corporation and the Grass Instrument Company for their generous financial support. Peggy Carney’s help with finding and administrating fellowship support is gratefully acknowledged; her assistance was most valuable for the completion of this work.

To my parents

Contents

1	Introduction and Background	15
1.1	Definitions and Motivation	15
1.2	Background: Fault Tolerance in Computational Systems	17
1.3	Fault Tolerance in Dynamic Systems	22
1.3.1	Redundant Implementation	25
1.3.2	Error-Correction	28
1.4	Scope and Major Contributions of the Thesis	29
1.5	Outline of the Thesis	32
2	Redundant Implementations of Algebraic Machines	33
2.1	Introduction	33
2.2	Background: Fault-Tolerant Computation in Groups and Semigroups	34
2.2.1	Fault Tolerance in Abelian Group Computations	34
2.2.2	Fault Tolerance in Semigroup Computations	37
2.3	Redundant Implementations of Algebraic Machines	41
2.3.1	Redundant Implementations of Group Machines	42
2.3.2	Redundant Implementations of Semigroup Machines	54
2.4	Redundant Implementations of Finite Semiautomata	59
2.4.1	Characterization of Non-Separate Redundant Implementations . . .	62
2.4.2	Characterization of Separate Redundant Implementations	64
2.5	Summary	67

3	Redundant Implementations of Linear Time-Invariant	
	Dynamic Systems	69
3.1	Introduction	69
3.2	Linear Time-Invariant Dynamic Systems	70
3.3	Characterization of Redundant Implementations	71
3.4	Hardware Implementation and Error Model	74
3.5	Examples of Fault-Tolerant Schemes	77
3.6	Summary	89
4	Redundant Implementations of Linear Finite-State Machines	91
4.1	Introduction	91
4.2	Linear Finite-State Machines	92
4.3	Characterization of Redundant Implementations	95
4.4	Examples of Fault-Tolerant Schemes	97
4.5	Summary	106
5	Failure Monitoring in Discrete Event Systems Using Redundant	
	Petri Net Implementations	107
5.1	Introduction	107
5.2	Petri Net Models of Discrete Event Systems	108
5.3	Error Model	112
5.4	Monitoring Schemes Using Separate	
	Redundant Implementations	115
5.4.1	Separate Redundant Petri Net Implementations	115
5.4.2	Failure Detection and Identification	119
5.5	Monitoring Schemes Using Non-Separate	
	Redundant Implementations	127
5.5.1	Non-Separate Redundant Petri Net Implementations	127
5.5.2	Failure Detection and Identification	133
5.6	Applications in Control	137
5.6.1	Monitoring Active Transitions	137
5.6.2	Detecting Illegal Transitions	139

5.7	Summary	142
6	Unreliable Error-Correction	143
6.1	Introduction	143
6.2	Problem Statement	144
6.3	Distributed Voting Scheme	145
6.4	Reliable Linear Finite-State Machines Using Constant Redundancy	150
6.4.1	Low-Density Parity Check Codes and Stable Memories	150
6.4.2	Reliable Linear Finite-State Machines	154
6.4.3	Further Issues	160
6.5	Summary	164
7	Conclusions and Future Directions	167
A	Conditions for Single-Error Detection and Correction	173
A.1	Semigroup Computations	173
A.2	Finite Semiautomata	175
B	Proof of Theorem 6.2	179
B.1	“Steady-State” Under No Initial Propagation Failure	180
B.2	Conditional Probabilities Given No Initial Propagation Failure	182
B.3	Bounds on the Probabilities of Failures	184
B.3.1	Bounding the Probability of Initial Propagation Failure	184
B.3.2	Bounding the Probability of Overall Failure	187

List of Figures

1-1	Fault tolerance using an arithmetic coding scheme.	20
1-2	Triple modular redundancy with correcting feedback.	24
1-3	Thesis approach for fault tolerance in a dynamic system.	26
2-1	Fault tolerance in a group computation using a homomorphic mapping. . .	36
2-2	Partitioning of semigroup (N, \times) into congruence classes.	40
2-3	Error detection and correction in a redundant implementation of a group machine.	43
2-4	Monitoring scheme for a group machine.	45
2-5	Series-parallel decomposition of a group machine.	49
2-6	Construction of a separate monitor based on group machine decomposition.	50
3-1	Delay-adder-gain circuit and the corresponding signal flow graph.	75
3-2	Digital filter implementation using delays, adders and gains.	80
3-3	Redundant implementation based on a checksum condition.	81
3-4	A second redundant implementation based on a checksum condition.	83
4-1	Example of a linear feedback shift register.	92
4-2	Three different implementations of a convolutional encoder.	100
5-1	Example of a Petri net with three places and three transitions.	109
5-2	Cat-and-mouse maze.	111
5-3	Petri net model of a distributed processing system.	114
5-4	Petri net model of a digital system.	115
5-5	Concurrent monitoring scheme for a Petri net.	116

5-6	Example of a separate redundant Petri net implementation that identifies single-transition failures in the Petri net of Figure 5-1.	121
5-7	Example of a separate redundant Petri net implementation that identifies single-place failures in the Petri net of Figure 5-1.	123
5-8	Example of a separate redundant Petri net implementation that identifies single-transition or single-place failures in the Petri net of Figure 5-1. . . .	125
5-9	Concurrent monitoring using a non-separate Petri net implementation. . . .	127
5-10	Example of a non-separate redundant Petri net implementation that identifies single-transition failures in the Petri net of Figure 5-1.	134
5-11	Example of a non-separate redundant Petri net implementation that identifies single-place failures in the Petri net of Figure 5-1.	136
5-12	Example of a separate redundant Petri net implementation that enhances control of the Petri net of Figure 5-3.	138
6-1	Reliable state evolution using unreliable error-correction.	146
6-2	Modular redundancy using a distributed voting scheme.	147
6-3	Hardware implementation of the modified iterative decoding scheme for LDPC codes.	152
6-4	Replacing k LFSM's with n redundant LFSM's.	155
6-5	Encoded implementation of k LFSM's using n redundant LFSM's.	157
A-1	Conditions for single-error detection in a finite semiautomaton.	175
A-2	Conditions for single-error correction in a finite semiautomaton.	176

List of Tables

- 6.1 Typical values for p , β and C given J , K , p_x and p_v . The bound on the probability of overall failure is shown for $d = 10$, $k = 10^7$ and $L = 10^5$ 159

Chapter 1

Introduction and Background

1.1 Definitions and Motivation

A *fault-tolerant* system tolerates internal failures and prevents them from unacceptably corrupting its overall behavior, output or final result. Fault tolerance is motivated primarily by applications that require high reliability (such as life-critical medical equipment, defense systems and aircraft controllers), or by systems that operate in remote locations where monitoring and repair may be difficult or even impossible (as in the case of space missions and remote sensors), [5, 85]. In addition, fault tolerance is desirable because it relaxes design/manufacturing specifications (leading for example to yield enhancement in integrated circuits, [59, 63, 80]), and also because it enables new technologies and the construction of reliable systems out of unreliable (possibly fast and inexpensive) components. As the complexity of computational and signal processing systems increases, their vulnerability to failures becomes higher, making fault tolerance necessary rather than simply desirable, [90]; the current trends towards higher clock speed and lower power consumption aggravate this problem even more.

Fault tolerance has been addressed in a variety of settings. The most systematic treatment has been for the case of reliable digital transmission through unreliable communication links and has resulted in error-correcting coding techniques that efficiently protect against channel noise, [95, 96, 38, 81, 11, 111]. Fault tolerance has also been used to protect com-

putational circuits against hardware failures. These failures¹ can be either *permanent* or *transient*: permanent failures could be due to manufacturing defects, irreversible physical damage, or stuck-at faults, whereas transient² failures could be due to noise, absorption of alpha particles or other radiation, electromagnetic interference, or environmental factors. Techniques for fault tolerance have also been applied at a higher level to protect special-purpose systems against a fixed number of “functional” failures, which could be hardware, software or other; these ideas were introduced within the context of algorithm-based fault tolerance techniques (see [50, 93]).

In this thesis we explore fault tolerance in *dynamic systems*:

Definition 1.1 *A dynamic (or state-space) system is a system that evolves in time according to some internal state. More specifically, the state of the system at time step t , denoted by $q[t]$, together with the input at time step t , denoted by $x[t]$, completely determine the system’s next state according to a state evolution equation*

$$q[t + 1] = \delta(q[t], x[t]) .$$

The output $y[t]$ of the system at time step t is based on the corresponding state and input, and is captured by the output equation

$$y[t] = \lambda(q[t], x[t]) .$$

Examples of dynamic systems include finite-state machines, digital filters, convolutional encoders, decoders, and algorithms or simulations running on a computer architecture over several time steps. This thesis will focus on failures that cause an unreliable dynamic system to take a transition to an incorrect state³. Depending on the underlying system and its actual implementation, these failures can be permanent or transient, and hardware or software. Due to the nature of dynamic systems, the effects of a state transition failure may last over several time steps; state corruption at a particular time step generally leads

¹For more details on hardware failures see [31, 109] and references therein.

²A transient or temporal failure is a failure whose *cause* (but not necessarily the effect) appears only temporarily.

³A study of this error model in the context of sequential VLSI circuits appears in [24].

to the corruption of the overall behavior and output at future time steps.

To understand the severity of the problem, consider the following situation: assume that an unreliable dynamic system (such as a finite-state machine that is constructed out of failure-prone gates) is subject to *transient* failures with a probability of making an incorrect transition (on any input at any given time step) that is fixed at p_s . If failures at different time steps are independent, then the probability that the system follows the correct state trajectory for L consecutive time steps is $(1 - p_s)^L$ and goes to zero exponentially with L . In general, the probability that we are in the correct state after L steps is also low⁴. This means that the output of the system at time step L will be erroneous with high probability (because it is calculated based on an erroneous state). Therefore, our first priority (and the topic of this thesis work) is to ensure that the system follows the correct state trajectory.

Before we discuss our approach for constructing fault-tolerant dynamic systems, we describe in more detail previous work on fault-tolerant *computational* circuits. The distinction between dynamic systems and computational circuits is that the former evolve in time according to their internal state (memory), whereas the latter have no internal state and no evolution with respect to time.

1.2 Background: Fault Tolerance in Computational Systems

A necessary condition for a computational system to be fault-tolerant is that it exhibit *redundancy*. “Structured redundancy” (that is, redundancy that has been intentionally introduced in some systematic way) allows a computational system to distinguish between valid and invalid results and, if possible, perform the necessary error-correction procedures. Structured redundancy can also be used to guarantee *acceptably degraded* performance despite failures. A well-designed fault-tolerant system makes efficient use of resources by adding redundancy in those parts of the system that are more liable to failures than others, and adding the redundancy in ways that are adapted to the operation of the system.

The traditional way of designing fault-tolerant computational systems that cope with

⁴The probability that we are in the correct state after L steps depends on the structure of the particular finite-state machine, on the error model and on whether multiple failures may lead to a correct state. The argument can be made more precise if we choose a particular structure for our machine (consider for example a linear feedback shift register with failures that cause each bit in its state vector to flip with probability p).

hardware failures is to use N -modular hardware redundancy, [107]. By replicating the original system N times, we compute the desired function multiple times in parallel. The outputs of all replicas are compared and the final result is chosen based on what the majority of them agrees upon. Modular redundancy has been the primary methodology for fault-tolerant system design because it is universally applicable⁵ and because it effectively decouples system design from fault tolerance design. Modular redundancy, however, is inherently expensive and inefficient due to system replication.

Research in communications has extensively explored alternative, more efficient ways of utilizing redundancy for achieving reliable digital transmission through an imperfect (“noisy”) channel. In his seminal work [95, 96], Shannon showed that, contrary to the common perception of the time, one can send multiple bits encoded in a way that achieves arbitrarily low probability of error per bit with a *constant* amount of redundancy (per bit). This result generated a variety of subsequent work in information and coding theory, [38, 81, 11, 111].

In more complex systems that involve not only simple transmission of the data but also some simple processing on the data (e.g., boolean circuits or signal processing systems with no evolution over time) the application of such coding ideas becomes more challenging. In addition, as pointed out in [5, 83], there have traditionally been two different philosophies in terms of dealing with failures in computational systems:

- One school of thought designs systems in a way that allows detection and/or correction of a *fixed* number of failures. For example, numerous systems have been designed with the capability to detect/correct single failures assuming that the error detecting/correcting mechanisms are *fault-free*. (Triple modular redundancy, which protects against a single failure in any one subsystem but not in the voter, is perhaps the most common case.) These approaches are based on the premise that failures are rare (therefore, protecting against a fixed number of failures is good enough⁶) and that the error-correcting mechanism is much simpler than the actual system implementation.

This approach has resulted in a lot of practical fault-tolerant systems, particularly for

⁵A number of commercial and other systems have used modular redundancy techniques, [6, 45]; a comprehensive list can be found in [8].

⁶For example, if failures are independent and happen with probability $p \ll 1$, then the probability of two simultaneous failures is of the order of p^2 , which is very small compared to p .

special-purpose tasks, where the structure of the underlying algorithm and/or hardware configuration can be exploited in order to minimize the hardware overhead, or the complexity of the redundant system and the corresponding correcting mechanism. Such ideas have been explored in sorting networks [25, 102, 64], 2-D systolic arrays for parallel matrix multiplication [50, 56], other matrix operations [1, 23], convolution using the fast Fourier transform [10], and many others. Similar principles prevail in the design of *self-checking* systems, [88, 84]. In these systems we are interested in ensuring that any combination of a fixed number of failures (including failures in the error-detecting mechanism) will be detected.

- The second approach to fault tolerance focuses on building reliable systems out of unreliable components. As we add redundancy into a fault-tolerant system, the probability of failure per component remains constant. Thus, the larger the system, the more failures it has to tolerate, but the more flexibility we have in using the added redundancy/functionality to ensure that, with high probability, the redundant system will have the desirable behavior. Work in this direction started with von Neumann [107], and has been continued by many others [112, 106], mostly in the context of fault-tolerant boolean circuits (see [83] for a comprehensive list).

The idea of adding a minimal amount of redundancy in order to detect/correct a (pre-specified) number of failures (i.e., the first of the two approaches described above) has been quite successful in cases where one can exploit structural features of a computation or an algorithm and introduce “analytical redundancy” in a way that offers more efficient fault coverage than modular redundancy (at the cost of narrower applicability and harder design). Work in this direction includes *arithmetic codes*, *algorithm-based fault tolerance* and *algebraic techniques*. We describe these ideas in more detail below:

Arithmetic Codes: Arithmetic codes are error-correcting codes with properties that remain invariant under the arithmetic operations of interest, [87, 88]. They are typically used as shown in Figure 1-1 (which is drawn for the case of two operands, but more operands are handled in the same way). We first add “analytical redundancy” into the representation of the data by using suitable encodings, denoted by the mappings ϕ_1 and ϕ_2 in the figure. The desired original computation $r = g_1 \circ g_2$ is then replaced

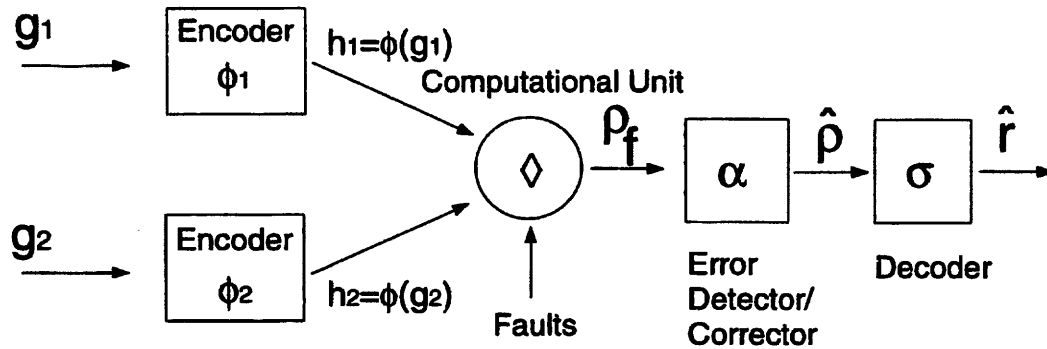


Figure 1-1: Fault tolerance using an arithmetic coding scheme.

by the modified computation \diamond on the encoded data. Under fault-free conditions, this modified operation produces $\rho = \phi_1(g_1) \diamond \phi_2(g_2)$, which results in r when decoded through the mapping σ (i.e., $r = \sigma(\rho)$). However, due to the possible presence of failures, the result of the redundant computation could be faulty, ρ_f instead of ρ . The redundancy in ρ_f is subsequently used to perform error detection and correction, denoted in the figure by the mapping α . Note that the detector/corrector α has no knowledge of the inputs and bases its decision solely on ρ_f . The output $\hat{\rho}$ of the error detector and corrector is decoded through the use of the decoding mapping σ . Under fault-free conditions or with correctable failures, $\hat{\rho}$ equals ρ , and the final result \hat{r} equals r . A common assumption in the model of Figure 1-1 is that the error detector/corrector is fault-free. This assumption is reasonable if the implementation of the decoder/corrector is simpler than the implementation of the computational unit (or if correcting occurs rarely). Another inherent assumption is that no failure takes place in the decoder unit; this assumption is in some sense inevitable: no matter how much redundancy we add, the output of a system will be faulty if the device that is supposed to provide the output fails (i.e., if there is a failure in the very last stage of the computational circuit/system). One way to avoid this problem is to assume that the output is provided to the user in an *encoded form* which can be correctly decoded by a fault-free final stage. In the modular redundancy case, for example, the output could be considered correct if the majority of the systems agree on the correct output (since a *fault-free* majority voter is then guaranteed to provide the correct output).

Algorithm-Based Fault Tolerance: More sophisticated coding techniques, known as Algorithm-Based Fault Tolerance (ABFT), were introduced by Abraham and coworkers [50, 56, 57, 74], starting in 1984. These schemes usually deal with arrays of real/complex data in concurrent multiprocessor systems. The classic example of ABFT is in the protection of $M \times M$ matrix multiplication on a 2-D systolic array, [50]. A variety of computationally intensive algorithms, such as other matrix computations [50, 56], FFT computational networks [57], and digital convolution [10], have since been adapted⁷ to the requirements of ABFT.

As described in [56], there are three critical steps involved in ABFT schemes: (i) encoding the input data for the algorithm (just as for arithmetic coding), (ii) reformulating the original algorithm so that it can operate on the encoded data and produce decodable results, and (iii) distributing the computational tasks among the different subsystems of the failure-prone system so that any failures occurring within these subsystems can be detected and, hopefully, corrected. The above three steps are evident in the ABFT scheme for matrix multiplication that was presented in [50]. The encoding step involves adding an extra “checksum” row/column to the original $M \times M$ matrices. The redundant operation involves multiplication of an $(M + 1) \times M$ matrix by an $M \times (M + 1)$ matrix. When using a 2-D systolic array to perform matrix multiplication, we manage to distribute both the computational tasks and the possible failures in a way that allows efficient failure detection, location and correction.

Algebraic Approaches: The most important challenge in both arithmetic coding and ABFT implementations is the recognition of structure in an algorithm that is amenable to the introduction of redundancy. A step towards providing a systematic approach for recognition and exploitation of such special structure was made for the case of computations that occur in a *group* or in a *semigroup*, [8, 9, 43, 44]. The key observation is that the desired analytical redundancy can be introduced by homomorphic embedding into a larger algebraic structure (group or semigroup). The approach extends

⁷As mentioned earlier, this approach attempts to protect against a pre-specified maximum number of failures assuming fault-free error-correction. Some researchers have actually analyzed the performance of these schemes when the probability of failure in each component remains constant, [12, 103]. As expected, the scheme performs well if the probability of failure per component is very small.

to semirings, rings, fields, modules and vector spaces (i.e., algebraic structures that have the underlying characteristics of a semigroup or a group). A relatively extensive set of computational tasks can therefore be modeled using this framework. We give a brief overview of this approach in the beginning of Chapter 2.

The above mentioned approaches were mostly tailored for computational systems (systems without internal state) and assumed that error-correction is fault-free. As mentioned earlier, this assumption may be tolerable if the complexity of the correcting mechanism is considerably less than the complexity of the state evolution mechanism. Also, a fault-free output stage is in some sense inevitable: if all components may fail then, no matter how much redundancy we add, the output of a system will be faulty if the device that is supposed to provide the output fails (i.e., if there is a failure in the very last stage of the computation/circuit).

A significant aspect of any work on fault tolerance is the development of an appropriate error model. The *error model* describes the *effect* of failures on the output of a computational system, effectively allowing the mathematical study of fault tolerance. The error model does not have to mimic the actual fault mechanism; for example, we can model the error due to a failure in a multiplier as additive, or the error due to a failure in an adder as multiplicative⁸. Efficient error models need to be close to reality, yet general enough to allow algebraic or algorithmic manipulation. If a single hardware failure manifests itself as an unmanageable number of errors in the analytical representation, then the performance of our error detection/correction scheme will be unnecessarily complicated.

1.3 Fault Tolerance in Dynamic Systems

Traditionally, fault tolerance in dynamic systems has used modular redundancy. The technique is based on having replicas of the unreliable dynamic system, each initialized at the same state and supplied with the same inputs. Each system goes through the same sequence of states unless failures in the state transition mechanism cause deviations from this correct

⁸The faulty result r_f of a *real-number* multiplier can always be modeled in an additive error fashion as $r_f = r + e$ where r is the correct result and e is the additive error that has taken place. Similarly for the multiplicative representation of a failure in an adder (if $r \neq 0$).

behavior. If we ensure that failures in the different system replicas are independent (e.g., by requiring that they are hardware- and/or software-independent), then the majority of the replicas at a certain time step will be in the correct state with high probability; an external voting mechanism can then decide what the correct state is using a majority voting rule.

If we revisit the toy example of the unreliable dynamic system that makes a transition to an incorrect next state with probability p_s (independently at different time steps), we see that the use of majority voting at the end of L time steps may be highly unsuccessful: after a system replica operates (without error-correction) for L time steps, the probability that it has followed the correct sequence of states is $(1 - p_s)^L$; in fact, at time step L , a system replica may be in an incorrect state with a prohibitively high probability⁹ (for example, if an incorrect state is more likely to be reached than the correct one, then a voting mechanism will be unable to decide what the correct state/result is, regardless of how many times we replicate the system). One solution could be to correct the state of our systems at the end of *each* time step¹⁰. This is shown in Figure 1-2: at the end of each time step, the voter decides what the correct state is, based on a majority voting rule; this “corrected” state is then fed back to all systems.

Another possibility could be to let the systems evolve for several time steps and then perform error-correction using a mechanism that is more complicated than a simple voter. For example, one could look at the *overall* state evolution (not just the final states) of all system replicas and then make an educated decision on what the correct state *sequence* is. A possible concern about this approach is that, by allowing the system to evolve incorrectly for several time steps, we may compromise system performance in the intervals between error-correction. We do not explore such alternatives in this thesis, mainly because we eventually allow failures in the error-correcting mechanism and this is a rich issue even within the simpler setting.

The approach in Figure 1-2, more generally known as *concurrent error-correction*, has two major drawbacks:

⁹Given an initial state and a length- L input sequence, one can in principle calculate the probability of being in a certain state after L steps; the overall probability distribution will depend on the structure of the particular dynamic system, on the error model and on whether multiple failures may lead to the correct state.

¹⁰We do not necessarily have to feed back the correct state at the end of each time step; if we feed it back after τ steps, however, we need to ensure that $(1 - p_s)^\tau$ does not become too small.

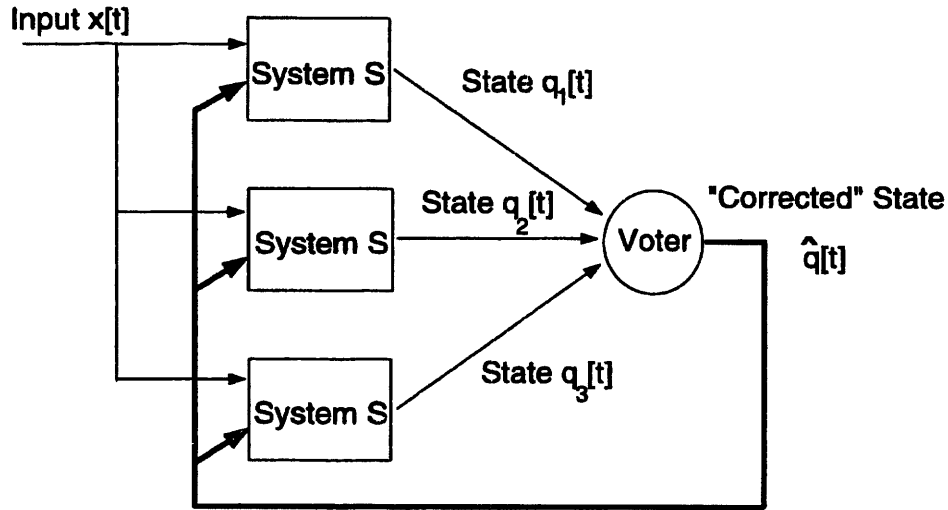


Figure 1-2: Triple modular redundancy with correcting feedback.

1. System replication may be unnecessarily expensive. In fact, this was the original motivation for arithmetic coding and ABFT schemes (namely the development of fault-tolerant computational systems that make better use of redundancy by taking into consideration the algorithmic structure of a given computational task).
2. The scheme relies heavily on the assumption that the voter is fault-free. If the voter also fails independently between time steps (e.g., with probability p_v a voter outputs a state that is different from the state at which the majority of the systems agree), then we face a problem: after L time steps the probability that the modular redundancy scheme performs correctly is at best $(1 - p_v)^L$ (ignoring the probability that a failure in the voter may accidentally result in feeding back the correct state in cases where most systems are in an incorrect state). Similarly, the probability that the majority of the replicas is in the correct state after L time steps is also very low. Clearly, given unreliable voters there appears to be a limit on the number of time steps for which we can guarantee reliable evolution using a simple replication scheme. Moreover, in a dynamic system setting, failures in the voting mechanism become more significant as we increase the number of time steps for which the fault-tolerant system operates. Therefore, even if p_v is significantly smaller than p_s (e.g., because the system is more complex than the voter), the probability that the modular redundancy scheme performs correctly is bounded above by $(1 - p_v)^L$ and will eventually become unacceptably

small for large enough L .

In this thesis we deal with both of the above problems. We initially aim to protect against a *pre-specified* number of failures using a *fault-free* correcting mechanism. To achieve this while avoiding replication and while using the least amount of redundancy, we introduce the concept of a *redundant implementation*, that is, a version of the dynamic system which is redundant and follows a restricted state evolution. Redundant implementations range from no redundancy to full replication and give us a way of characterizing and parameterizing constructions that are appropriate for fault tolerance. The thesis demonstrates and exploits certain flexibilities that exist when constructing redundant implementations. We make no systematic attempt to choose from among these different redundant implementations ones that are optimal according to a particular criterion; our examples, however, illustrate how such questions may be posed in future work.

We also address the case when failures in each component happen with constant probability, independently between different components and independently between time steps. This problem is much harder, as we can no longer guarantee that the fault-tolerant system will be in the right state at the end of each time step. We introduce techniques that deal with *transient failures in the error-correcting mechanism* by developing and analyzing the performance of a *distributed* voting scheme. Our approach uses redundancy in a way that ensures that, with high probability, the fault-tolerant system will be within a *set* of states that represent (and can be decoded to) the actual state; the goal then becomes to make efficient use of redundancy while achieving any given probability of failure. For example, by increasing redundancy we can increase the probability that a fault-tolerant system follows the correct state trajectory for a certain time interval. Our analysis is very general and provides a better understanding of the tradeoffs that are involved when designing fault-tolerant systems out of unreliable components. These include constraints on the probabilities of failure in the system/corrector, the length of operation and the required amount of redundancy.

1.3.1 Redundant Implementation

In order to avoid replication when constructing fault-tolerant dynamic systems, we replace the original system with a larger, redundant system that preserves the state, evolution and

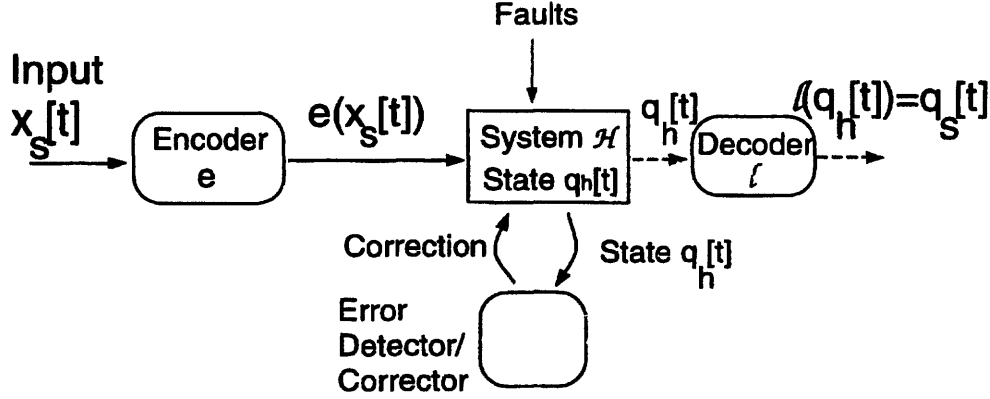


Figure 1-3: Thesis approach for fault tolerance in a dynamic system.

properties of the original system — perhaps in some encoded form. We impose restrictions on the set of states that are allowed in the larger dynamic system, so that an external mechanism can perform error detection and correction by identifying and analyzing violations of these restrictions. The larger dynamic system is called a *redundant implementation* and is part of the overall fault-tolerant structure shown in Figure 1-3: the input to the redundant implementation at time step t , denoted by $e(x_s[t])$, is an encoded version of the input $x_s[t]$ to the original system; furthermore, at any given time step t , the state $q_s[t]$ of the original dynamic system can be recovered from the corresponding state $q_h[t]$ of the redundant system through a decoding mapping l (i.e., $q_s[t] = l(q_h[t])$). Note that we require the error detection/correction procedure to be input-independent, so that we ensure the next-state function is *not* evaluated in the error-correcting circuit.

The following definition formalizes the notion of a *redundant implementation* for a dynamic system:

Definition 1.2 Let S be a dynamic system with state set Q_S , input set X_S , initial state $q_s[0]$ and state evolution

$$q_s[t + 1] = \delta_S(q_s[t], x_s[t]) ,$$

where $q_s[\cdot] \in Q_S$, $x_s[\cdot] \in X_S$ and δ_S is the next-state function. A dynamic system \mathcal{H} with

state set $Q_{\mathcal{H}}$, input set $X_{\mathcal{H}}$, initial state $q_h[0]$ and state evolution equation

$$q_h[t+1] = \delta_{\mathcal{H}}(q_h[t], e(x_s[t]))$$

(where $e : X_{\mathcal{S}} \mapsto X_{\mathcal{H}}$ is an injective input encoding mapping) is a redundant implementation for \mathcal{S} if it concurrently simulates \mathcal{S} in the following sense: there exists one-to-one state decoding mapping $\ell : Q'_{\mathcal{H}} \mapsto Q_{\mathcal{S}}$ such that

$$\ell(\delta_{\mathcal{H}}(\ell^{-1}(q_s[t]), e(x_s[t]))) = \delta_{\mathcal{S}}(q_s[t], x_s[t])$$

for all $q_s[\cdot] \in Q_{\mathcal{S}}$, $x_s[\cdot] \in X_{\mathcal{S}}$. The set $Q'_{\mathcal{H}} = \ell^{-1}(Q_{\mathcal{S}}) \subset Q_{\mathcal{H}}$ is called the subset of valid states in \mathcal{H} .

If we initialize the redundant implementation \mathcal{H} to state $q_h[0] = \ell^{-1}(q_s[0])$ and encode the input $x_s[\tau]$ using the encoding mapping e , the state of \mathcal{S} at all discrete-time steps $\tau \geq 0$ can be recovered from the state of \mathcal{H} through the decoding mapping ℓ (under fault-free conditions at least); this can be proved easily by induction. Knowledge of the subset of valid states allows an external error detecting/correcting mechanism to handle failures. Any failures that cause transitions to *invalid* states (i.e., states outside the subset $Q'_{\mathcal{H}} = \{q'_h[\cdot] = \ell^{-1}(q_s[\cdot]) \mid \forall q_s[\cdot] \in Q_{\mathcal{S}}\}$) will be detected and perhaps corrected.

During each time step, the redundant implementation \mathcal{H} evolves to a (possibly corrupted) next state. We then perform error detection/correction by checking whether the resulting state is in the subset of valid states $Q'_{\mathcal{H}}$ and by making appropriate corrections when necessary. When we apply this general approach to specific dynamic systems, we manage to parameterize different redundant implementations, develop appropriate error models, make connections with hardware and systematically devise schemes capable of detecting/correcting a fixed number of failures.

Note that our definition of a redundant implementation does not specify next-state transitions when the redundant system is in a state outside the set of valid states¹¹. Due to this flexibility, there are multiple different redundant implementations for a given error detecting/correcting scheme. In many cases we will be able to systematically characterize

¹¹This issue becomes very important when the error detector/corrector is *not* fault-free.

and exploit this flexibility to our advantage (e.g., to minimize hardware or to perform error detection/correction periodically).

1.3.2 Error-Correction

In Chapter 6 we describe how to handle transient failures¹² in both the redundant implementation and the error-correcting mechanism. We assume that components in our systems can suffer transient failures (more specifically, we assume that they can fail independently between components and between time steps) and describe implementations that operate with an arbitrarily small probability of failure for a specified (finite) number of steps. In particular, given an unreliable dynamic system (e.g., one that takes an incorrect state transition with probability p_s at any given time step) and unreliable voters (that fail with probability p_v), we describe ways to guarantee that the state evolution of a redundant fault-tolerant implementation will be correct with high probability for any specified (finite) number of steps. Our scheme is a variation of modular redundancy and is based on using a distributed set of voters. We show that, under this very general approach, there is a logarithmic trade-off between the number of time steps and the amount of redundancy. In other words, if we want to maintain a given probability of failure while doubling the number of time steps for which our system operates, we need to increase the amount of redundancy by a constant amount. For the case of linear finite-state machines, we show that there are efficient ways of protecting many *identical* machines that operate in *parallel* on *distinct* input sequences. In this special setting, our approach can achieve a low probability of failure for any finite time interval using only a *constant* amount of redundancy per machine.

Our techniques relate well to existing techniques that have been used in the context of “reliable computational circuits” or “stable memories”. As in those cases, our approach can provide fault tolerance to a dynamic system (that is, low probability of failure over any pre-specified finite time interval) at the expense of system replication. More specifically, given a certain time interval, we can achieve a low probability of failure by increasing the

¹²Permanent failures can be handled more efficiently using reconfiguration techniques rather than concurrent error detection and correction. In some sense, permanent failures are easier to deal with than transient failures. For example, if we are testing for permanent failures in an integrated circuit, it may be reasonable to assume that our testing mechanism (error-detecting mechanism) has been verified to be fault-free. Since such verification only needs to take place once, we can devote large amounts of resources and time in order to ensure the absence of permanent failures in this testing/correcting mechanism.

amount of redundancy; alternatively, for a given probability of failure, we can increase operation time (i.e., the length of time for which the fault-tolerant system needs to operate reliably for) by increasing the amount of redundancy. Our method ensures that, with high probability, our fault-tolerant system will go through a sequence of states that *correctly represent* the fault-free state sequence. More specifically, at each time step, the state of the redundant system is within a set of states that can be decoded to the state in which the fault-free system would be in. Note that this is the best we can do since all components at our disposal can fail.

1.4 Scope and Major Contributions of the Thesis

Fault tolerance in dynamic systems has traditionally been addressed using techniques developed for fault tolerance in computational circuits. This thesis generalizes these techniques, studies the implications that the dynamic nature of systems has on fault tolerance and develops a framework that encompasses most previous schemes for concurrent error detection and correction in dynamic systems.

Adopting the traditional assumption that the error detecting/correcting mechanism is fault-free, we describe fault-tolerant schemes that protect against a specified, constant number of failures. Our approach is systematic and our goal (in each of the cases we study) is two-fold: (i) develop appropriate error models and techniques that satisfy the error detecting/correcting requirements, and (ii) parameterize redundant implementations for use in conjunction with a given error detecting/correcting mechanism. We study a variety of different dynamic systems, specifically those listed below, and in some cases we are able to make explicit connections with redundant hardware implementations, hardware failure modes, and error detecting/correcting techniques. We do not specifically address the issue of choosing an optimal redundant implementation (e.g., one that minimizes the hardware or cost involved), but we point out related questions in our examples.

Algebraic Machines: We develop appropriate (algebraic) encoding and decoding mappings for group/semigroup machines and demonstrate that each encoding/decoding pair has a number of possible redundant implementations, which may offer varying fault coverage. Our approach in this setting is purely algebraic and hardware-

independent. We do not make connections with actual hardware implementations and hardware failure modes, but use algebraic techniques to illustrate how different (algebraic) machine decompositions may capture different sets of errors/failures. In particular, we show that certain decompositions are undesirable because the failures of interest are always undetectable. We also extend these results to redundant implementations of finite semiautomata. The virtue of our approach is that it focuses on the desired functionality of a redundant implementation and not on the specifics of a particular hardware construction; this allows the development of novel fault-tolerant hardware constructions.

Linear Dynamic Systems: We study redundant implementations for linear time-invariant dynamic systems and linear finite-state machines, obtaining in each case a characterization of all redundant implementations with states that are *linearly* constrained (encoded according to a linear code). We show that within this class of redundant implementations each pair of encoding and decoding mappings permits a variety of state evolution mechanisms. Thus, there is some flexibility in terms of choosing the redundant implementation which was not considered in previous work. A variant of our core result has been known in the control community (for continuous-time linear dynamic systems) but was not developed in the context of fault tolerance. Our approach results in a systematic way of constructing redundant implementations and allows us to make explicit connections with hardware constructions and hardware failure modes. In addition, using this flexibility we demonstrate examples of implementations that require less hardware than traditional ones, and new schemes for fault tolerance (including parity check schemes with memory).

Petri Nets: Following a similar approach, we systematically develop embeddings of Petri net models of discrete event systems (DES). The idea is based on enforcing constraints on the state (*marking*) of a given Petri net in way that retains its properties and overall functionality while allowing easy detection and identification of failures that may occur in the underlying DES. This leads to monitoring schemes for DES of interest, such as network protocols or manufacturing systems. The approach is general and can handle a variety of error models. We focus primarily on *separate embeddings* in which

the functionality of the original Petri net is retained in its exact form. Using these embeddings we construct monitors that operate concurrently with the original system and allow us to detect and identify different types of failures by performing consistency checks between the state of the original Petri net and that of the monitor. The methods that we propose are attractive because the resulting monitors are robust to failures, they may not require explicit acknowledgments from each activity and their construction is systematic and easily adaptable to restrictions in the available information. We also discuss briefly how to construct *non-separate* Petri net embeddings. There are a number of interesting directions that emanate from this work, particularly in terms of optimizing our embeddings (e.g., to minimize communication cost or other quantities of interest). We do not explicitly address such optimization questions but rather focus on establishing this new approach, highlighting its potential advantages, and describing the different parameters of the problem.

Unlike the situation in static (computational) circuits, fault tolerance in dynamic systems requires considerations about error propagation, and forces us to consider the possibility of failures in the error detecting/correcting mechanism. The problem is that a failure causing a transition to an incorrect next state at a particular time step will not only affect the output at a particular time step (which may be an unavoidable possibility, given that we use failure-prone elements), but will also affect the state (and therefore the output) of the system at later times. In addition, the problem of error propagation intensifies as we increase the number of time steps for which the dynamic system operates. On the contrary, failures in the implementation of static circuits only affect the output at a particular time step but have *no* aftereffects on the future performance of the systems (they do not intensify as we increase the number of time steps for which the systems operate).

The thesis addresses the problem of failures in the error detecting/correcting mechanism and shows that our two-stage approach to fault tolerance can be used successfully (and, in some cases that we illustrate, efficiently) to construct reliable systems out of unreliable components. First, we develop a distributed voting scheme and show how it can be used to construct redundant systems that evolve reliably for any given finite number of time steps. Our approach is novel, but related techniques can be found in computational circuits and

stable memories. By combining this distributed voting scheme with low-complexity error-correcting codes, we construct interconnections of identical linear finite-state machines that operate in parallel on distinct inputs and use a *constant* amount of hardware per machine in order to achieve a desired low probability of failure for any finite number of time steps. We also make comparisons and connections with related work, and point out interesting future directions and possible improvements to our construction.

1.5 Outline of the Thesis

This thesis is organized as follows:

Chapters 2 through 5 systematically explore the concurrent error detection/correction approach of Figure 1-3 for different dynamic systems under the assumption that the error-correcting mechanism is fault-free. In Chapter 2 we focus on algebraic machines (group and semigroup machines, and finite semiautomata); in Chapters 3 and 4 we study redundant implementations for linear time-invariant dynamic systems and linear finite-state machines; in Chapter 5 we use similar ideas to construct Petri net embeddings and obtain robust monitoring schemes for discrete event systems.

In Chapter 6 we develop ways to handle failures in the error-correcting mechanism, both for the general case and also for the special case of linear finite-state machines.

We conclude in Chapter 7 with a summary of our results and future research directions.

Chapter 2

Redundant Implementations of Algebraic Machines

2.1 Introduction

In this chapter we develop a general, hardware-independent characterization of fault-tolerant schemes for *group/semigroup machines* and for *finite semiautomata*. More specifically, we use homomorphic embeddings to construct redundant implementations for algebraic machines, describe the corresponding error detection/correction techniques, and demonstrate that for a particular encoding/decoding scheme there exist many possible redundant implementations, each offering potentially different fault coverage.

Throughout our development, we assume that the error detecting/correcting mechanism is fault-free¹ and focus on algebraically characterizing redundant implementations. We assume a hardware-independent error model in which failures cause incorrect state transitions in the redundant machine. In later chapters of the thesis the fruits of our abstract approach become clearer, as we make explicit connections to hardware implementations and hardware failures. For example, in Chapters 3 and 4 we outline such extensions for linear time-invariant dynamic systems (implemented using adder, gain and memory elements)

¹As mentioned in the Introduction, the assumption that the error detecting/correcting mechanism is fault-free appears in most concurrent error detection and correction schemes. It is a reasonable assumption in many cases, particularly if the error checking mechanism is much simpler than the state evolution mechanism. In Chapter 6 we extend our approach to handle failures in the error detecting/correcting mechanism.

and linear finite-state machines (implemented using XOR gates and flip-flops).

This chapter is organized as follows. In Section 2.2 we provide background on the use of group/semigroup homomorphisms in constructing fault-tolerant computational systems, [43, 44, 8, 9]. Then, in Section 2.3, we develop redundant implementations for *group* and *semigroup machines* (in Sections 2.3.1 and 2.3.2 respectively). Our approach results in an algebraic characterization of the different redundant implementations under a given encoding/decoding scheme and also leads to discussions about the role of machine decomposition. In Section 2.4 we make connections with redundant implementations for finite semiautomata. Finally, in Section 2.5 we summarize the theoretical approach of this chapter and the key insights that it has provided.

2.2 Background: Fault-Tolerant Computation in Groups and Semigroups

Before we discuss fault tolerance in algebraic machines, we present some previous results on fault-tolerant computation in systems with algebraic structure.

2.2.1 Fault Tolerance in Abelian Group Computations

A group (G, \circ) is a set of elements G together with a binary operation \circ such that the following are satisfied:

- For all $g_1, g_2, g_3 \in G$, $g_1 \circ g_2 \in G$ (closure) and $g_1 \circ (g_2 \circ g_3) = (g_1 \circ g_2) \circ g_3$ (associativity).
- There is an element 1_\circ , called the *identity element* such that for all $g \in G$, $g \circ 1_\circ = 1_\circ \circ g = g$.
- For every $g \in G$, there is an *inverse element* $g^{-1} \in G$ such that $g^{-1} \circ g = g \circ g^{-1} = 1_\circ$.

An *abelian* group also satisfies commutativity:

- For all $g_1, g_2 \in G$, $g_1 \circ g_2 = g_2 \circ g_1$.

A computation that takes place in an abelian group is protected in [8] by a coding scheme like the one shown in Figure 1-1. Redundancy is added to the operands by the encoding

mappings ϕ_1 and ϕ_2 , which map operands in the abelian group (G, \circ) (e.g., g_1 and g_2 in the figure) to elements in a *larger* abelian group (H, \diamond) (these elements are denoted by h_1 and h_2 in the figure). The original group operation \circ in G is replaced by the redundant group operation \diamond in H . Ideally, under no failures, the result $r = g_1 \circ g_2$ can be obtained via the decoding mapping σ from the result $\rho = h_1 \diamond h_2$ in the redundant group (i.e., $r = \sigma(\rho)$). The subset of valid results in H is given by the set $G' = \{\phi_1(g_1) \diamond \phi_2(g_2) \mid g_1, g_2 \in G\}$. The objective is to utilize the redundancy that exists in H to provide fault tolerance for the computation in G . By imposing the requirement that under fault-free conditions the decoding mapping $\sigma : G' \rightarrow G$ be one-to-one, it can be shown that the encoding mappings ϕ_1 and ϕ_2 need to be the same mapping, which we denote by ϕ , and that $\sigma^{-1} = \phi$. Moreover, ϕ is shown to be a *group homomorphism*: for all $g_1, g_2 \in G$, we have $\phi(g_1) \diamond \phi(g_2) = \phi(g_1 \circ g_2)$.

Under the homomorphic mapping ϕ , the subset of valid results G' forms a *subgroup* of H that is isomorphic to G . If we assume that failures in the computation keep us in H (i.e., failures do not cause the computation to hang or behave in some unpredictable way, but simply result in an incorrect group element), then any result that falls outside of G' is invalid and is detected as erroneous (which is the case for result ρ_f in Figure 2-1). In particular, if we model² ρ_f as $\phi(g_1) \diamond \phi(g_2) \diamond e \equiv \rho \diamond e$ (where e is an element in the *error set* $E = \{1_\diamond, e_1, e_2, \dots\}$), then error detection and correction are based on the structure of the cosets of G' in H (i.e., on the *factor* or *quotient group* H/G' , [49]). In the absence of failures results lie in the zero coset (that is, in G' itself). Every detectable error $e_d \in E$ forces the result of the computation into a non-zero coset (i.e., $G' \diamond e_d \neq G'$), while every correctable error $e_c \in E$ forces the result into a coset that is uniquely associated with that particular error (i.e., $G' \diamond e_c \neq G' \diamond e_j$ for every e_c, e_j in E such that $e_j \neq e_c$). In Figure 2-1, if e is a correctable error, then $\hat{\rho} = \rho$ and $\hat{r} = r$.

One of the most important results in [8] (also presented in [9]) is obtained for the special case of *separate* codes. These are codes in which redundancy is added through a separate “parity” computational system. In this case, the redundant group H is the cartesian product $G \times T$, where T is the group of parity symbols. Finding a suitable encoding homomorphism

²Since H is a group, we can always model the result of a computation as $\rho_f = \rho \diamond e$ where $e = \rho^{-1} \diamond \rho_f$. Therefore, given all possible hardware failures, we can generate the set of errors E . The identity 1_\diamond is included in E so that we can handle the fault-free case.

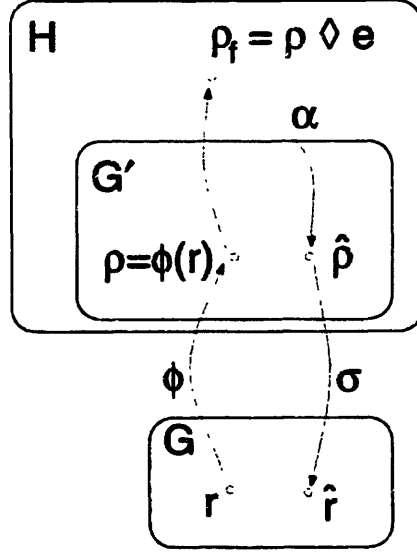


Figure 2-1: Fault tolerance in a group computation using a homomorphic mapping.

reduces to finding a homomorphism π such that $[g, \pi(g)]$ is the element of $H = G \times T$ corresponding to the operand g . If we impose the requirement that π be surjective (onto), the problem of finding all possible parity codings reduces to that of finding all surjective homomorphisms (epimorphisms) from G onto T (unlike ϕ , mapping π maps G onto a *smaller* group T). This is a reasonable requirement because if π was not onto, then T would contain elements that are never used by the parity computation (and can therefore be eliminated). By an important homomorphism theorem from group theory [49], these epimorphisms are isomorphic to the *canonical* epimorphisms, namely those that map G to its quotient groups G/N , where N denotes a (normal³) subgroup of G . Hence the problem of finding all possible parity codings reduces to that of finding all possible subgroups of G .

By simply exploiting the abelian group structure, the above results were extended in [8, 9] to higher algebraic systems with an embedded group structure, such as rings, fields and vector spaces. The framework thereby embraces a large variety of arithmetic codes and Algorithm-Based Fault Tolerance schemes already developed in some other way. In the following example we discuss how αM and parity check codes can be analyzed within the abelian group framework; additional examples can be found in [8].

³A subgroup N of a group (G, \circ) is called *normal* if for all $g \in G$, the set of elements $g \circ N \circ g^{-1}$ is contained in N . In the abelian group case considered in [8], any subgroup is trivially normal ($g \circ N \circ g^{-1} = g \circ g^{-1} \circ N = N$).

Example 2.1 αM -codes provide fault tolerance to modulo- M addition (which takes place in Z_M , the cyclic group of order M) by multiplying each of the operands by an integer α and performing modulo- αM addition, [87]. Such codes can be placed into the abelian group framework by using the injective homomorphism $\phi : G = Z_M \mapsto H = Z_{\alpha M}$ such that for $g \in Z_M$, $\phi(g) = \alpha g$. Naturally, the specifics of the errors that can be detected or corrected depend very much on the actual hardware implementation and on the particular values of M and α . For instance, with $M = 5$ and $\alpha = 11$, this arithmetic code can detect all failures that result in single-bit errors in a digitally implemented binary adder, [87]. In these implementations the operands and the result have a binary representation (i.e., 0 maps to 00000, 1 maps to 00001, etc.) and failures flip a “0” to a “1” and vice-versa.

An alternative way of providing fault tolerance to modulo- M addition is by performing a parity check using a separate (parity) adder alongside the original one. Using the algebraic framework we are able to enumerate and identify all appropriate parity computations (additions) that can be performed. More specifically, we know that each parity computation needs to lie in a group T that is isomorphic to a quotient group Z_M/N for a normal subgroup N of Z_M . Using standard results on cyclic groups [55], we conclude that all such groups T are isomorphic to Z_P where P is a divisor of M . Therefore, all parity computations for modulo- M addition are given by modulo- P addition (where P is a divisor of M). \square

2.2.2 Fault Tolerance in Semigroup Computations

The results for the abelian group case were extended to computations occurring in a *semigroup* in [43, 44]. A semigroup (S, \circ) is a set of elements S that is closed under an associative binary operation (denoted by \circ). Clearly, every group is a semigroup; familiar examples of semigroups that are not groups are the set of integers under the operation of multiplication, the set of *nonnegative* integers under addition and the set of polynomials with real-number coefficients under the operation of polynomial multiplication. All of the above examples are *abelian* semigroups in which the underlying operation \circ is commutative (for all $s_1, s_2 \in S$, $s_1 \circ s_2 = s_2 \circ s_1$). Examples of non-abelian semigroups are the set of polynomials under polynomial substitution and the set of $M \times M$ matrices under matrix multiplication. Other semigroups, as well as theoretical analysis, can be found in [65, 66].

A semigroup S is called a *monoid* when it possesses an *identity* element. The identity

element, denoted by 1_\circ , is the *unique* element that satisfies $s \circ 1_\circ = 1_\circ \circ s = s$ for all $s \in S$. We can focus on monoids without loss of generality because an identity element can always be adjoined to a semigroup that does not initially possess one. (The construction is straightforward: let $S^1 = S \cup \{1_\circ\}$ and define $s \circ 1_\circ = 1_\circ \circ s = s$ for all $s \in S^1$; all other products in S^1 are defined just as in S . By definition, element 1_\circ is the identity of S^1 .)

In order to protect a computation in a monoid (S, \circ) , we follow the model of Figure 1-1. To introduce the redundancy needed for fault tolerance, we map the computation $s_1 \circ s_2$ in (S, \circ) to a computation $\phi_1(s_1) \diamond \phi_2(s_2)$ in a *larger* monoid (H, \diamond) . The encoding mappings ϕ_1 and ϕ_2 are used to encode the first and second operands respectively (the results can be generalized to more than two operands). After performing the redundant computation $\phi_1(s_1) \diamond \phi_2(s_2)$ in H , we obtain a (possibly faulty) result ρ_f , which we assume still lies in H . Again, we perform error-correction through a mapping α and decoding through a one-to-one mapping $\sigma : S' \rightarrow S$ (where $S' = \{\phi_1(s_1) \diamond \phi_2(s_2) \mid s_1, s_2 \in S\}$ is the subset of valid results in H).

Under fault-free conditions, the decoding mapping σ satisfies:

$$\sigma(\phi_1(s_1) \diamond \phi_2(s_2)) = s_1 \circ s_2$$

for all $s_1, s_2 \in S$. Since we have assumed that σ is one-to-one, the inverse mapping $\sigma^{-1} : S' \rightarrow S$ is well-defined and satisfies $\sigma^{-1}(s_1 \circ s_2) = \phi_1(s_1) \diamond \phi_2(s_2)$. If we assume further that both ϕ_1 and ϕ_2 map the identity of S to the identity of H , then by setting $s_2 = 1_\circ$, we get $\sigma^{-1}(s_1) = \phi_1(s_1)$ for all $s_1 \in S$ (because $\phi_2(1_\circ) = 1_\circ$). Similarly, $\sigma^{-1}(s_2) = \phi_2(s_2)$ for all $s_2 \in S$, and we conclude that $\sigma^{-1} = \phi_1 = \phi_2 \equiv \phi$. Note that (i) $\phi(s_1 \circ s_2) = \phi(s_1) \diamond \phi(s_2)$, and (ii) $\phi(1_\circ) = 1_\circ$. Condition (i) is the defining property of a semigroup homomorphism, [66, 65]. A *monoid homomorphism* is additionally required to satisfy condition (ii), [55, 42]. Mapping ϕ is thus an injective monoid homomorphism, which maps the original computation in S into a larger monoid that contains an isomorphic copy of S .

The generalization of the framework of [8] to monoids allows non-abelian computations, for which inverses might not exist, to be treated algebraically. The generalization to monoids, however, comes at a cost since error detection and correction can no longer be based on coset constructions. The problem is two-fold: first, in a semigroup setting we

may be unable to model the possibly faulty result ρ_f as $\phi(s_1) \circ \phi(s_2) \circ e$ for some element e in H (because inverses do not necessarily exist in H and because the semigroup may be non-abelian); second, unlike the subgroup G' of valid results, the subsemigroup S' does not necessarily induce a natural partitioning⁴ on the semigroup H . (For instance, it is possible that the set $S' \circ h$ is a strict subset of S' for all $h \in H$.) Conditions for single-error detection and correction are discussed in Appendix A.

If the redundant monoid H is a cartesian product of the form $S \times T$, where (S, \circ) is the original monoid and (T, \odot) is the “parity” monoid, then the corresponding encoding mapping ϕ can be expressed as $\phi(s) = [s, \pi(s)]$ for all $s \in S$ and an appropriate mapping π . In such case, the set of valid results is given by $\{[s, \pi(s)] \mid s \in S\}$ and error-detection simply verifies that the result is of this particular form.

Using the fact that the mapping ϕ is a homomorphism, we can easily show that the parity mapping π is a homomorphism as well. As in the case of abelian groups, if we restrict this parity mapping to be surjective, we can obtain a characterization of all possible parity mappings and, thus, of all separate codes. However, the role that was played in the abelian group framework by the (normal) subgroups N of the group G is now played by the so-called *congruence relations* in S . Just as a normal subgroup induces a partitioning of a group (into the normal subgroup and the corresponding set of cosets), a congruence relation induces a partitioning of a monoid. Unlike the group case, however, the number of elements in each partition is not necessarily the same. In order that a given partitioning $\{P_i\}$ correspond to a congruence relation, the partitions need to be preserved by the monoid operation: when an element of partition P_j is composed with an element of partition P_k , the result must be confined to a single partition P_l (i.e., for all $s_j \in P_j$ and all $s_k \in P_k$ the products $s_j \circ s_k$ lie in partition P_l). Note that this is also true for the partitioning of a group into cosets. More formally, an equivalence relation \sim on the elements of a monoid S is called a congruence relation if, for all $a, a', b, b' \in S$, $a \sim a', b \sim b' \Rightarrow aob \sim a'ob'$. The partitions are referenced to as *congruence classes*.

An example of a partitioning into congruence classes is shown in Figure 2-2 for the semigroup (\mathbb{N}, \times) of positive integers under multiplication. Congruence class A contains

⁴A partitioning of a set S is a collection of *disjoint* subsets $\{P_i\}$, the union of which forms the set S .

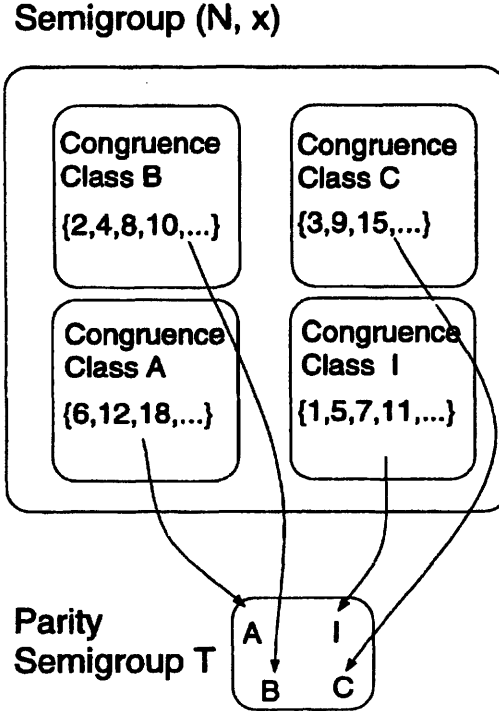


Figure 2-2: Partitioning of semigroup (N, \times) into congruence classes.

multiples of 2 *and* 3 (i.e., multiples of 6); congruence class *B* contains multiples of 2 but *not* 3; congruence class *C* contains multiples of 3 but *not* 2; and congruence class *I* contains all the remaining positive integers (i.e., integers that are neither multiples of 2 nor 3). One easily checks that the partitioning is preserved under the semigroup operation.

Let S/\sim denote the set of equivalence classes of S under congruence relation \sim . For two congruence classes $[a], [b]$ (where $[a]$ denotes the congruence class containing a), we define a binary operation \otimes by $[a] \otimes [b] = [a \circ b]$ (note that \otimes is well-defined if \sim is a congruence relation). With this definition, $(S/\sim, \otimes)$ is a monoid, referred to as the *factor* or *quotient* monoid of \sim in S and congruence class $[1_0]$ functions as its identity element.

If we apply a homomorphism theorem from semigroup theory [66, 65], which is the natural generalization of the theorem used earlier in the group case, we get that: surjective homomorphisms from S onto T are isomorphic to the *canonical* surjective homomorphisms, namely those that map S to its quotient semigroups S/\sim , where \sim denotes a congruence relation in S . The semigroup (T, \odot) is isomorphic to $(S/\sim, \otimes)$ for a suitable congruence relation \sim . Thus, for each congruence relation \sim , there is a corresponding surjective ho-

homomorphism and, for each surjective homomorphism, there is a corresponding congruence relation. Effectively, the problem of finding all possible parity codes reduces to that of finding all possible congruence relations in S .

When comparing these results with the abelian group case in [8], we find one major difference: in the abelian group case, finding a subgroup N of the group G completely specifies the parity homomorphism π because the inverse images of the elements of the parity group T are exactly the cosets of G with respect to the subgroup N (this is simply saying that $T \cong G/N$). In the more general setting of a monoid, however, specifying a normal subsemigroup for S does not completely specify the homomorphism π (and therefore does not determine the structure of the parity monoid T). In order to define the surjective homomorphism $\pi : S \mapsto T$ (or, equivalently, in order to define a congruence relation \sim on S), we may need to specify *all* congruence classes⁵.

2.3 Redundant Implementations of Algebraic Machines

In this section we construct redundant implementations of dynamic systems with algebraic structure, such as *group* and *semigroup machines*, using the preceding approach for fault tolerance in computational systems. We systematically develop *separate* and *non-separate* encodings that can be used by our two-stage concurrent error detection/correction scheme. Our approach is purely algebraic and aims at gaining insight for redundant implementations, error detection/correction techniques and appropriate error models. We show that algebraic homomorphisms can facilitate the design of fault-tolerant machines and the analysis of error detection and correction algorithms. We do not make connections to particular hardware constructions and hardware failure modes; when we couple our results with techniques for machine *decomposition*, however, we obtain interesting insight regarding the use of redundancy in non-separate implementations and regarding the functionality of separate monitors.

We start with group machines (in Section 2.3.1) and then generalize our approach to semigroup machines (Section 2.3.2) and finite semiautomata (Section 2.4). A *finite-state*

⁵This makes the search for encodings in a monoid setting more complicated than the search for such encodings in an abelian group setting. As the examples in [43] showed, however, we have a larger variety to choose from.

machine (FSM) has a finite set of states Q , a finite set of inputs X and a finite set of outputs Y . The *next-state function* is given by $\delta : Q \times X \mapsto Q$ and specifies the next state based on the current state and the current input. The *output function*, given by the mapping $\lambda : Q \times X \mapsto Y$, specifies the current output based on the current state and input. (Functions δ and λ need not be defined for all pairs in $Q \times X$.) A *finite semiautomaton* is an FSM without outputs (or, equivalently, one whose state is its output). A *semigroup machine* is a finite semiautomaton whose states and inputs are drawn from a finite semigroup (S, \circ) , [2, 3]. The next-state function is given by $\delta(s_1, s_2) = s_1 \circ s_2$, where the current state s_1 and input s_2 are elements of (S, \circ) . In the special case when (S, \circ) is a group (not necessarily abelian), the machine is known as a *group* or *permutation machine*, [2, 3, 40].

Our analysis of redundant implementations for these algebraic machines will be hardware-independent; for discussion purposes, however, we will make reference to *digital implementations*, i.e., implementations of FSM's that are based on digital circuits. Thus, states are encoded as binary vectors and stored into arrays of single-bit memory registers (flip-flops); the next-state function and the output function (when applicable) are implemented by combinational logic. When a hardware failure occurs, the desired transition to a state q_i ($q_i \in Q$) with binary encoding $(q_{1i}, q_{2i}, \dots, q_{ki})$ is replaced by a transition to an incorrect state q_j with encoding $(q_{1j}, q_{2j}, \dots, q_{kj})$. We will say that a single-bit error occurs when the encoding of q_i differs from the encoding of q_j in exactly one bit-position⁶.

2.3.1 Redundant Implementations of Group Machines

The next-state function of a group machine is given by $\delta(g_1, g_2) = g_1 \circ g_2$, where both the current state g_1 and input g_2 are elements of a group (G, \circ) . Examples of group machines include additive accumulators, multi-input linear shift registers, counters and cyclic autonomous machines; group machines also play an important role as essential components of arbitrary state machines.

In order to construct redundant implementations of a group machine (G, \circ) (with state $g_1 \in G$, input $g_2 \in G$ and next-state function $\delta(g_1, g_2) = g_1 \circ g_2$), we *embed* it into a

⁶There are many other error models, such as the *stuck-at failure model* or the *delay failure model*, [31]. Note that, depending on the hardware implementation, a single hardware failure can cause *multiple-bit errors*.

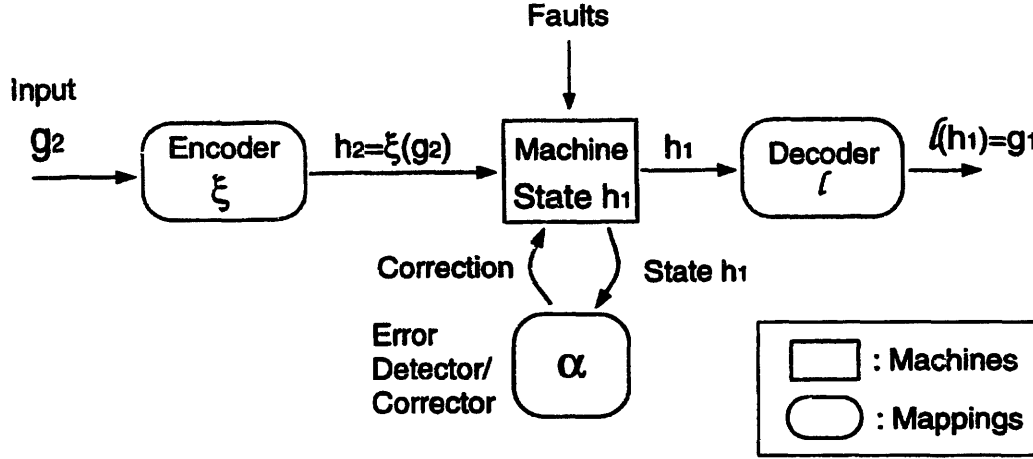


Figure 2-3: Error detection and correction in a redundant implementation of a group machine.

larger group machine (H, \diamond) (with state $h_1 \in H$, input $h_2 \in H$ and next-state function $\delta_H(h_1, h_2) = h_1 \diamond h_2$). As shown in Figure 2-3, machine H receives as input $h_2 = \xi(g_2)$ (which is an encoded version of the input g_2 that machine G would receive) and *concurrently simulates* G , so that, under fault-free operation, the state g_1 of the original group machine G can be recovered from the corresponding state h_1 of the redundant machine H through a *one-to-one* decoding mapping ℓ (i.e., $g_1 = \ell(h_1)$ at all time steps). The mapping ℓ is only defined for the subset of valid states in H , denoted by $G' = \ell^{-1}(G) \subset H$. Erroneous operations cause transitions to invalid states in H ; these errors will be detected and, if possible, corrected by the detector/corrector α at the end of the corresponding time step (for now, we assume that all mappings in Figure 2-3 are fault-free).

More formally we have the following definition for redundant implementations of group machines:

Definition 2.1 A redundant implementation for a group machine (G, \circ) is a group machine (H, \diamond) that concurrently simulates G in the following sense: there exist a one-to-one mapping $\ell : G' \mapsto G$ (where $G' = \ell^{-1}(G) \subset H$ is the subset of valid states) and an appropriate input encoding mapping $\xi : G \mapsto H$ (from G into H) such that the following condition holds for all $g_1, g_2 \in G$:

$$\ell(\ell^{-1}(g_1) \diamond \xi(g_2)) = g_1 \circ g_2. \quad (2.1)$$

Note that when H is properly initialized and fault-free, there is a one-to-one correspondence between the state h_1 of H and the corresponding state g_1 of G ; specifically, $g_1 = \ell(h_1)$ or $h_1 = \ell^{-1}(g_1)$ for all time steps. At the beginning of each time step, input $g_2 \in G$ is supplied to machine H encoded via ξ . The next state of H is then given by $h' = h_1 \diamond \xi(g_2) = \ell^{-1}(g_1) \diamond \xi(g_2)$; since ℓ is one-to-one, it follows easily from eq. (2.1) that h' has to satisfy $h' = \ell^{-1}(g_1 \circ g_2) = \ell^{-1}(g')$, where $g' = g_1 \circ g_2$ is the next state of machine G . Note that h' belongs to the subset of valid states $G' = \ell^{-1}(G) \subset H$. At the end of the time step, the error detector verifies that the newly reached state h' is in G' ; when an error is detected, necessary correction procedures are initiated and completed before the next input is supplied.

The concurrent simulation condition of eq. (2.1) is an instance of the coding scheme of Figure 1-1 (where we had $\sigma(\phi_1(g_1) \diamond \phi_2(g_2)) = g_1 \circ g_2$): the decoding mapping ℓ plays the role of σ , whereas ξ corresponds to mapping ϕ_2 . (The situation described in eq. (2.1) is actually slightly more restrictive than the one in Figure 1-1, because ϕ_1 is restricted to be ℓ^{-1} .) Therefore, the results of Section 2.2.2 apply and we can design redundant implementations for group machines by *homomorphically embedding* them into larger group machines. More specifically, by choosing $\xi \equiv \ell^{-1}$ to be an injective group homomorphism from G into H , we automatically satisfy (2.1):

$$\begin{aligned} \ell(\ell^{-1}(g_1) \diamond \xi(g_2)) &= \ell(\ell^{-1}(g_1) \diamond \ell^{-1}(g_2)) \\ &= \ell(\ell^{-1}(g_1 \circ g_2)) \\ &= g_1 \circ g_2 . \end{aligned}$$

Just as we did for the group/semigroup⁷ cases in Sections 2.2.1 and 2.2.2, we will use the notation ϕ in place of ℓ^{-1} and ξ , and σ in place of ℓ . With this in mind, the condition in eq. (2.1) simplifies to

$$\sigma(\phi(g_1) \diamond \phi(g_2)) = g_1 \circ g_2 \tag{2.2}$$

for all states g_1 and all inputs g_2 in G .

⁷Remember that this is *not* necessarily an *abelian* group.

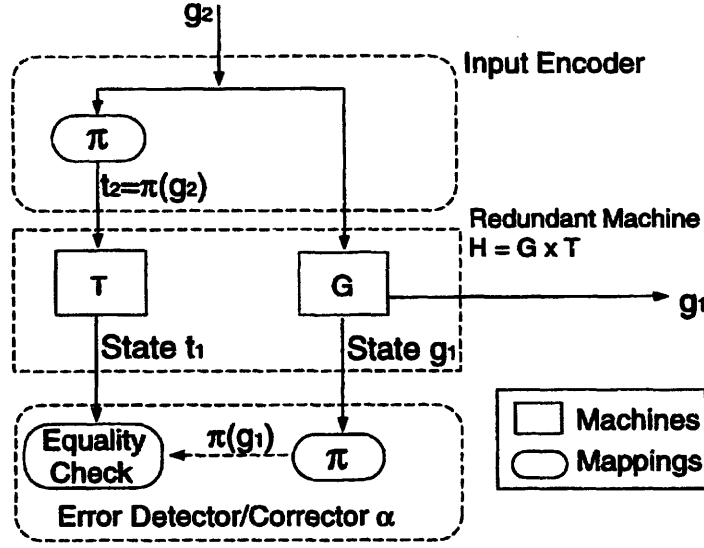


Figure 2-4: Monitoring scheme for a group machine.

When the redundant group machine is of the form $H = G \times T$, we recover the results obtained in Sections 2.2.1 and 2.2.2 for the separate case: the encoding homomorphism $\phi : G \mapsto H$ (where $\phi(g) = \xi(g) = \ell^{-1}(g)$) is of the form $\phi(g) = [g, \pi(g)]$ for an appropriate mapping π . If we assume that π is surjective, then the redundant machine H consists of the original machine G and an *independent* parity machine T as shown in Figure 2-4. Machine T is *smaller* than G and we will refer to it as a (*separate*) *monitor* or a *monitoring machine* (the latter term has been used in finite semiautomata [54, 78, 77], and in other settings). Mapping $\pi : G \mapsto T$ produces the encoded input $t_2 = \pi(g_2)$ of the separate monitor T (where g_2 is the input for G) and is easily shown to be a homomorphism, i.e., it satisfies

$$\pi(g_1) \odot \pi(g_2) = \pi(g_1 \circ g_2)$$

for all $g_1, g_2 \in G$. It can be easily shown that, if machines G and T are properly initialized and fault-free, then the state t of the monitor at any time step will be given by $t = \pi(g)$, where g is the corresponding state of the original machine G . Error-detection checks if this condition is satisfied. Depending on the actual hardware implementation and the error model, we may be able to detect and correct certain errors in the original machine and/or in the separate monitor.

Next, we use the approach outlined in eqs. (2.1) and (2.2) to discuss examples of separate

monitors for group machines.

Separate Monitors for Group Machines

In the previous section we concluded that the problem of designing a separate monitor T for a group machine G can be solved algebraically: using the results of Sections 2.2.1 and 2.2.2, and retaining the assumption that the mapping $\pi : G \mapsto T$ (which maps states and inputs in machine G to states and inputs in machine T) is surjective, we concluded that group machine (T, \odot) can monitor group machine (G, \circ) if and only if T is a surjective homomorphic image of G or, equivalently, if and only if there exists a normal subgroup N of G such that $T \cong G/N$.

Example 2.2 Consider the group machine $G = Z_6 = \{0, 1, 2, 3, 4, 5\}$ (i.e., a modulo-6 adder). The non-trivial⁸ (normal) subgroups of G are $N = \{0, 3\} \cong Z_2$ and $N' = \{0, 2, 4\} \cong Z_3$, resulting in quotient groups $G/N \cong Z_2$ and $G/N' \cong Z_3$ respectively.

If we decide to use Z_2 as a separate monitor, we need to partition the elements of Z_6 in two partitions as $\{p_0 = \{0, 2, 4\}, p_1 = \{1, 3, 5\}\}$. If the original machine is *digitally implemented* (i.e., using three bits to encode each of its states) and if the operation of the monitor is fault-free⁹, then in order to *detect* failures that result in *single-bit errors* in the *digital* implementation of Z_6 we need the binary encodings of states within the same partition to have Hamming distance¹⁰ greater than 1. If we consider the partitioning $\{p_0 = \{000, 011, 110\}, p_1 = \{111, 100, 001\}\}$ (which corresponds to a digital implementation in which 0 is encoded to 000, 1 to 111, 2 to 011, 4 to 110, and so on), we see that an error is detected whenever the state encoding does not lie in the partition specified by the presumably fault-free monitor or when the result is an invalid codeword. For example, if the monitor is in partition p_0 and if the current state is 111 or 010, a single-bit error has been detected. We see that, under the assumption that the monitor is fault-free, this particular

⁸The group $\{1_o\}$ (where 1_o denotes the identity element) is a trivial normal subgroup of any group.

⁹This assumption is realistic if the hardware implementation of the monitor is considerably simpler than the implementation of the actual machine.

¹⁰The Hamming distance between two binary vectors $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_n)$ is the number of positions at which x and y differ, [87]. The minimum Hamming distance d_{min} between the codewords of a code (collection of binary vectors of length n) determines its error detecting and correcting capabilities: a code can detect $d_{min} - 1$ single-bit errors; it can correct $\lfloor \frac{d_{min}-1}{2} \rfloor$ single-bit errors.

scheme can detect failures that result in single-bit errors in the original machine. Note, however, that a single hardware failure may not necessarily result in a single-bit error.

If we use Z_3 as the monitoring machine, we obtain the partitioning $\{p_0 = \{0, 3\}, p_1 = \{1, 4\}, p_2 = \{2, 5\}\}$. Again, to detect single-bit errors in a *digital implementation*, we require that states within the same partition have Hamming distance greater than 1, as for example in the encoding $\{p_0 = \{000, 011\}, p_1 = \{001, 100\}, p_2 = \{010, 101\}\}$ (which corresponds to encoding 0 to 000, 1 to 001, 2 to 010, and so on). If states within the same partition have Hamming distance at least 3 (partitioning $\{p_0 = \{000, 111\}, p_1 = \{001, 110\}, p_2 = \{101, 010\}\}$ is one such possibility), then we can actually correct single-bit errors. We use the fault-free monitor to locate the correct partition and, since codewords within the same partition have Hamming distance greater or equal to 3, we can correct single-bit errors by choosing as the correct state the one that has the smallest Hamming distance from the corrupted state. \square

Example 2.3 The authors of [78, 77] investigate separate monitors for *cyclic autonomous machines*. The only input to an autonomous machine is the clock input. The dynamics are therefore completely predetermined because from any given state only one transition is possible, and it occurs at the next clock pulse. Since the number of states is finite, an autonomous machine will eventually enter a cyclic sequence of states. A cyclic autonomous machine is one whose states form a pure cycle (i.e., there are no transients involved). Such a machine is essentially the cyclic group machine Z_M , but with only one allowable input (namely element 1) instead of the whole set $\{0, 1, 2, \dots, M-1\}$.

Using our algebraic framework and some rather standard results from group theory, we can characterize all possible monitors T for the autonomous machine Z_M : each monitor needs to be a group machine (T, \odot) that is isomorphic to Z_M/N , where N is a normal subgroup of Z_M . The (normal) subgroups for Z_M are exactly the cyclic groups of order $|N| = D$ that divides M , [55]; therefore, the monitors correspond to quotient groups $T \cong Z_M/N = Z_M/Z_D$ that are cyclic and of order $P = \frac{M}{D}$ (that is, $T \cong Z_P$). Of course, since only one input is available to the original machine G , we should restrict T to only one input as well. This results in a monitor that is a cyclic autonomous machine with P states (where P is a divisor of M).

Using graph-based constructions, the authors of [78, 77] concluded that the minimum number of states required for a separate monitor is the smallest prime factor of the cycle length. Our result is a generalization of that conclusion: using the algebraic framework we are able to describe the structure of all possible monitors (not only the ones with the minimum number of states) independently of the error model. More importantly, our result is obtained through simple applications of group theory and can be generalized to machines with non-trivial inputs and more complicated structure. \square

When N is a normal subgroup of G , we can actually decompose the group machine G into an interconnection of two simpler group machines. This and other results on *machine decomposition* introduce some interesting possibilities into our analysis of redundant implementations. For this reason, in the next section we briefly review some decomposition results and apply them to the analysis of separate monitors. Machine decomposition is important because the implementation of group machines (and FSM's in general) as interconnections of smaller components may result in an improved circuit design¹¹, [4].

Group Machine Decomposition

Whenever group G has a non-trivial *normal* subgroup N , the corresponding group machine can be decomposed into two smaller group machines: the *coset leader* machine with group G/N and the *subgroup* machine with group N , [2, 3, 40]. Figure 2-5 conveys this idea. Group machine G , with current state g_1 and input g_2 , can be decomposed into the "series-parallel" interconnection in the figure. Note that the input is encoded differently for each submachine. The overall state is obtained by combining the states of both submachines.

The above decomposition is possible because the normal subgroup N induces a partition of the elements of G into cosets, [2, 3]. Each element g of G can be expressed uniquely as

$$g = n \circ c_i \text{ for some } n \in N, c_i \in C,$$

where $C = \{c_1, c_2, \dots, c_l\}$ is the set of distinct (right) coset leaders (there is exactly one

¹¹Machine decomposition typically results in reductions of the chip area, of the longest path (between latch inputs and outputs) and of the clock duration. Furthermore, it tends to minimize the clock skew and utilize more efficiently the programmable gate arrays or logic devices (if FSM's are implemented using such technologies), [47, 4, 39].

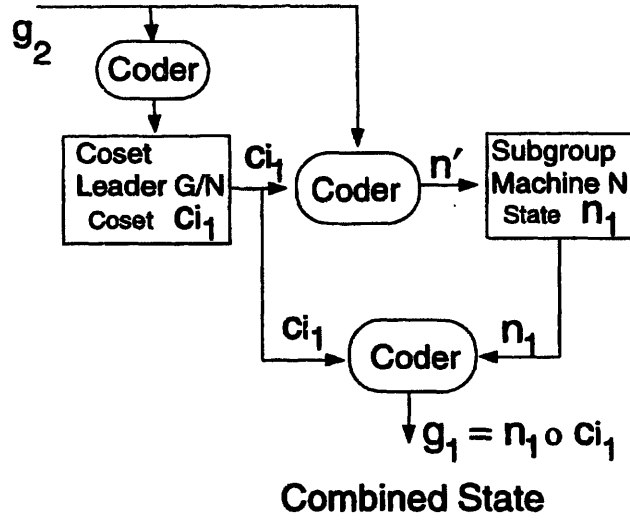


Figure 2-5: Series-parallel decomposition of a group machine.

representative for each coset). The decomposition in Figure 2-5 simply keeps track of this parameterization. If the machine is in state $g_1 = n_1 \circ c_{i_1}$ and an input $g_2 = n_2 \circ c_{i_2}$ is received, the new state can be expressed as $g_3 = n_3 \circ c_{i_3}$. One possibility is to take $c_{i_3} = \overline{c_{i_1} \circ g_2} = \overline{c_{i_1} \circ n_2 \circ c_{i_2}}$ (here, \overline{x} denotes the coset leader of the element $x \in G$); then, we put $n_3 = n_1 \circ c_{i_1} \circ g_2 \circ (\overline{c_{i_1} \circ g_2})^{-1}$. Note that $c_{i_1} \circ g_2 \circ (\overline{c_{i_1} \circ g_2})^{-1}$ is an element of N and therefore this group operation can be computed within the subgroup machine¹². The encoders are used to appropriately encode the input for each machine and to provide the combined output. The decomposition can continue if either of the groups N or G/N of the two submachines has a non-trivial normal subgroup.

Recall that in the previous section we concluded that a group machine (T, \odot) can monitor a machine (G, \circ) if and only if there exists a normal subgroup N of G such that $T \cong G/N$. Since N is a normal subgroup of G , we can also decompose the original group machine G into an interconnection of a subgroup machine N and a coset leader machine G/N . Therefore, we have arrived at an interesting observation: under this particular decomposition and at a finer level of detail, the monitoring approach corresponds to *partial modular redundancy*, because T is *isomorphic* to the coset leader machine. Error-detection in this special case

¹²The above choice of decomposition is general enough to hold even if N is not a normal subgroup of G . In such case, however, the (right) coset leader machine is no simpler than the original machine; its group is still G , [2].

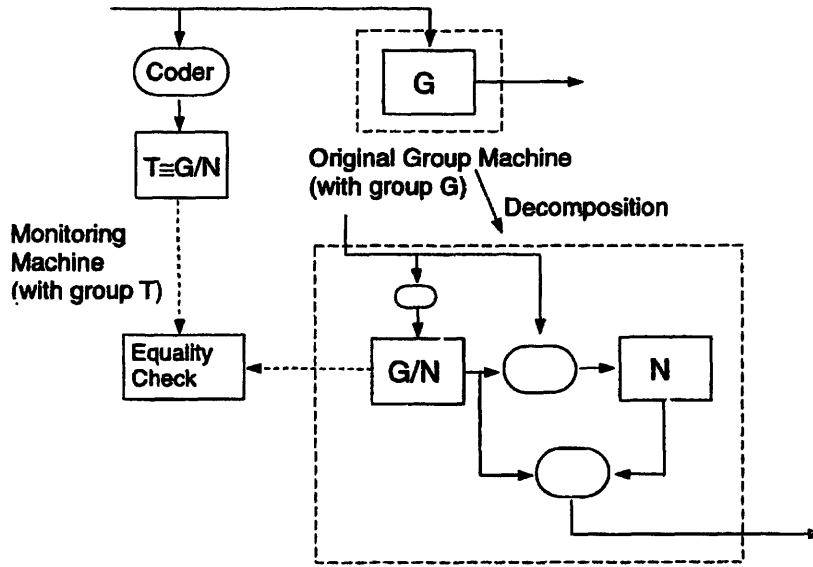


Figure 2-6: Construction of a separate monitor based on group machine decomposition.

is straightforward because, as shown in Figure 2-6, failures in T or G/N can be detected by concurrently comparing their corresponding states. The comparison is a simple equality check (up to isomorphism) and an error is detected whenever there is a disagreement. Failures in the subgroup machine N *cannot* be detected. The error detection and correction capabilities are different, however, if G is implemented using a different decomposition (or not decomposed at all).

Example 2.4 Consider the group machine $Z_4 = \{0, 1, 2, 3\}$ which performs modulo-4 addition (its next-state function is given by the modulo-4 sum of its current state and input). A normal subgroup for Z_4 is given by $N = \{0, 2\}$ ($N \cong Z_2$); the cosets are $\{0, 2\}$ and $\{1, 3\}$, and the resulting coset leader machine $Z_4/N \cong Z_4/Z_2$ is also isomorphic to Z_2 .

Despite the fact that both the coset leader and the subgroup machines have groups isomorphic to Z_2 , the overall functionality is different from $Z_2 \times Z_2$ (since $Z_4 \neq Z_2 \times Z_2$) due to the interconnecting coder between the coset leader and subgroup machines. The output of this coder (denoted in Figure 2-5 by n') is a symbol based on the current state of the coset leader machine (c_{i_1} in Figure 2-5) and the current input (denoted by g_2 in the figure). In this particular example the output functions like the carry-bit in a binary adder: the coset leader machine performs the addition of the least significant bits, whereas the subgroup machine deals with the most significant bits. Since this “carry-bit” is available

concurrently to the subgroup machine (i.e., it depends on the *current* state of the coset leader machine and the *current* input), this decomposition is reminiscent of the use of carry-lookahead to perform modulo- 2^k addition using binary adders (in our case $k = 2$), [110].

Using $N \cong Z_2$ as a normal subgroup of Z_4 , we conclude from the analysis of the previous section that an appropriate separate monitor is given by $T = G/N = Z_4/Z_2 \cong Z_2$. It functions as follows: it encodes the inputs in coset $\{0, 2\}$ into 0_2 and those in $\{1, 3\}$ into 1_2 ; then, it adds its current state to its current input modulo-2. Therefore, the functionality of this separate monitor is identical to the coset leader machine in the decomposition described above. As illustrated in Figure 2-6, under this particular decomposition of Z_4 , the monitor will only be able to detect failures that cause errors in the least significant bit (i.e., errors in the coset leader machine). Errors in the most significant bit (which correspond to errors in the subgroup machine) will remain completely undetected. \square

If one replicated the subgroup machine N (instead of the coset leader machine), the resulting “monitor” $T \cong N$ would *not* correspond to a separate code. The reason is that the subgroup machine N receives input from the coset leader machine G/N through the interconnecting coder (e.g., the “carry-bit” in the example above).

Non-Separate Redundant Implementations of Group Machines

A redundant implementation of a group machine (G, \circ) need not necessarily use a separate monitor. More generally, we can appropriately embed (G, \circ) into a larger, redundant group machine (H, \diamond) that preserves the behavior of G in some *non-separately* encoded form (as in Figure 2-3). At the beginning of Section 2.3.1, we showed that such an embedding can be achieved via an *injective* group homomorphism $\phi : G \mapsto H$, used to encode the inputs and states of machine G into those of machine H . Furthermore, since ϕ is injective, there exists a one-to-one *decoding mapping* $\sigma : G' \mapsto G$ (where $G' = \phi(G)$ was defined earlier as the subset of valid results) that is simply the inverse of mapping ϕ . With these choices, the concurrent simulation condition in eq. (2.2) is satisfied.

In the above analysis the set $G' = \phi(G)$ is a subgroup of H . If, in addition, G' is a *normal* subgroup of H , then it is possible to decompose H into a series-parallel interconnection of a

subgroup machine G' (isomorphic to G) and a coset leader machine H/G' (in the terminology introduced in the beginning of Section 2.3.1, G' plays the role of the normal subgroup N). If we actually *implement* H in this decomposed form, then our fault-tolerant scheme attempts to protect the computation in G by performing an isomorphic computation (in the subgroup machine G') and a coset leader computation H/G' . Failures are detected whenever the overall state of H lies outside G' , that is, whenever the state of the coset leader machine deviates from the identity. Since the coset leader machine does *not* receive any input from the subgroup machine G' , failures in the subgroup machine are not reflected in the state of H/G' ; therefore, failures in G' are completely undetected and the only detectable failures are the ones that force H/G' to a state different than the identity. In effect, the added redundancy is checking for failures within itself rather than for failures in the computation in G' (which is isomorphic to the computation in G) and turns out to be rather useless for error detection or correction. As demonstrated in the following example, we can avoid this problem (while keeping the same encoding, decoding and error correcting procedures) by implementing H using a different decomposition; each such decomposition may offer different fault coverage.

Example 2.5 Suppose that we want to protect addition modulo-3, that is, $G = Z_3 = \{0, 1, 2\}$ and decide to do this by using an αM coding scheme where $\alpha = 2$. Therefore, we multiply by 2 and perform addition modulo 6, that is, $H = Z_6 = \{0, 1, \dots, 5\}$. The subgroup $G' = \{0, 2, 4\}$ is isomorphic to G and results in cosets $\{0, 2, 4\}$ and $\{1, 3, 5\}$. If we choose 0 and 1 as the coset leaders, now denoting them by 0_2 and 1_2 to avoid confusion, the coset leader machine has the following state transition function:

Input	$0_2 = \{0, 2, 4\}$	$1_2 = \{1, 3, 5\}$
State		
0_2	0_2	1_2
1_2	1_2	0_2

The coding *function* between the coset leader machine and the subgroup machine (which has no internal state and provides the input to the subgroup machine based on the current coset and input) is given by the following table:

Input	0	1	2	3	4	5
State						
0 ₂	0	0	2	2	4	4
1 ₂	0	2	2	4	4	0

Note that the input to machine H will always be a multiple of 2. Therefore, as is clear from the table, if we start from the 0₂ coset, we will remain there (at least under fault-free conditions). Moreover, the input to the subgroup machine will be essentially the same as in the non-redundant machine (only the symbols used will be different — {0, 2, 4} instead of {0, 1, 2}).

A failure will be detected whenever the overall state of H does not lie in G' , i.e., whenever the coset leader machine H/G' is in a state *different* from 0₂. Since the coset leader machine does not receive any input from the subgroup machine, a deviation from the 0₂ coset reflects a failure in the coset leader machine. Therefore, the redundancy we have added checks itself and not the original machine.

We get better results if we decompose H in some other way. If we use the normal subgroup $N_H = \{0, 3\}$, the corresponding cosets are {0, 3}, {1, 4} and {2, 5} (we will denote the coset leaders by 0₃, 1₃ and 2₃ respectively). The state transition function of the coset leader machine is given by

Input	0 ₃ = {0, 3}	1 ₃ = {1, 4}	2 ₃ = {2, 5}
State			
0 ₃	0 ₃	1 ₃	2 ₃
1 ₃	1 ₃	2 ₃	0 ₃
2 ₃	2 ₃	0 ₃	1 ₃

In this case, the output of the coding function between the two machines is given by the following table:

Input	0	1	2	3	4	5
State						
0 ₃	0	0	0	3	3	3
1 ₃	0	0	3	3	3	0
2 ₃	0	3	3	3	0	0

This situation is quite different from the one described before. The valid results under fault-free conditions do not lie in the same coset anymore. Instead, for each state in the coset leader machine there is exactly one valid state in the subgroup machine. More specifically, the valid results (the ones that comprise the subgroup G') are given by the following (c, n_h) pairs (where c is a coset leader and n_h is an element of the subgroup machine N_H): $(0_3, 0)$, $(1_3, 3)$ and $(2_3, 0)$. We can exploit this “structured redundancy” to perform error detection and correction.

The analysis in this example can be generalized to all cyclic group machines Z_M that are to be protected through αM coding. The encoding of the states and the inputs involves simple multiplication by α , whereas the computation should be reformulated using a group machine decomposition that does not have Z_M as a (normal) subgroup. (Otherwise, it is not possible to detect/correct errors in the computation of Z_M .) \square

The example above has illustrated that non-separate redundancy can be inefficient (or even useless), depending on the particular group machine decomposition. Research work in the past had focused on a given (fixed) *hardware implementation* of the redundant machine. For example, αM codes were applied to arithmetic circuits with a *specific* architecture in mind and with the objective of choosing the parameter α so that an acceptable level of error detection/correction is achieved, [87, 81]. Similarly, the design of *self-testing* and *fault-secure* networks in [108] is based on requiring that all failures under the given implementation cause transitions to invalid states. Again, the indirect assumption is that the machine implementation and decomposition are fixed. Our approach is different because we characterize the encoding and decoding mappings *abstractly*, and allow the possibility of implementing and decomposing the redundant machine in different ways; each such decomposition will likely result in different fault coverage. Chapters 3 and 4 illustrate the kind of flexibility that we have when constructing these redundant implementations.

2.3.2 Redundant Implementations of Semigroup Machines

The development of eqs. (2.1) and (2.2) in Section 2.3.1 can be generalized to arbitrary semigroup machines. For this case, we have the following definition:

Definition 2.2 A redundant implementation for a semigroup machine (S, \circ) is a semigroup machine (H, \diamond) that concurrently simulates S in the following sense: there exist a one-to-one mapping $\ell : S' \mapsto S$ (where $S' = \ell^{-1}(S) \subset H$) and an appropriate input encoding mapping $\xi : S \mapsto H$ (from S into H) such that the following condition holds true:

$$\ell(\ell^{-1}(s_1) \diamond \xi(s_2)) = s_1 \circ s_2 \quad (2.3)$$

for all $s_1, s_2 \in S$.

Using similar analysis to Section 2.3.1 (i.e., under the assumptions that $\ell^{-1}(1_\circ) = 1_\diamond$ and $\xi(1_\circ) = 1_\diamond$), we conclude that ξ and ℓ^{-1} have to be the same injective semigroup homomorphism, if we use ϕ to denote ξ and ℓ^{-1} , and σ in place of ℓ , we arrive at

$$\sigma(\phi(s_1) \diamond \phi(s_2)) = s_1 \circ s_2 \quad (2.4)$$

for all $s_1, s_2 \in S$, which is the same as the condition for fault-tolerant semigroup computations in Section 2.2.2. We can thus construct redundant implementations for semigroup machines by embedding them into *larger* semigroup machines using homomorphisms.

The decomposition of group machines described in Section 2.3.1 has generalizations to semigroup machines, the most well-known result being the Krohn-Rhodes decomposition theorem, [2, 3]. This theorem states that an arbitrary semigroup machine (S, \circ) can be decomposed in a *non-unique* way into a series-parallel interconnection of simpler components that are either *simple-group*¹³ machines or one of four *basic* types of semigroup machines. These basic machines correspond to the following semigroups (known as *units*):

- $U_3 = \{1, r_1, r_2\}$ such that for $u, r_i \in U_3$, $u \circ 1 = 1 \circ u = u$ and $u \circ r_i = r_i$.
- $U_2 = \{r_1, r_2\}$ such that for $u, r_i \in U_2$, $u \circ r_i = r_i$.
- $U_1 = \{1, r\}$ such that 1 is the identity element and $r \circ r = r$.
- $U_0 = \{1\}$.

¹³A simple group is a group that does not have any non-trivial normal subgroups.

Note that U_0 , U_1 and U_2 are in fact subsemigroups of U_3 . Each simple-group machine in a Krohn-Rhodes decomposition has a simple group that is a homomorphic image of some subgroup of S . It is possible that the decomposition uses multiple copies of a particular simple-group machine or no copy at all. Some further results and ramifications can be found in [40].

A semigroup machine is called a *reset* if it corresponds to a right-zero semigroup R , that is, for all r_i, r_j in R , $r_i \circ r_j = r_j$. A *reset-identity* machine $R^1 = R \cup \{1\}$ corresponds to a right-zero semigroup R with 1 included as the identity. A *permutation-reset* machine has a semigroup (S, \circ) that is the union of a set of right zeros $R = \{r_1, r_2, \dots, r_n\}$ and a group $G = \{g_1, g_2, \dots, g_m\}$. (The product $r_i \circ g_j$ for $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$ is defined to be $r_i \circ g_j = r_k$ for some $k \in \{1, \dots, n\}$. The remaining products are defined so that G forms a group and R is a set of right zeros.) A permutation-reset machine can be decomposed into a series-parallel pair with the group machine G at the front-end and the reset-identity machine $R^1 = R \cup \{1\}$ at the back-end. This construction can be found in [2].

The Zieger decomposition is a special case of the Krohn-Rhodes decomposition. It states that any general semigroup machine S may be broken down into permutation-reset components. All groups involved are homomorphic images of subgroups of S . More details and an outline of the procedure may be found in [2].

Next, we discuss redundant implementations for reset-identity machines. By the Zieger decomposition theorem, such machines together with simple-group machines are the only building blocks needed to construct all possible semigroup machines.

Separate Monitors for Reset-Identity Machines

For a right-zero semigroup R , any equivalence relation (i.e., any partitioning of its elements) is a congruence relation, [42]. This result extends to the monoid $R^1 = R \cup \{1\}$: any partitioning of the elements of R^1 is a congruence relation, as long as the identity forms its own partition. Using this we can characterize and construct all possible (separate) monitors for a given reset-identity machine R^1 .

Example 2.6 Consider the standard semigroup machine U_3 defined in the previous section. Its next-state function is given by the following table:

Input	1	r_1	r_2
State			
1	1	r_1	r_2
r_1	r_1	r_1	r_2
r_2	r_2	r_1	r_2

The only possible non-trivial partitioning is $\{\{1\}, \{r_1, r_2\}\}$; it results in the parity semigroup $T = \{1_T, r\}$, defined by the surjective homomorphism $\pi : U_3 \mapsto T$ with $\pi(1) = 1_T$ and $\pi(r_1) = \pi(r_2) = r$. Note that T is actually isomorphic to U_1 . Under this monitoring scheme, machine T is simply a coarser version of the original machine U_3 (it treats both right zeros, r_1 and r_2 , in the same way). \square

Example 2.7 Consider the reset-identity machine¹⁴ $R_7^1 = \{1_7, r_{1_7}, r_{2_7}, \dots, r_{7_7}\}$. A possible partitioning for it is $\{\{1_7\}, \{r_{1_7}, r_{2_7}, \dots, r_{7_7}\}\}$ and it results in the same parity semigroup $T \cong U_1$ as in the previous example (the mapping $\pi : R_7^1 \mapsto T$ is given by $\pi(1_7) = 1_T$, $\pi(r_{1_7}) = \pi(r_{2_7}) = \dots = \pi(r_{7_7}) = r$).

Other partitionings are also possible as long as the identity forms its own class. This flexibility in the choice of partitioning could be used to exploit to our advantage the error model and the actual failures expected in the implementations of the original machine R_7^1 and/or the monitor T .

For example, if R_7^1 is implemented digitally (each state being coded to three bits), then we could choose our partitions to consist of states whose encodings are separated by large Hamming distances. For example, if the binary encodings for the states of R_7^1 are 000 for the identity, and 001, 010, ..., 111 for r_{1_7} to r_{7_7} respectively, then an appropriate partitioning could be $\{p_0 = \{000\}, p_1 = \{001, 010, 100, 111\}, p_2 = \{011, 101, 110\}\}$. This results in a monitoring machine with semigroup $T \cong U_3$: state 000 maps to the identity, whereas states in partition p_1 map to r_1 and states in partition p_2 map to r_2 . Under this scheme, we will be able to detect failures that cause single-bit errors in the original machine as long as the monitoring machine operates correctly (to see this, notice that the Hamming distance within each of the partitions is larger than 1).

¹⁴In general, R_n^1 will indicate the reset-identity machine with n right zeros (denoted by $\{r_{1_n}, r_{2_n}, \dots, r_{n_n}\}$) and an identity element (denoted by 1_n); thus, $U_3 = R_2^1$.

The scheme above can be made c -error correcting by ensuring that the Hamming distance within any partition is at least $2c + 1$ (still assuming no failures in the monitoring machine). Under more restrictive error models, other partitionings could be more effective. For example, if failures in a given implementation cause bits to stick at “1”, then we should aim for partitions with states separated by a large *asymmetric* distance, [87]. \square

As we have seen through the above examples (and can easily prove), the monitoring machine for R_n^1 is a smaller reset-identity machine R_p^1 with $1 \leq p \leq n$. Moreover, just as in the group case, there exists a particular decomposition of R_n^1 in which the monitor R_p^1 appears as a front-end submachine. In fact, R_n^1 can be realized as a *parallel* decomposition of R_p^1 and R_q^1 as follows: partition the n resets into p classes, each of which has at most q elements (clearly, $q \leq n$ and $q \times p \geq n$); then, under appropriate encoding, the state of machine R_p^1 can be used to specify the partition and the state of machine R_q^1 can specify the exact element within each partition.

Note that in our examples we have assumed for simplicity that the monitoring machine is fault-free. This might be a reasonable assumption if the monitor is a lot simpler than the original machine. In case we want to consider failures in the monitor, the encoding of its states should also enter the picture.

Non-Separate Redundant Implementations for Reset-Identity Machines

Just as in the case of group machines, a redundant implementation of a reset-identity machine R_n^1 can be based on an injective semigroup homomorphism $\phi : R_n^1 \mapsto H$ that reflects the state and input of R_n^1 into a larger semigroup machine H so that eq. (2.4) is satisfied. Under proper initialization and fault-free conditions, machine H simulates the reset-identity machine R_n^1 ; furthermore, since ϕ is injective, there exists a mapping σ that can decode the state of H into the corresponding state of R_n^1 .

An interesting case occurs when the monoid $R_n^1 = \{1_n, r_{1n}, r_{2n}, \dots, r_{nn}\}$ is homomorphically embedded into a larger monoid $R_m^1 = \{1_m, r_{1m}, r_{2m}, \dots, r_{mm}\}$ for $m > n$ (i.e., when $H = R_m^1$). The homomorphism $\phi : R_n^1 \mapsto R_m^1$ is given by $\phi(1_n) = 1_m$ and $\phi(r_{in}) \neq \phi(r_{jn})$ for $i \neq j$, i, j in $\{1, 2, \dots, n\}$. Clearly, ϕ is injective and there is a one-to-one decoding mapping σ from the subsemigroup $R_n^1 = \phi(R_n^1) \subset R_m^1$ onto R_n^1 . Assuming that the system

is implemented digitally (i.e., each state is encoded as a binary vector), then in order to protect against single-bit errors we would need to ensure that the encodings of the states in the set of valid results R_n^1 are separated by large Hamming distances. Bit errors can be detected by checking whether the resulting encoding is in R_n^1 .

Example 2.8 One way to add redundancy into the semigroup machine $R_2^1 = \{1_2, r_{1_2}, r_{2_2}\}$ is by mapping it into machine R_7^1 . Any mapping ϕ of the form $\phi(1_2) = 1_7$, $\phi(r_{1_2}) = r_{i_7}$ and $\phi(r_{2_2}) = r_{j_7}$ ($j, i \in \{1, 2, \dots, 7\}$, $j \neq i$) is a valid embedding. In order to achieve detection of single failures, we need to ensure that each failure will result in a state outside the set of valid results S' .

If machine R_7^1 is implemented digitally (with its states encoded into 3-bit binary vectors), failures that result in single-bit errors can be detected by choosing the encodings for $\phi(1_2) = 1_7$, $\phi(r_{1_2}) = r_{i_7}$ and $\phi(r_{2_2}) = r_{j_7}$ ($j, i \in \{1, 2, \dots, 7\}$) to be separated by a Hamming distance of at least 2 (e.g., 001 for 1_7 , 010 for r_{i_7} and 100 for r_{j_7}). \square

2.4 Redundant Implementations of Finite Semiautomata

Error detection and correction is extremely important in the design of finite-state controllers for critical applications. In this section we use algebraic homomorphisms to develop and analyze separate and non-separate redundant implementations for finite semiautomata.

Given a finite semiautomaton (FS) \mathcal{S} with state set Q , input set X and next-state function δ , we can associate with it a *unique* semigroup (S, \circ) , [2, 3]. The construction of S is as follows: for every input sequence of finite length n , denoted by $x = x_{i_1} x_{i_2} \dots x_{i_n}$ (where $x_{i_j} \in X$ and $n \geq 0$), let $s_x : Q \rightarrow Q$ be the induced¹⁵ state mapping: $s_x(q_0) = q_n$ if starting from state q_0 and sequentially applying inputs x_{i_1} , x_{i_2} , and so on (up to x_{i_n}), FS \mathcal{S} ends up in state q_n . There is a finite number of such mappings and our construction is likely to use only a subset of them. If we let S be the union of these mappings s_x , then (S, \circ) forms a monoid under the composition of mappings given by

$$s_x \circ s_{x'} = s_{xx'},$$

¹⁵The underlying assumption is that at any given state there are defined transitions for *all* possible inputs. This is not necessarily true, but it can be fixed easily by adding an absorbing “dummy” state and using it as the next state for all transitions that were not defined originally.

where xx' denotes the concatenation of the two input sequences x and x' . One can check that \circ is an associative operation (it inherits associativity from the composition of mappings). The identity element of S is the identity mapping s_Λ which corresponds to the empty input sequence Λ (Λ is assumed to have length 0).

The semigroup machine (S, \circ) of a given FS \mathcal{S} has a larger number of states, potentially exponential in the number of states of the original semiautomaton. It captures, however, the *behavior* of the finite semiautomaton by specifying equivalences between different sequences of inputs (which is also the description an engineer is likely to be given before constructing a finite semiautomaton). This representation allows one to explore techniques for algebraic machine decomposition and parallelism. Furthermore, it introduces alternative possibilities for semiautomata implementation and state assignment, and it is matched to our framework for introducing redundancy and eventually fault tolerance.

Example 2.9 A *permutation-reset* finite semiautomaton is one for which every input either produces a permutation of the states or resets the semiautomaton to a constant state, [47]. Here, we consider the permutation-reset FS \mathcal{S} with state set $Q = \{q_1, q_2, q_3\}$, input set $X = \{x_1, x_2\}$ (input x_1 generates a permutation whereas input x_2 is a reset) and the following next-state function:

State	Input	
	x_1	x_2
q_1	q_2	q_1
q_2	q_1	q_1
q_3	q_3	q_1

To generate the semigroup of the finite semiautomaton, we extend this table to sequences of length 0, 1, 2, etc. as follows:

Input Sequence	Λ	x_1	x_2	x_1x_1	x_1x_2	x_2x_1	x_2x_2
State							
q_1	q_1	q_2	q_1	q_1	q_1	q_2	q_1
q_2	q_2	q_1	q_1	q_2	q_1	q_2	q_1
q_3	q_3	q_3	q_1	q_3	q_1	q_2	q_1

We see that the input sequences induce four different state mappings: $a \equiv s_\Lambda =$

$\{\Lambda, x_1x_1\}$, $b \equiv s_{x_1} = \{x_1\}$, $c \equiv s_{x_2} = \{x_2, x_1x_2, x_2x_2\}$, $d \equiv s_{x_2x_1} = \{x_2x_1\}$. It is not hard to convince oneself that longer input sequences will induce mappings that have already been generated by one of these four partitions. For example, $x_2x_1x_2x_1 = x_2(x_1x_2)x_1 \cong x_2(x_2)x_1 = (x_2x_2)x_1 \cong (x_2)x_1 = x_2x_1$ (where \cong denotes input sequence equivalence), and so on. The semigroup S of the permutation-reset FS \mathcal{S} is therefore given by the following table:

"Input"	a	b	c	d
"State"				
a	a	b	c	d
b	b	a	c	d
c	c	d	c	d
d	d	c	c	d

One can check that the table defines an associative operation. The table can also be regarded as the next-state transition table for the semigroup machine S : the "states" and the "inputs" are the same set of elements (given by $\{a, b, c, d\}$). Inputs $b \equiv s_{x_1}$ and $c \equiv s_{x_2}$ are in some sense equivalent to the inputs x_1 and x_2 of the original FS \mathcal{S} . Similarly, if we assume that the starting state is q_3 , states a, b are "equivalent" to state q_3 in \mathcal{S} , state c is "equivalent" to q_1 and state d to q_2 (because, if we start at state q_3 , we end up in state q_3 under input sequence x_1x_1 ($a \equiv s_{x_1x_1}$), in state q_3 under input sequence x_1 ($b \equiv s_{x_1}$), and so forth. Note that the semigroup S consists of the subgroup $\{a, b\}$ and the set of resets $\{c, d\}$; more generally, the semigroup of a permutation-reset finite semiautomaton is the union of a group and a set of resets. \square

The reverse construction is also possible but not unique: given a finite monoid (S, \circ) , one can generate many different finite semiautomata that have S as their monoid. One possibility is to take $Q = S$, $X = S$, and to set $\delta(s_1, s_2) = s_1 \circ s_2$; another one is to take any set of generators for S as the input set X . Therefore, even though each finite semiautomaton has a unique semigroup, there are many finite semiautomata corresponding to a given semigroup.

Suppose that (S, \circ) is the semigroup of a *reduced*¹⁶ FS \mathcal{S} (with state set Q , input set

¹⁶The FS \mathcal{S} is *reduced* if for all pairs of states $q_1, q_2 \in Q$ ($q_1 \neq q_2$) there exists a finite input sequence $x = x_{i_1}x_{i_2}\dots x_{i_n}$ such that $s_x(q_1) \neq s_x(q_2)$, [47].

X and next-state function δ). Then S can also be regarded as a semigroup machine (with state $s_1 \in S$, input $s_2 \in S$ and next-state function $s_1 \circ s_2$). In certain cases that will be of interest for us later on, we will need to restrict the inputs of semigroup machine S to the *subset of available inputs* $I = \{s_{x_1}, s_{x_2}, \dots, s_{x_m}\} \subset S$ (where $m = |X|$ is the number of distinct inputs for FS S). By restricting the inputs to subset I , we obtain an FS S_I (which is in some sense simpler than S) with state $s \in S$, input $s_{x_i} \in I$ ($1 \leq i \leq m$) and next state given by $s \circ s_{x_i} \in S$. Under the condition that the original FS S has a *starting state*, i.e., a state from which all other states are reachable, it can be shown that FS S can be *simulated* by S_I , [47]. Specifically, there exist a surjective mapping $\zeta : S \mapsto Q$ such that for all $s \in S$ and all $s_{x_i} \in I$

$$\zeta(s \circ s_{x_i}) = \delta_S(\zeta(s), x_i) .$$

The following example illustrates this, using FS S in Example 2.9:

Example 2.9 (continued): The set of available inputs for the semigroup machine S in Example 2.9 is given by $I = \{s_{x_1}, s_{x_2}\} \equiv \{b, c\}$; therefore, FS S_I has the following next-state function:

Input	b	c
State		
a	b	c
b	a	c
c	d	c
d	c	c

Machine S_I simulates the original FS S . To see this, define ζ using the following construction: pick the starting state q_3 of S (any starting state would do) and for each state mapping $s_x \in S$, set $\zeta(s_x) = s_x(q_3)$. This results in $\zeta(a) = \zeta(s_A) = q_3$, $\zeta(b) = \zeta(s_{x_1}) = q_3$, $\zeta(c) = \zeta(s_{x_2}) = q_1$ and $\zeta(d) = \zeta(s_{x_2x_2}) = q_2$. \square

2.4.1 Characterization of Non-Separate Redundant Implementations

In this section we construct redundant implementations for finite semiautomata by reflecting the state of the original finite semiautomaton into a larger, redundant semiautomaton that

preserves the properties and state of the original one in some encoded form. The redundancy in the larger semiautomaton can be used for error detection and correction.

The following definition formalizes the notion of a non-separate redundant implementation:

Definition 2.3 *A redundant implementation of an FS \mathcal{S} (with state set $Q_{\mathcal{S}}$, input set $X_{\mathcal{S}}$, initial state $q_{0_{\mathcal{S}}}$ and next-state function $\delta_{\mathcal{S}}$) is an FS \mathcal{H} (with a larger state set $Q_{\mathcal{H}}$, input set $X_{\mathcal{H}}$, initial state $q_{0_{\mathcal{H}}}$ and next-state function $\delta_{\mathcal{H}}$) that concurrently simulates \mathcal{S} in the following sense: there exists a one-to-one mapping $\ell : Q'_{\mathcal{H}} \mapsto Q_{\mathcal{S}}$ (where $Q'_{\mathcal{H}} = \ell^{-1}(Q_{\mathcal{S}}) \subset Q_{\mathcal{H}}$) and an injective mapping $\xi : X_{\mathcal{S}} \mapsto X_{\mathcal{H}}$ (from $X_{\mathcal{S}}$ into $X_{\mathcal{H}}$) such that:*

$$\ell(\delta_{\mathcal{H}}(\ell^{-1}(q_s), \xi(x_s))) = \delta_{\mathcal{S}}(q_s, x_s) \quad (2.5)$$

for all $q_s \in Q_{\mathcal{S}}$, $x_s \in X_{\mathcal{S}}$.

If we initialize \mathcal{H} to state $q_{0_{\mathcal{H}}} = \ell^{-1}(q_{0_{\mathcal{S}}})$ and encode the input x_s using mapping ξ , then the state of \mathcal{S} at all time steps can be recovered from the state of \mathcal{H} through the decoding mapping ℓ , i.e., $q_s = \ell(q_{\mathcal{H}})$; this can be proved by induction. The subset $Q'_{\mathcal{H}}$ can be regarded as the subset of valid states; detecting a state outside $Q'_{\mathcal{H}}$ implies that a failure has taken place. Note that in the special case where a group (semigroup) machine (G, \circ) is to be protected through an embedding into a group (semigroup) machine (H, \diamond) , the next-state functions are given by the group (semigroup) products and the equation above reduces to eqs. (2.1) and (2.3).

At the end of each time step in the redundant FS \mathcal{H} , we perform concurrent error detection/correction to check whether the resulting state belongs to the set of valid states ($Q'_{\mathcal{H}}$). Note that this detection/correction stage is input-independent (i.e., the detector/corrector does not keep track of the input that has been applied). The conditions for single-error detection and correction can be found in Appendix A.

Theorem 2.1 *If FS \mathcal{H} is a redundant implementation of FS \mathcal{S} (as defined in eq. (2.5)), then the semigroup S of \mathcal{S} is isomorphic to a subsemigroup of H (H denotes the semigroup of FS \mathcal{H}).*

Proof: If we let $q'_h = \ell^{-1}(q_s)$ (since ℓ is invertible), eq. (2.5) becomes

$$\ell(\delta_{\mathcal{H}}(q'_h, \xi(x_s))) = \delta_S(\ell(q'_h), x_s) \quad (2.6)$$

for all $q'_h \in Q'_{\mathcal{H}}$, $x_s \in X_S$. The redundant implementation \mathcal{H} as defined by the above equation is a particular instance of a *cover machine* (or *simulator*). A cover machine \mathcal{C} for S is a finite semiautomaton that simulates S as in eq. (2.6), except that the mapping ℓ is only required to be surjective (i.e., the mapping ℓ^{-1} does not necessarily exist and the set $Q'_{\mathcal{H}}$ may have order larger or equal to the order of Q_S). It can be shown that the semigroup S is homomorphic to a subsemigroup of \mathcal{C} (where \mathcal{C} is the semigroup of the cover machine \mathcal{C}), [40]. For the redundant implementation \mathcal{H} , the requirement that ℓ is one-to-one ensures that S will be *isomorphic* to a subsemigroup of H . \square

The preceding discussion shows how one can use the results of Section 2.2.2 to study fault tolerance for an FS \mathcal{S} . We construct the corresponding semigroup S and study injective homomorphisms of the form $\xi : S \mapsto H$ (where H is a *larger* semigroup that includes an isomorphic copy of S as a subsemigroup). Such homomorphisms incorporate redundancy into the implementation of S in a non-separate way.

2.4.2 Characterization of Separate Redundant Implementations

Definition 2.4 An FS \mathcal{T} (with state set $Q_{\mathcal{T}}$, input set $X_{\mathcal{T}}$ and next-state function $\delta_{\mathcal{T}}$) is a *separate monitor* for an FS \mathcal{S} (with state set Q_S , input set X_S and next-state function δ_S) if the following condition is satisfied: for all $q_s \in Q_S$, $x_s \in X_S$

$$\zeta(\delta_S(q_s, x_s)) = \delta_{\mathcal{T}}(\zeta(q_s), \xi(x_s)) , \quad (2.7)$$

where ζ is a surjective mapping from Q_S onto $Q_{\mathcal{T}}$ and ξ is a surjective mapping from X_S onto $X_{\mathcal{T}}$.

Note that if the input mapping ξ is one-to-one, then FS \mathcal{T} being a separate monitor for FS \mathcal{S} implies that FS \mathcal{S} is a cover for \mathcal{T} (the converse is *not* true because a cover may have additional states that are not used when simulating \mathcal{T}).

The above definition for separate monitors (but with ξ being the identity mapping) has appeared in [54, 78, 77]. Monitor \mathcal{T} operates in parallel and separately from \mathcal{S} ; it requires no state information from \mathcal{S} , only (a function ξ of) its input x_s . Condition (2.7) guarantees that, if FS \mathcal{T} is initialized to $q_{0\mathcal{T}} = \zeta(q_{0\mathcal{S}})$ (where $q_{0\mathcal{S}}$ is the initial state of \mathcal{S}), its future state at all time steps will be given by $\zeta(q_s)$ (where q_s is the corresponding state in \mathcal{S}). This can be proved by induction. Therefore, the separate monitor \mathcal{T} can serve as a separate (“parity”) checker for FS \mathcal{S} .

Theorem 2.2 *If FS \mathcal{T} is a separate monitor for an FS \mathcal{S} , then the semigroup T of \mathcal{T} is isomorphic to S/\sim for a congruence relation \sim of semigroup S (the semigroup of FS \mathcal{S}). In other words, there exists a homomorphism $\pi : S \mapsto T$ such that $T = \pi(S)$.*

Proof: It can be shown that if \mathcal{T} is a separate monitor for \mathcal{S} , then its semigroup T is a (surjective) homomorphic image of S , the semigroup of \mathcal{S} , [40]. Combining this with the results of Section 2.2.2 (where we used the fact that T is a surjective homomorphic image of S only if $T \cong S/\sim$ for some appropriate congruence relation \sim), we conclude that \mathcal{T} can be a monitor for \mathcal{S} only if its semigroup T is isomorphic to S/\sim for some congruence relation \sim in S . In particular, when S is a group, T has to be a quotient group of S . \square

The above theorem provides an indirect method of constructing monitoring schemes for semiautomata:

1. given an FS \mathcal{S} , construct the corresponding semigroup machine S ;
2. find $T = S/\sim$ for a congruence relation \sim in S , i.e., find a surjective semigroup homomorphism $\pi : S \mapsto T$ such that $T = \pi(S)$;
3. obtain FS \mathcal{T}_{I_T} (with semigroup T) by restricting the semigroup machine T to the input set $I_T = \pi(I_S)$ (where I_S corresponds to the available inputs X_S in the original FS \mathcal{S}). This forces \mathcal{T}_{I_T} to use inputs that correspond to the original input set X_S in \mathcal{S} ;
4. FS \mathcal{T}_{I_T} can be used to monitor \mathcal{S}_{I_S} (which in turn can simulate the original FS \mathcal{S}).

This procedure is best illustrated by an example, continuing Example 2.9.

Example 2.9 (continued): Let S be the semigroup obtained for FS \mathcal{S} in Example 2.9. One easily checks that $\pi : S \rightarrow U_1 = \{1, r\}$ defined by $\pi(a) = \pi(b) = 1$, $\pi(c) = \pi(d) = r$ is a surjective homomorphism. The subset of available inputs for S , given by $I_S = \{s_{x_1}, s_{x_2}\} = \{b, c\}$, maps to the input set $I_T = \pi(I_S) = \{1, r\}$, which is the subset of available inputs for the separate monitor T . The restriction of the semigroup machine T to inputs I_T gives FS \mathcal{T}_{I_T} which is a separate monitor for FS \mathcal{S}_{I_S} . To see this, notice that:

- The mapping ξ from the inputs of \mathcal{S}_{I_S} to the inputs of \mathcal{T}_{I_T} is given by the restriction of π on the input set I_S . In this particular case, we get: $\xi(b) = \pi(b) = 1$, $\xi(c) = \pi(c) = r$.
- The mapping ζ from the states of \mathcal{S}_{I_S} to the states of \mathcal{T}_{I_T} is given by π , that is, $\zeta(c) = \pi(c) = r$, $\zeta(d) = \pi(d) = r$ and $\zeta(a) = \pi(a) = 1$.

Note that FS \mathcal{S}_{I_S} and monitor \mathcal{T}_{I_T} are not necessarily in a reduced form; if desirable, one can employ state reduction techniques to reduce the number of states, [47]. \square

Using the approach illustrated in the above example, we can design separate monitors \mathcal{T}_{I_T} for FS \mathcal{S}_{I_S} by finding semigroup homomorphisms from S onto T (where S and T are the corresponding semigroups for the two finite semiautomata). Equivalently, as we argued in Section 2.2.2, one can look for congruence relations on semigroup S .

The authors of [54] designed separate monitors for an FS \mathcal{S} using *substitution property (SP) partitions* on its state set Q_S . (A partitioning $\{P_j\}$ of the state set Q_S is an SP partition if, for each partition P_k and each input x_i in the input set X_S , the next states of all states in P_k under input x_i are confined to some partition P_l .) Our approach focuses instead on congruence relations in the semigroup S of the given FS \mathcal{S} . The result is a monitor for FS \mathcal{S}_{I_S} , the restriction of semigroup machine S to the set of available inputs. If we reduce both \mathcal{S}_{I_S} and \mathcal{T}_{I_T} so that they have the minimum number of states [47], we recover the results of [54]. Our approach, however, focuses more on the structure of the finite semiautomata (as summarized by their semigroups) and can take advantage of machine decomposition concepts (resulting in important simplifications in certain special cases, as we have seen with group or reset-identity machines).

An additional advantage of constructing the semigroup of S is that the dynamics of the

finite semiautomata are completely specified. For instance, we can analyze not only the next-state function, but also the n -step next-state function δ_n defined as

$$\delta_n(q_s, x) = \delta(\dots(\delta(\delta(q_s, x_{i_1}), x_{i_2}), \dots), x_{i_n})$$

for all $q_s \in Q_S$ and all length- n sequences $x = x_{i_1}x_{i_2}\dots x_{i_n}$ (with $x_{i_j} \in X_S$). Note that for $n = 1$, $\delta_1 \equiv \delta$, whereas for $n = 0$, δ_0 can be defined as $\delta_0(q_s, \Lambda) = q_s$. By focusing on the n -step next-state function δ_n , we can construct n -step monitors for S . Such monitors have as inputs sequences of length n and compare their state against the state of the original finite semiautomaton once every n time steps.

Example 2.10 The 2-step next-state function for the finite semiautomaton in Example 2.9 is given by the following table:

Input Sequence	x_1x_1	x_1x_2	x_2x_1	x_2x_2
State				
q_1	q_1	q_1	q_2	q_1
q_2	q_2	q_1	q_2	q_1
q_3	q_3	q_1	q_2	q_1

The semigroup generated by this function is given by the subsemigroup of S generated by the elements $\{a, c, d\}$ (which correspond to inputs $\{x_1x_1, x_2x_1, x_1x_2 \equiv x_2x_2\}$ respectively):

"Input"	a	c	d
"State"			
a	a	c	d
c	c	c	d
d	d	c	d

By considering surjective homomorphisms of the above semigroup, one can construct separate monitors that check the operation of FS S_{I_S} once every two inputs. \square

2.5 Summary

In this chapter we considered the problem of systematically constructing redundant implementations for algebraic machines (group/semigroup machines and finite semiautomata).

Our approach was hardware-independent and resulted in appropriate redundant implementations that are based on algebraic embeddings. We did not made any explicit connections with hardware failure modes, but we did address issues regarding machine decomposition. Our techniques take advantage of algebraic structure in order to analyze procedures for error-correction, to avoid decompositions under which failures in the original machine are always undetectable, and to construct separate monitors that perform checking periodically. Future work should focus on making explicit connections with hardware implementations (e.g., digital implementations based on gates and flip-flops) and on exploiting further the role of machine decomposition. It would also be interesting to extend these ideas to other settings (e.g., *group homomorphic* systems, [15]) and to investigate whether fault tolerance can be achieved through a combination of our techniques and the techniques for dynamical systems and codes over finite abelian groups, [34, 16].

Chapter 3

Redundant Implementations of Linear Time-Invariant Dynamic Systems

3.1 Introduction

In this chapter we apply our two-stage approach for fault tolerance to linear time-invariant (LTI) dynamic systems, primarily for the discrete-time case. We focus on error detection and correction that are based on *linear* coding techniques. Specifically, we reflect the state of the original system into a larger LTI dynamic system in a way that preserves the state of the original system in some *linearly* encoded form. At the end of each time step, we use the redundancy in the state representation of the larger system to perform error detection and correction. Our approach results in a complete characterization of this class of redundant implementations for LTI dynamic systems. By adopting a particular hardware implementation that uses delay, adder and gain elements, we demonstrate through examples novel redundant implementations for error detection and correction. Our methodology for mapping to hardware and hardware failure modes is systematic and ensures that single hardware failures result in the corruption of a single state variable. This allows us to employ techniques from linear coding theory to detect and correct failures.

This chapter is organized as follows. In Section 3.2 we provide an introduction to LTI

dynamic systems and in Section 3.3 we characterize the class of redundant implementations that are based on linear coding techniques. In Section 3.4 we map our mathematical equations to explicit hardware constructions by adopting realizations that use delay, adder and gain elements. In Section 3.5 we discuss the implications of our approach by presenting examples of concurrent error detection and correction schemes for LTI dynamic systems.

3.2 Linear Time-Invariant Dynamic Systems

Linear time-invariant (LTI) dynamic systems have a variety of applications in digital filter design, system simulation and model-based control, [92, 58, 67]. Although our discussion is focused on the discrete-time case, most of our results and examples of redundant implementations can be translated to the continuous-time case in a straightforward manner¹. The state evolution of an LTI dynamic system \mathcal{S} is given by

$$\mathbf{q}_s[t+1] = \mathbf{A}\mathbf{q}_s[t] + \mathbf{B}\mathbf{x}[t], \quad (3.1)$$

where t is the discrete-time index, $\mathbf{q}_s[t]$ is the *state vector* and $\mathbf{x}[t]$ is the *input vector*. We assume that $\mathbf{q}_s[\cdot]$ is d -dimensional, $\mathbf{x}[\cdot]$ is u -dimensional and \mathbf{A} and \mathbf{B} are constant matrices of appropriate dimensions (all vectors and matrices have real numbers as entries). Equivalent state-space models (with d -dimensional state vector $\mathbf{q}'_s[t]$) can be obtained through *similarity transformation*, [58, 67]:

$$\begin{aligned} \mathbf{q}'_s[t+1] &= \underbrace{(\mathbf{T}^{-1}\mathbf{A}\mathbf{T})}_{\mathbf{A}'}\mathbf{q}'_s[t] + \underbrace{(\mathbf{T}^{-1}\mathbf{B})}_{\mathbf{B}'}\mathbf{x}[t] \\ &\equiv \mathbf{A}'\mathbf{q}'_s[t] + \mathbf{B}'\mathbf{x}[t], \end{aligned}$$

where \mathbf{T} is an invertible $d \times d$ matrix such that $\mathbf{q}_s[t] = \mathbf{T}\mathbf{q}'_s[t]$. The initial conditions for the transformed system can be obtained as $\mathbf{q}'_s[0] = \mathbf{T}^{-1}\mathbf{q}_s[0]$. Systems related in such a way are known as *similar* systems.

¹Error *detection* in continuous-time LTI dynamic systems can be performed in a way that is similar to the discrete-time case. Error *correction* scenarios, however, will need rethinking.

3.3 Characterization of Redundant Implementations

Our redundant implementations for a given LTI dynamic system will be LTI systems of higher dimensionality. Specifically, a redundant implementation of the LTI dynamic system \mathcal{S} (with state evolution as given in eq. (3.1)) is an LTI dynamic system \mathcal{H} with dimension η ($\eta \equiv d + s$, $s > 0$) and state evolution

$$\mathbf{q}_h[t + 1] = \mathcal{A}\mathbf{q}_h[t] + \mathcal{B}\mathbf{x}[t]. \quad (3.2)$$

We assume that the redundant system \mathcal{H} will be implemented in the same way as the original system \mathcal{S} (Section 3.4 discusses implementations that use delay, adder and gain elements).

We will carefully choose the initial state $\mathbf{q}_h[0]$, and the matrices \mathcal{A} and \mathcal{B} of the redundant system \mathcal{H} , so that under fault-free conditions the states in the redundant system will remain within a subspace $\mathcal{V} \subset \mathbb{R}^\eta$. This subspace will contain all valid states in \mathcal{H} (i.e., states that are obtainable in \mathcal{H} under fault-free conditions) and, as we will show, it will be invariant² under the matrix \mathcal{A} of eq. (3.2). Furthermore, we will require that there exists an appropriate decoding mapping ℓ such that during *fault-free* operation

$$\mathbf{q}_s[t] = \ell(\mathbf{q}_h[t]) \text{ for } \mathbf{q}_h[t] \in \mathcal{V} \text{ and for all } t \geq 0.$$

Note that ℓ is only defined from the subset of *valid* states \mathcal{V} and is required to be *one-to-one* (bijection). This means that the mapping ℓ^{-1} is well-defined and each valid state $\mathbf{q}_h[t] \in \mathcal{V}$ of the redundant system at any time step t corresponds to a unique state $\mathbf{q}_s[t]$ of system \mathcal{S} . In other words, there exists an *encoding* mapping $g \equiv \ell^{-1}$ such that $\mathbf{q}_h[t] = g(\mathbf{q}_s[t])$. We will restrict ourselves to *linear* decoding and encoding techniques. More specifically, we will assume that there exist

- a $d \times \eta$ *decoding* matrix \mathbf{L} such that $\mathbf{q}_s[t] = \mathbf{L}\mathbf{q}_h[t]$ for all t , $\mathbf{q}_h[\cdot] \in \mathcal{V}$,
- an $\eta \times d$ *encoding* matrix \mathbf{G} such that $\mathbf{q}_h[t] = \mathbf{G}\mathbf{q}_s[t]$ for all t .

Under the above assumptions, $\mathbf{L}\mathbf{G} = \mathbf{I}_d$ (where \mathbf{I}_d is the $d \times d$ identity matrix). Note that

² \mathcal{V} is an invariant subspace of matrix \mathcal{A} if $\mathcal{A}\mathbf{v} \in \mathcal{V}$ for all vectors $\mathbf{v} \in \mathcal{V}$, [113].

this equation by itself does not uniquely determine \mathbf{L} given \mathbf{G} , or vice-versa. (In fact, as we will see, even by specifying *both* \mathbf{L} and \mathbf{G} we do not uniquely specify the corresponding redundant system \mathcal{H} .)

The error detector/corrector has to make a decision at the end of each time step based solely on the state $\mathbf{q}_h[t]$ of the redundant system (it does not have access to previous inputs). Since our construction of \mathcal{H} and our choice of initial condition has ensured that under fault-free conditions

$$\mathbf{q}_h[t] = \mathbf{G}\mathbf{q}_s[t] ,$$

our error-detection strategy simply verifies that the redundant state vector $\mathbf{q}_h[t]$ is in the column space of \mathbf{G} . Equivalently, we can check that $\mathbf{q}_h[t]$ is in the null space of an appropriate *parity check matrix* \mathbf{P} , so that $\mathbf{P}\mathbf{q}_h[t] = \mathbf{0}$ under fault-free conditions. All failures that force the state $\mathbf{q}_h[t]$ to fall outside the column space of \mathbf{G} (producing a non-zero parity check $\mathbf{p}[t] \equiv \mathbf{P}\mathbf{q}_h[t]$) will be detected.

For example, a corruption of the i th state variable at time step t (e.g., due to a failure in an adder or in a gain element — see Section 3.4) will produce a state vector $\mathbf{q}_f[t]$, given by

$$\mathbf{q}_f[t] = \mathbf{q}_h[t] + \mathbf{e}_i ,$$

where $\mathbf{q}_h[t]$ is the state vector that would have been obtained under fault-free conditions and \mathbf{e}_i is a column vector with a unique non-zero entry at the i th position. The parity check will then be

$$\begin{aligned} \mathbf{p}[t] \equiv \mathbf{P}\mathbf{q}_f[t] &= \mathbf{P}(\mathbf{q}_h[t] + \mathbf{e}_i) \\ &= \mathbf{P}\mathbf{q}_h[t] + \mathbf{P}\mathbf{e}_i \\ &= \mathbf{P}\mathbf{e}_i . \end{aligned}$$

Single-error correction will be possible if the columns of \mathbf{P} are not multiples of each other. By identifying the column of \mathbf{P} that is a multiple of $\mathbf{p}[t]$, we can locate the corrupted state

variable and correct it³.

We are now in position to state the following theorem:

Theorem 3.1 *In the setting described above, the system \mathcal{H} (of dimension $\eta \equiv d + s$, $s > 0$) and state evolution as in eq. (3.2) is a redundant implementation of S if and only if it is similar to a standard redundant system \mathcal{H}_σ whose state evolution equation is given by*

$$\mathbf{q}_\sigma[t+1] = \begin{bmatrix} \mathbf{A} & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{A}_{22} \end{bmatrix} \mathbf{q}_\sigma[t] + \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix} \mathbf{x}[t]. \quad (3.3)$$

Here, \mathbf{A} and \mathbf{B} are the matrices in eq. (3.1), \mathbf{A}_{22} is an $s \times s$ matrix that describes the dynamics of the redundant modes that have been added, and \mathbf{A}_{12} is a $d \times s$ matrix that describes the coupling from the redundant to the non-redundant modes. Associated with this standard redundant system is the standard decoding matrix $\mathbf{L}_\sigma = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \end{bmatrix}$, the standard encoding matrix $\mathbf{G}_\sigma = \begin{bmatrix} \mathbf{I}_d \\ \mathbf{0} \end{bmatrix}$ and the standard parity check matrix $\mathbf{P}_\sigma = \begin{bmatrix} \mathbf{0} & \mathbf{I}_s \end{bmatrix}$.

Proof: Let \mathcal{H} be a redundant implementation of S . From $\mathbf{L}\mathbf{G} = \mathbf{I}_d$, we know that \mathbf{L} is full-row rank and \mathbf{G} is full-column rank. Furthermore, there exists an invertible $\eta \times \eta$ matrix \mathcal{T} such that $\mathbf{L}\mathcal{T} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \end{bmatrix}$ and $\mathcal{T}^{-1}\mathbf{G} = \begin{bmatrix} \mathbf{I}_d \\ \mathbf{0} \end{bmatrix}$. If we apply the transformation $\mathbf{q}_h[t] = \mathcal{T}\mathbf{q}'_h[t]$ to system \mathcal{H} , we obtain a similar system \mathcal{H}' with decoding mapping $\mathbf{L}' = \mathbf{L}\mathcal{T} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \end{bmatrix}$ and encoding mapping $\mathbf{G}' = \mathcal{T}^{-1}\mathbf{G} = \begin{bmatrix} \mathbf{I}_d \\ \mathbf{0} \end{bmatrix}$. The state evolution of the redundant system \mathcal{H}' is given by

$$\begin{aligned} \mathbf{q}'_h[t+1] &= (\mathcal{T}^{-1}\mathcal{A}\mathcal{T})\mathbf{q}'_h[t] + (\mathcal{T}^{-1}\mathcal{B})\mathbf{x}[t] \\ &\equiv \mathcal{A}'\mathbf{q}'_h[t] + \mathcal{B}'\mathbf{x}[t]. \end{aligned} \quad (3.4)$$

³As mentioned in Chapter 1, one of the main assumptions in most concurrent error detection and correction schemes is that the detecting/correcting mechanism is fault-free. Here, we assume that the evaluation of $\mathbf{P}\mathbf{q}_h[t]$ and all actions that may be subsequently required for error-correction are fault-free. This is a reasonable assumption if the complexity of evaluating $\mathbf{P}\mathbf{q}_h[t]$ is considerably less than the complexity of evaluating $\mathcal{A}\mathbf{q}_h[t] + \mathcal{B}\mathbf{x}[t]$. This would be the case, for example, if the size of \mathbf{P} is much smaller than the size of \mathcal{A} , or if \mathbf{P} requires simpler operations (e.g., if \mathbf{P} only has integer entries whereas \mathcal{A} has real-number entries). In Chapter 6 we address the issue of failures in the error detecting/correcting mechanism.

For all time steps t and under fault-free conditions, $\mathbf{q}'_h[t] = \mathbf{G}'\mathbf{q}_s[t] = \begin{bmatrix} \mathbf{q}_s[t] \\ \mathbf{0} \end{bmatrix}$. Combining the state evolution equations of the original and redundant systems (eqs. (3.1) and (3.4) respectively), we see that

$$\begin{bmatrix} \mathbf{A}\mathbf{q}_s[t] + \mathbf{B}\mathbf{x}[t] \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathcal{A}'_{11} & \mathcal{A}'_{12} \\ \mathcal{A}'_{21} & \mathcal{A}'_{22} \end{bmatrix} \begin{bmatrix} \mathbf{q}_s[t] \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathcal{B}'_1 \\ \mathcal{B}'_2 \end{bmatrix} \mathbf{x}[t].$$

By setting the input $\mathbf{x}[t] \equiv \mathbf{0}$ for all t , we conclude that $\mathcal{A}'_{11} = \mathbf{A}$ and $\mathcal{A}'_{21} = \mathbf{0}$. With the input now allowed to be non-zero, we deduce that $\mathcal{B}'_1 = \mathbf{B}$ and $\mathcal{B}'_2 = \mathbf{0}$. The system \mathcal{H}' is therefore in the form of the standard system \mathcal{H}_σ in eq. (3.3) with appropriate decoding, encoding and parity check⁴ matrices.

The converse, namely that if \mathcal{H} is similar to a standard \mathcal{H}_σ as in eq. (3.3) then it is a redundant implementation of the system in eq. (3.1), is easy to show. \square

Theorem 3.1 establishes a complete characterization of all possible redundant implementations for our fault-tolerant designs of a given LTI dynamic system (subject to the restriction that we use linear encoding and decoding techniques). The additional modes introduced by the redundancy never get excited under fault-free conditions because they are initialized to zero and because they are unreachable from the input. Due to the existence of the coupling matrix \mathbf{A}_{12} , the additional modes are not necessarily unobservable through the decoding matrix. Theorem 3.1 (but stated for the continuous-time case) essentially appears in [52], although the proof is different and the motivation apparently very different. As we will see in the next sections, the application of this theorem in the context of fault tolerance allows us to choose the redundant dynamics and coupling to our advantage.

3.4 Hardware Implementation and Error Model

In order to demonstrate the implications of Theorem 3.1 to our redundant implementations and to our error detection and correction procedures, we need to discuss in more detail

⁴The check matrix can be $\mathbf{P}' = \begin{bmatrix} \mathbf{0} & \Theta \end{bmatrix}$, where Θ is *any* invertible $s \times s$ matrix; a trivial similarity transformation will ensure that the parity check matrix takes the form $\begin{bmatrix} \mathbf{0} & \mathbf{I}_s \end{bmatrix}$, while keeping the system in the standard form \mathcal{H}_σ in eq. (3.3) — except with $\mathbf{A}_{12} = \mathcal{A}'_{12}\Theta$ and $\mathbf{A}_{22} = \Theta^{-1}\mathcal{A}'_{22}\Theta$.

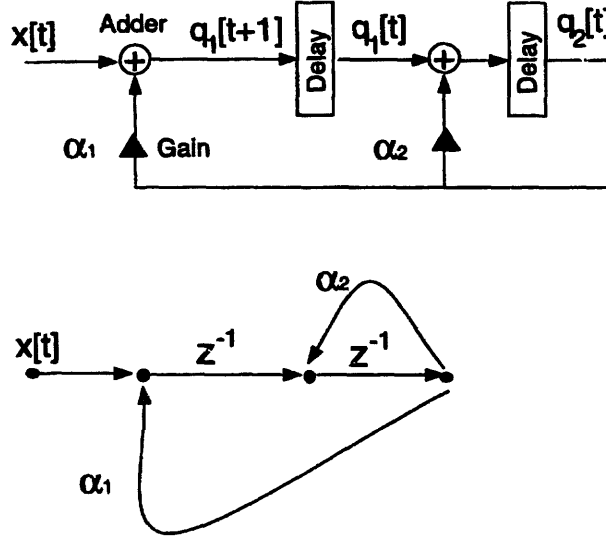


Figure 3-1: Delay-adder-gain circuit and the corresponding signal flow graph.

how our systems are implemented and what kinds of failures we expect. We assume that an LTI dynamic system \mathcal{S} (like the one in eq. (3.1)) is implemented using appropriately interconnected delays (memory elements), adders and gain elements (multipliers). These realizations can be represented by signal flow graphs or, equivalently, by delay-adder-gain diagrams as shown in Figure 3-1 for an LTI dynamic system with state evolution

$$\mathbf{q}[t+1] \equiv \begin{bmatrix} q_1[t+1] \\ q_2[t+1] \end{bmatrix} = \begin{bmatrix} 0 & \alpha_1 \\ 1 & \alpha_2 \end{bmatrix} \mathbf{q}[t] + \begin{bmatrix} 1 \\ 0 \end{bmatrix} x[t].$$

Nodes in a signal flow graph sum up all of their incoming arcs; delays are represented by arcs labeled z^{-1} .

A given state evolution equation has multiple possible realizations using delay, adder and gain elements, [92]. In order to define a *unique* mapping from a state evolution equation to a hardware implementation, we assume⁵ that our implementations correspond to signal flow graphs whose delay-free paths are all of length 1. The signal flow graph in Figure 3-1 is one such example. One easily verifies that for such implementations the entries of the matrices in the state evolution equation are directly reflected as gain constants in the

⁵One can study more general descriptions by studying *factored state variable* descriptions [92], or by employing the computation trees in [22], or using the techniques of Example 3.5.

signal flow graph, [92]. Furthermore, each of the variables in the next-state vector $\mathbf{q}[t + 1]$ is calculated using *separate* gain and adder elements (sharing only the input $\mathbf{x}[t]$ and the previous state vector $\mathbf{q}[t]$). This means that a failure in a single gain element or in a single adder will initially result in the corruption of a *single* state variable; in fact, any combination of failures in the gains or adders that are used in the calculation of the next value of the i th state variable will only corrupt the i th state variable. Note that for arbitrary realizations single failures may corrupt multiple state variables. Under our assumptions, however, there are no shared gain elements or adders; this guarantees that single failures will result in the corruption of a single variable in the state vector.

We consider both *transient* (soft) and *permanent* (hard) failures in the gains and adders of our implementations. A transient failure at time step t causes errors at that particular time step but disappears at the following ones. Therefore, if the errors are corrected before the initiation of time step $t + 1$, the system will resume its normal mode of operation. A permanent failure, on the other hand, causes errors at all remaining time steps. Clearly, a permanent failure can be treated as a transient failure for *each* of the remaining time steps (assuming successful error-correction at every time step), but in certain cases one can deal with it in more efficient ways (e.g., reconfiguration).

Note that we cannot use the standard redundant system \mathcal{H}_σ of Theorem 3.1 to provide fault tolerance to system S . Since we assume that our implementation will employ delay-adder-gain circuits that have delay-free paths of length 1, the implementation of \mathcal{H}_σ will result in a system that only identifies failures in the redundant part of the system. Since the state variables in the lower part of $\mathbf{q}_\sigma[\cdot]$ are not influenced by the variables in the upper part and since our parity check matrix is given by $\mathbf{P}_\sigma = \begin{bmatrix} \mathbf{0} & \mathbf{I}_s \end{bmatrix}$, our fault-tolerant implementation *cannot* identify failures in the original system. The situation is similar to the one in Example 2.5 in Chapter 2. Our objective is to use the redundancy to protect the original system, not to protect the redundancy itself.

Theorem 3.1 is important, however, because it says that we only need to search among systems and implementations that are similar to the standard redundant system \mathcal{H}_σ . Specifically, given an LTI dynamic system S (as in eq. (3.1)), Theorem 3.1 characterizes all possible redundant implementations that have the given (fixed) encoding, decoding and parity check matrices (\mathbf{L} , \mathbf{G} and \mathbf{P} respectively). Since the choice of matrices \mathbf{A}_{12} and \mathbf{A}_{22} is completely

free, there is an infinite number of redundant implementations for system \mathcal{S} . All of these systems have the same encoding, decoding and parity check matrices, and offer the same fault coverage: depending on the redundancy in the parity check matrix \mathbf{P} , all of these implementations can detect and/or correct the same number of errors in the state vector $\mathbf{q}_h[t]$.

Previous suggestions for concurrent error-detection in LTI dynamic systems used linear and real-number codes (for example, in [50, 22]), but overlooked some of the available freedom, essentially setting \mathbf{A}_{12} and \mathbf{A}_{22} to zero. As we will see in the next section, there are a number of interesting redundant implementations with non-zero \mathbf{A}_{12} and \mathbf{A}_{22} ; in fact, in Chapter 6 we see that non-zero \mathbf{A}_{12} and \mathbf{A}_{22} are essential in handling failures in the error detecting/correcting mechanism.

3.5 Examples of Fault-Tolerant Schemes

In this section we discuss the implications of Theorem 3.1 for redundant implementations of LTI dynamic systems. Example 3.1 illustrates how linear coding schemes can be devised using the standard redundant implementation as a starting point. Example 3.2 shows that non-zero \mathbf{A}_{12} and/or \mathbf{A}_{22} can lead to designs that make better use of redundancy and Example 3.3 shows that triple modular redundancy is a special case of our setup. Example 3.4 uses a non-zero \mathbf{A}_{22} to design parity checks with “memory.” Finally, Example 3.5 discusses a reconfiguration scheme that allows the decoding matrix \mathbf{L} to adjust once permanent failures are detected in a gain element or in an adder.

Example 3.1 Using our framework we can easily develop schemes that provide detection and correction of transient failures. The following is a simple motivating example to illustrate the idea. Let the original system \mathcal{S} be

$$\mathbf{q}_s[t+1] = \begin{bmatrix} .2 & 0 & 0 & 0 \\ 0 & .5 & 0 & 0 \\ 0 & 0 & .1 & 0 \\ 0 & 0 & 0 & .6 \end{bmatrix} \mathbf{q}_s[t] + \begin{bmatrix} 3 \\ -1 \\ 7 \\ 0 \end{bmatrix} \mathbf{x}[t].$$

To protect this system against a single transient failure in a gain element or in an adder, we will use three additional modes. More specifically, the standard redundant system will be as follows:

$$\mathbf{q}_\sigma[t+1] = \begin{bmatrix} .2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & .5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & .6 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & .2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & .5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & .3 \end{bmatrix} \mathbf{q}_\sigma[t] + \begin{bmatrix} 3 \\ -1 \\ 7 \\ 0 \\ \hline 0 \\ 0 \\ 0 \end{bmatrix} \mathbf{x}[t].$$

with parity check matrix given by

$$\mathbf{P}_\sigma = \begin{bmatrix} \mathbf{0} & \mathbf{I}_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

For error-detection, we need to check whether $\mathbf{P}_\sigma \mathbf{q}_\sigma[t]$ is $\mathbf{0}$. However, as we argued earlier, redundant systems in standard form cannot be used for detecting failures that cause errors in the *original* modes: given a faulty state vector $\mathbf{q}_f[t]$, a non-zero parity check ($\mathbf{P}_\sigma \mathbf{q}_f[t] \neq \mathbf{0}$) would simply mean that a failure has resulted in an error in the calculation of the *redundant* modes. What we really want is to protect against errors that appear in the *original* modes. One way to achieve this is to employ a system similar to the standard redundant system, but with parity check matrix

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}. \quad (3.5)$$

This choice of \mathbf{P} is motivated by the structure of Hamming codes in communications, [81, 111, 11]. With a suitable similarity transformation \mathcal{T} (so that $\mathbf{P}\mathcal{T} = \mathbf{P}_\sigma$), the corresponding

redundant system is

$$\mathbf{q}_h[t+1] = \begin{bmatrix} .2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & .5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & .6 & 0 & 0 & 0 \\ 0 & -.3 & .1 & 0 & .2 & 0 & 0 \\ .3 & 0 & 0 & -.1 & 0 & .5 & 0 \\ .1 & 0 & .2 & -.3 & 0 & 0 & .3 \end{bmatrix} \mathbf{q}_h[t] + \begin{bmatrix} 3 \\ -1 \\ 7 \\ 0 \\ -9 \\ -2 \\ -10 \end{bmatrix} \mathbf{x}[t]. \quad (3.6)$$

This system can detect and locate transient failures that cause the value of a single state variable to be incorrect at a particular time step. To do this, we check for non-zero entries in the parity vector $\mathbf{p}[t] \equiv \mathbf{P}\mathbf{q}_h[t]$. If, for example, $p_1[t] \neq 0$, $p_2[t] \neq 0$ and $p_3[t] \neq 0$, then $q_1[t]$, the value of the first state variable in $\mathbf{q}_h[t]$ is corrupted; if $p_1[t] \neq 0$, $p_2[t] \neq 0$ and $p_3[t] = 0$, then a failure has corrupted $q_2[t]$; and so forth. Once the erroneous variable is located, we can correct it using any of the parity equations in which it appears. For example, if $q_2[t]$ is corrupted, we can calculate the correct value by setting $q_2[t] = -q_1[t] - q_3[t] - q_5[t]$ (i.e., using the first parity equation). If failures are transient, the operation of the system will resume normally in the following time steps.

Hamming codes, like the ones we used in this example, are able to perform correction of a single error in the state vector. In other words, they can detect and identify a single failure in an adder or in a multiplier. Instead of replicating the whole system, we only need to add a few redundant modes; as long as $2^s - 1 > \eta$ (where $\eta \equiv d + s$ is the dimension of the redundant system), we can guarantee the existence of a Hamming code and a redundant implementation that achieve single-error correction. Note that the authors in [22] developed a *real coding* scheme (also included in our framework) which performs single-error correction using only two additional state variables. It requires, however, more complicated error detection and correction mechanisms. The methods used in [22] (as well as in [21], where one of the authors of [22] analyzes the continuous-time case) do not consider different similarity transformations and do not permit additional modes to be non-zero. Our framework is more general and can take advantage of additional non-zero modes to reduce

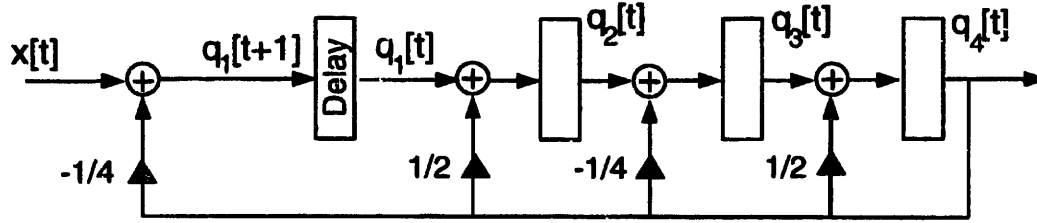


Figure 3-2: Digital filter implementation using delays, adders and gains.

redundant hardware, to devise stable fault-tolerant schemes for continuous-time systems⁶, or, as we will see in Example 3.4, to construct schemes in which checking can be done non-concurrently (e.g., periodically). \square

Example 3.2 Suppose that the original system \mathcal{S} that we would like to protect is the digital filter shown in Figure 3-2. It is implemented using delays, adders and gains interconnected as shown in the figure (note that the corresponding signal flow graph has delay-free paths of length 1). If we let the contents of the delays be the state $\mathbf{q}_s[t]$ of system \mathcal{S} , then its state evolution is given by

$$\begin{aligned} \mathbf{q}_s[t+1] &= \mathbf{A}\mathbf{q}_s[t] + \mathbf{b}x[t] \\ &= \begin{bmatrix} 0 & 0 & 0 & -1/4 \\ 1 & 0 & 0 & 1/2 \\ 0 & 1 & 0 & -1/4 \\ 0 & 0 & 1 & 1/2 \end{bmatrix} \mathbf{q}_s[t] + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} x[t]. \end{aligned}$$

As expected, the entries of \mathbf{A} and \mathbf{b} are reflected directly as gains in the diagram (simple connections have unity gain); furthermore, the only variables that are shared when calculating the next-state vector are the previous state vector and the input; no hardware is shared during the state update.

⁶In [21], they used “negative feedback” or “lossy integrators” to deal with the stability issue. Our use of non-zero redundant modes avoids this problem completely.

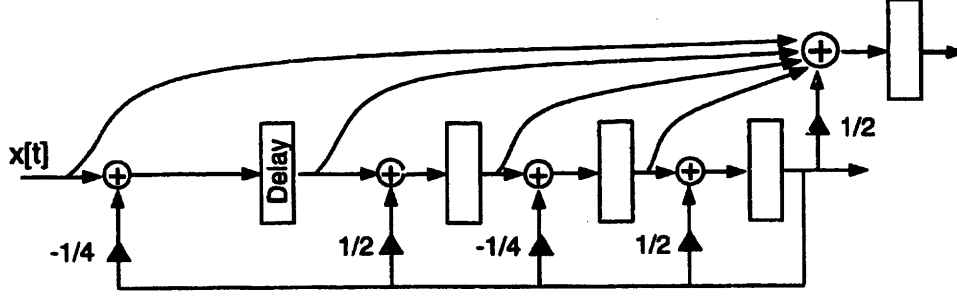


Figure 3-3: Redundant implementation based on a checksum condition.

In order to detect a single failure in a gain element or in an adder, we can use an extra “checksum” state variable as shown in Figure 3-3, [50, 22]. The resulting redundant implementation \mathcal{H} has state evolution

$$\begin{aligned} \mathbf{q}_h[t+1] &= \left[\begin{array}{ccc|c} \mathbf{A} & 0 \\ \mathbf{c}^T \mathbf{A} & 0 \end{array} \right] \mathbf{q}_h[t] + \left[\begin{array}{c} \mathbf{b} \\ \mathbf{c}^T \mathbf{b} \end{array} \right] x[t] \\ &= \left[\begin{array}{cccc|c} 0 & 0 & 0 & -1/4 & 0 \\ 1 & 0 & 0 & 1/2 & 0 \\ 0 & 1 & 0 & -1/4 & 0 \\ 0 & 0 & 1 & 1/2 & 0 \\ \hline 1 & 1 & 1 & 1/2 & 0 \end{array} \right] \mathbf{q}_h[t] + \left[\begin{array}{c} 1 \\ 0 \\ 0 \\ 0 \\ \hline 1 \end{array} \right] x[t], \end{aligned}$$

where $\mathbf{c}^T = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$. There are a number of different delay-adder-gain diagrams that are consistent with the above state evolution equation; the one shown in Figure 3-3 is the only one consistent with our requirement that signal flow graphs have delay paths of length 1.

Under fault-free conditions, the additional state variable is always the sum of all other state variables (which are the original state variables in system \mathcal{S}). Error-detection is based on verifying the validity of this very simple checksum condition; no complicated multiplications are involved, which may make it reasonable to assume that error-detection is fault-free.

The above approach is easily seen to be consistent with our setup. Specifically, the encoding, decoding and parity check matrices are given by:

$$\mathbf{G} = \begin{bmatrix} \mathbf{I}_4 \\ \mathbf{c}^T \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{L} = \left[\mathbf{I}_4 \mid \mathbf{0} \right] = \begin{bmatrix} 1 & 0 & 0 & 0 & \big| & 0 \\ 0 & 1 & 0 & 0 & \big| & 0 \\ 0 & 0 & 1 & 0 & \big| & 0 \\ 0 & 0 & 0 & 1 & \big| & 0 \end{bmatrix},$$

$$\mathbf{P} = \left[-\mathbf{c}^T \mid 1 \right] = \left[-1 \quad -1 \quad -1 \quad -1 \mid 1 \right].$$

Using the transformation matrix $\begin{bmatrix} \mathbf{I}_4 & \mathbf{0} \\ \mathbf{c}^T & 1 \end{bmatrix}$, we can show that system \mathcal{H} is similar to a standard system \mathcal{H}_σ with state evolution

$$\mathbf{q}_\sigma[t+1] = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{q}_\sigma[t] + \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix} x[t],$$

where \mathbf{A} , \mathbf{b} are the ones in eq. (3.1).

Indirectly, the constructions in [50, 22] were choosing \mathbf{A}_{12} and \mathbf{A}_{22} to be zero. As stated earlier, with each choice of \mathbf{A}_{12} and \mathbf{A}_{22} , there is a different redundant implementation with the same encoding, decoding and parity check matrices. If, for example, we set $\mathbf{A}_{12} = \mathbf{0}$ and $\mathbf{A}_{22} = [1]$, and then transform back, we get a redundant implementation \mathcal{H}' whose state evolution equation is given by

$$\begin{aligned} \mathbf{q}_{h'}[t+1] &= \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{c}^T \mathbf{A} - \mathbf{A}_{22} \mathbf{c}^T & \mathbf{A}_{22} \end{bmatrix} \mathbf{q}_{h'}[t] + \begin{bmatrix} \mathbf{b} \\ \mathbf{c}^T \mathbf{b} \end{bmatrix} x[t] \\ &= \begin{bmatrix} 0 & 0 & 0 & -1/4 & 0 \\ 1 & 0 & 0 & 1/2 & 0 \\ 0 & 1 & 0 & -1/4 & 0 \\ 0 & 0 & 1 & 1/2 & 0 \\ 0 & 0 & 0 & -1/2 & 1 \end{bmatrix} \mathbf{q}_{h'}[t] + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} x[t]. \end{aligned}$$

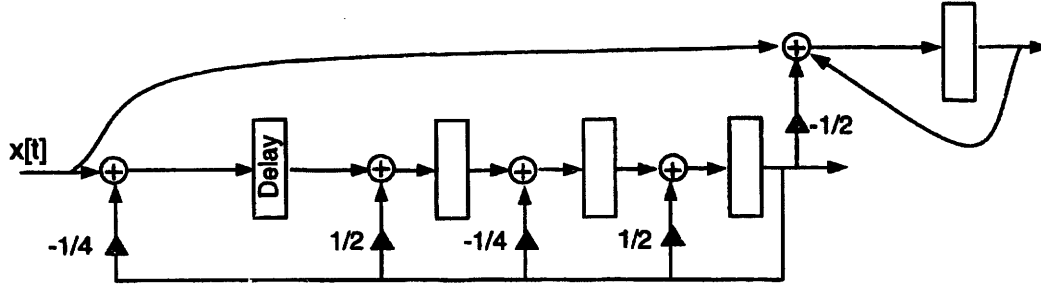


Figure 3-4: A second redundant implementation based on a checksum condition.

This redundant implementation is shown in Figure 3-4.

Both redundant implementations \mathcal{H} and \mathcal{H}' have the same encoding, decoding and parity check matrices and are able to detect failures that corrupt a single state variable (e.g., a single failure in an adder or in a gain element). The (added) complexity in system \mathcal{H}' , however, is lower than that in system \mathcal{H} (the computation of the redundant state variable is less involved). More generally, as illustrated in this example for the case of a non-zero A_{22} , we can explore different versions of these redundant implementations by exploiting the dynamics of the redundant modes (A_{22}) and/or their coupling with the original system (A_{12}). For certain choices of A_{12} and A_{22} , we may get designs that utilize less hardware than others. \square

Example 3.3 Triple modular redundancy (TMR) consists of three copies of the original system, each initialized at the same state and subject to the same input. By comparing their state vectors at the end of each time step, we can detect and correct any failure that appears in one of the subsystems. The correction is performed via a simple voting scheme that selects the state agreed upon by two or more systems. Assuming that the voting scheme can be performed reliably, a TMR system can tolerate any combination of failures that corrupts a single copy of the system.

TMR in our setting corresponds to a redundant system of the form

$$\mathbf{q}_h[t+1] \equiv \begin{bmatrix} \mathbf{q}_s^1[t+1] \\ \mathbf{q}_s^2[t+1] \\ \mathbf{q}_s^3[t+1] \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A} \end{bmatrix} \mathbf{q}_h[t] + \begin{bmatrix} \mathbf{B} \\ \mathbf{B} \\ \mathbf{B} \end{bmatrix} \mathbf{x}[t],$$

where $\mathbf{q}_s^1[t]$, $\mathbf{q}_s^2[t]$ and $\mathbf{q}_s^3[t]$ evolve in the same way (because $\mathbf{q}_s^1[0] = \mathbf{q}_s^2[0] = \mathbf{q}_s^3[0] = \mathbf{q}_s[0]$). Each of $\mathbf{q}_s^1[\cdot]$, $\mathbf{q}_s^2[\cdot]$ and $\mathbf{q}_s^3[\cdot]$, however, is calculated using a separate set of delays, adders and gains (according to our assumptions about mapping to hardware as outlined in Section 3.4).

The encoding matrix \mathbf{G} is given by $\begin{bmatrix} \mathbf{I}_d \\ \mathbf{I}_d \\ \mathbf{I}_d \end{bmatrix}$, the decoding mapping \mathbf{L} can be⁷ $\begin{bmatrix} \mathbf{I}_d & \mathbf{0} & \mathbf{0} \end{bmatrix}$ and the parity check matrix \mathbf{P} can be $\begin{bmatrix} -\mathbf{I}_d & \mathbf{I}_d & \mathbf{0} \\ -\mathbf{I}_d & \mathbf{0} & \mathbf{I}_d \end{bmatrix}$. A non-zero entry in the upper (respectively lower) half of $\mathbf{P}\mathbf{q}_h[t]$ indicates a failure in subsystem 2 (respectively 3). Non-zero entries in both the top and bottom half-vectors, indicate a failure in subsystem 1.

The TMR system is easily shown (for example, with transformation matrix $\begin{bmatrix} \mathbf{I}_d & \mathbf{0} & \mathbf{0} \\ \mathbf{I}_d & \mathbf{I}_d & \mathbf{0} \\ \mathbf{I}_d & \mathbf{0} & \mathbf{I}_d \end{bmatrix}$)

to be similar to

$$\mathbf{q}_o[t+1] = \begin{bmatrix} \mathbf{A} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A} \end{bmatrix} \mathbf{q}_o[t] + \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \mathbf{x}[t],$$

which is of the form described in Theorem 3.1. All modes of the original system are replicated twice and no coupling is involved from the redundant to the original modes. Once the encoding matrix \mathbf{G} is fixed, the additional freedom in choosing the decoding matrix \mathbf{L} can be used to our advantage. For example, if failures are permanent and there is a failure in the first subsystem, we can change the decoding matrix to $\mathbf{L} = \begin{bmatrix} \mathbf{0} & \mathbf{I}_d & \mathbf{0} \end{bmatrix}$. This ensures that the final output is correct; we expand on this idea in Example 3.5. \square

Example 3.4 In this example we show how non-zero redundant modes can be used to construct parity checks with “memory.” These parity checks “remember” an error and allow us to perform checking periodically. Instead of checking at the end of each time step, we can check once every N time steps and still be able to detect and identify transient failures that took place at earlier time steps.

⁷Other decoding and parity check matrices are also possible.

Suppose that \mathcal{S} is an LTI dynamic system as in eq. (3.1). Starting from the standard redundant system \mathcal{H}_σ in eq. (3.3) with $\mathbf{A}_{12} = \mathbf{0}$ and using the similarity transformation $\mathbf{q}_\sigma[t] = \mathcal{T}\mathbf{q}_h[t]$ (where $\mathcal{T} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \\ -\mathbf{C} & \mathbf{I}_s \end{bmatrix}$) and \mathbf{C} is a $d \times s$ matrix, we obtain the following redundant implementation \mathcal{H} :

$$\begin{aligned} \mathbf{q}_h[t+1] &= \left[\begin{array}{c|c} \mathbf{A} & \mathbf{0} \\ \hline \mathbf{CA} - \mathbf{A}_{22}\mathbf{C} & \mathbf{A}_{22} \end{array} \right] \mathbf{q}_h[t] + \left[\begin{array}{c} \mathbf{B} \\ \hline \mathbf{CB} \end{array} \right] \mathbf{x}[t] \\ &\equiv \mathbf{A}\mathbf{q}_h[t] + \mathbf{B}\mathbf{x}[t]. \end{aligned}$$

The encoding, decoding and parity check matrices for system \mathcal{H} are given by

$$\mathbf{G} = \mathcal{T}^{-1}\mathbf{G}_\sigma = \begin{bmatrix} \mathbf{I}_d \\ \mathbf{C} \end{bmatrix}, \quad \mathbf{L} = \mathbf{L}_\sigma\mathcal{T} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \end{bmatrix}, \quad \mathbf{P} = \mathbf{P}_\sigma\mathcal{T} = \begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix}.$$

Without loss of generality, suppose that at time step 0 a transient failure (e.g., noise) corrupts system \mathcal{H} so that the faulty state is given by

$$\mathbf{q}_f[0] = \mathbf{q}_h[0] + \mathbf{e},$$

where \mathbf{e} is an additive error vector that models the effect of the transient failure. If we perform the parity check at the end of time step 0, we will get the following syndrome:

$$\begin{aligned} \mathbf{p}[0] = \mathbf{P}\mathbf{q}_f[0] &= \mathbf{P}\mathbf{q}_h[0] + \mathbf{P}\mathbf{e} \\ &= \mathbf{0} + \mathbf{P}\mathbf{e} \\ &= \begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix} \mathbf{e}. \end{aligned}$$

If transient failures affect only a single variable in the state vector (i.e., if \mathbf{e} is a vector with a single non-zero entry), then we can detect, identify and correct failures as long as the columns of $\mathbf{P} = \begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix}$ are not multiples of each other. For example, if \mathbf{P} is the parity check matrix of a Hamming code (as in eq. (3.5) of Example 3.1), then we can identify the column of \mathbf{P} that is a multiple of the obtained syndrome $\mathbf{p}[0]$, find out exactly what \mathbf{e} was and make the appropriate correction.

If, however, we perform the parity check only periodically (e.g., once every N time steps), the syndrome at time step $N - 1$, given a failure at time step $N - 1 - m$ ($0 \leq m \leq N - 1$), will be

$$\begin{aligned} \mathbf{p}[N - 1] &= \mathbf{P}\mathbf{q}_f[N - 1] = \mathbf{P}\mathbf{q}_h[N - 1] + \mathbf{P}\mathbf{A}^m\mathbf{e} \\ &= \mathbf{0} + \begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix} \left[\begin{array}{c|c} \mathbf{A}^m & \mathbf{0} \\ \hline \mathbf{C}\mathbf{A}^m - \mathbf{A}_{22}^m\mathbf{C} & \mathbf{A}_{22}^m \end{array} \right] \mathbf{e} \\ &= \mathbf{A}_{22}^m \begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix} \mathbf{e}. \end{aligned}$$

This assumes that no other transient failures occurred between time step 0 and $N - 1$. If $\mathbf{A}_{22} = \mathbf{0}$, then the parity check will be $\mathbf{0}$ (i.e., \mathbf{e} will go undetected). More generally, however, we can choose \mathbf{A}_{22} so that the parity check will be non-zero. For example, if $\mathbf{A}_{22} = \mathbf{I}_s$, then we get the same syndrome as if we checked at time step $N - 1 - m$. The problem is, of course, that we will not know what m is (i.e., we will not know when the transient failure took place). This situation can be remedied if we choose a different matrix \mathbf{A}_{22} . For example, if \mathbf{P} is the parity check matrix of a Hamming code and \mathbf{A}_{22} is the diagonal matrix

$$\mathbf{A}_{22} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1/2 & \cdots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \cdots & (1/2)^s \end{bmatrix},$$

then we can easily identify the corrupted state variable. Furthermore, we can find out when the corruption took place (i.e., what m is) and by how much it affected the erroneous state variable (i.e., what \mathbf{e} was). In this particular case we assume that only a single transient failure took place between periodic checks, but the approach can be extended to handle multiple failures. \square

Example 3.5 In the TMR case in Example 3.3 a transient failure in any subsystem was corrected using a voting scheme. When the failure is permanent, however, we would like to avoid the overhead of error-correction at each time step. This can be done in a straight-

forward way in the TMR case: if a failure permanently corrupts the first subsystem (by corrupting gains or adders in its hardware implementation), we can switch our decoding matrix from $L = \begin{bmatrix} I_d & 0 & 0 \end{bmatrix}$ to $L = \begin{bmatrix} 0 & I_d & 0 \end{bmatrix}$ (or $L = \begin{bmatrix} 0 & 0 & I_d \end{bmatrix}$ or others) and ignore the state of the first subsystem. This ensures that the overall state of our fault-tolerant implementation is still correct. We can continue to perform error-detection by comparing the state of the two remaining subsystems, but we have lost the ability to do error-correction. In this example we formalize and generalize this idea.

Consider again the redundant system \mathcal{H} whose state evolution equation is given by eq. (3.2). Under fault-free conditions, $\mathbf{q}_s[t] = L\mathbf{q}_h[t]$ and $\mathbf{q}_h[t] = G\mathbf{q}_s[t]$ for all t . Suppose that we implement this redundant system using a delay-adder-gain interconnection with delay-free paths of unit length. A permanent failure in a gain element manifests itself as a corrupted entry in matrices \mathcal{A} or \mathcal{B} . The i th state variable in $\mathbf{q}_h[t]$ (and other variables at later time steps) will be corrupted if some of the entries⁸ $\mathcal{A}(i, l_1)$ and/or $\mathcal{B}(i, l_2)$ (l_1 in $\{1, 2, \dots, d\}$, l_2 in $\{1, 2, \dots, u\}$) are affected right after time step $t-1$. We assume that we can locate the faulty state variable through the use of some linear error detecting/correcting scheme as in Example 3.1. We do not have control over the entries in \mathcal{A} and \mathcal{B} (i.e., the multipliers, gains and interconnections), but we are allowed to adjust the decoding matrix L to a new matrix L_a . We would like to know which gain or adder corruptions can be tolerated and how to choose L_a .

The first step is to find out which state variables will be corrupted eventually. If at time step N we detect a corruption at the i th state variable, then we know that at time step $N+1$, state variable $q_i[N]$ will corrupt the state variables that depend on it (let M_{i_1} be the set of indices of these state variables — including i); at time step $N+2$, the state variables with indices in set M_{i_1} will corrupt the state variables that depend on them; let their indices be in set M_{i_2} (which includes M_{i_1}); and so on. Eventually, the final set of indices for all corrupted state variables is given by the set M_{i_η} (note that $M_{i_\eta} = M_{i_1} \cup M_{i_2} \cup M_{i_3} \dots \cup M_{i_\eta}$). The sets of indices M_{i_j} for all i in $\{1, 2, \dots, \eta\}$ can be *pre-calculated* in an efficient manner by computing $R(\mathcal{A})$, the *reachability matrix* of \mathcal{A} , as outlined in [76].

⁸We use $A(i, l)$ to denote the element in the i th row and the l th column of matrix A .

Once we have detected an error at the i th state variable, our new decoding matrix L_a (if it exists) should not make use of state variables with indices in M_{i_f} . Equivalently, we can ask the question: does there exist a decoding matrix L_a such that $L_a G_a = I_d$? Here, G_a is the same as the original encoding matrix G except that $G_a(j, l)$ is set to zero for all l in $\{1, 2, \dots, d\}$ and j in M_{i_f} . If G_a is full-column rank, such an L_a exists (any L_a that satisfies $L_a G_a = I_d$ is suitable). We conclude that our redundant system can withstand permanent corruptions of entries in the i th row(s) of A and/or B as long as the resulting G_a is full-column rank.

TMR is clearly a special case of the above formulation: corruption of a state variable of the first subsystem is guaranteed to remain within the first subsystem. Therefore, $M_f \subseteq \{1, 2, \dots, d\}$ and (conservatively) $G_a = \begin{bmatrix} 0 & I_d & I_d \end{bmatrix}^T$. One possible L_a is (among others) $\begin{bmatrix} 0 & I_d & 0 \end{bmatrix}$.

Less obvious is the following case (based on the earlier linear coding scheme in Example 3.1). Consider the system with state evolution as in eq. (3.6). Its decoding matrix is given by $L = \begin{bmatrix} I_4 & 0 \end{bmatrix}$. If $A(2, 2)$ (whose value is .5) becomes corrupted, then the set of indices of corrupted state variables is $M_{2_f} = \{2, 5\}$. Below, we show the original encoding matrix G , together with the new encoding matrix G_a (resulting after the corruption of entry $A(2, 2)$) and a suitable L_a :

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & 0 \\ -1 & -1 & 0 & -1 \\ -1 & 0 & -1 & -1 \end{bmatrix}, \quad G_a = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & -1 \\ -1 & 0 & -1 & -1 \end{bmatrix},$$

$$L_a = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

Using the above \mathbf{L}_a the redundant system can continue to function properly and provide the correct state vector $\mathbf{q}_s[t]$ despite the corrupted entry $\mathcal{A}(2, 2)$. We can still use the parity check matrix of eq. (3.5) for error-detection, except that the checks involving the second and/or fifth state variables (i.e., the first and second checks in $\mathbf{P}\mathbf{q}_h[t]$) will be invalid. \square

3.6 Summary

In this chapter we obtained a characterization of all redundant implementations for a given LTI dynamic system under *linear* encoding and decoding schemes. Based on this characterization, we have shown that the set of available redundant implementations for concurrent error detection and correction in such systems is larger than what was considered in previous work because the redundancy may have its own dynamics and/or it may interact with the original system. We have shown that our method essentially amounts to augmenting the original system with redundant *modes* that are *unreachable* but *observable* under fault-free conditions. Because these additional modes are not excited initially, they manifest themselves only when a failure takes place. Our characterization resembles results on continuous-time LTI system “inclusion” treated in [52], although the issue of creating redundancy for fault tolerance does not seem to have been a motivation for [52].

We have adopted an explicit mapping to hardware (using delays, adders and gains) which allowed us to develop an attractive error model in a way that maps single failures in an adder or multiplier to an error in a single state variable. By employing linear coding techniques, one can develop schemes that detect/correct a fixed number of failures. By investigating the redundant implementations that are possible under a particular error detection/correction scheme, we constructed novel fault-tolerant schemes that make better use of additional hardware or have other desirable properties, such as reconfigurability or memory. We did not directly address criteria that may allow us to “optimally” select the “best” possible redundant implementation; our examples, however, presented a variety of open questions for future research.

It would be interesting to explore extensions of the ideas presented in this chapter to linear systems over other fields, rings [14], or semirings, or to nonlinear/time-varying systems (e.g., by extending the “inclusion principle” ideas in [51, 53]). These generalizations are

unlikely to use similarity transformations to *explicitly* display the original system embedded in the redundant one, but they should allow us to claim that there are some invariant or self-contained dynamics of the redundant system that are isomorphic to the original ones. It would also be interesting to investigate different techniques for mapping to hardware (e.g., using factored state variables, [92]).

The application of these ideas to max-plus systems appears to be particularly interesting, [7, 27, 26, 18, 17]. Max-plus systems are “linear” with respect to two different operations: regular addition is replaced by the MAX operation, whereas multiplication is replaced by addition. The net result is to replace the field of real numbers under addition and multiplication with the *semifield* of reals under the MAX (additive) and + (multiplicative) operations. Redundancy can be introduced in such “generalized linear” systems in ways analogous to those we used for LTI dynamic systems. The absence of an inverse for the MAX operation forces us to consider issues related with error-detection and robust performance rather than error-correction. These ideas may be useful in building robust flow networks, real-time systems and scheduling algorithms.

Chapter 4

Redundant Implementations of Linear Finite-State Machines

4.1 Introduction

In this chapter we extend the ideas of Chapter 3 to linear finite-state machines (LFSM's). We focus on *linear* coding techniques and reflect the state of the original LFSM into a larger LFSM in a way that preserves the state of the original system in some *linearly* encoded form. At the end of each time step, we use the redundancy in the state representation of the larger system to perform error detection and correction. Our approach results in a complete characterization of the class of appropriate redundant implementations for LFSM's (as was the case for linear time-invariant dynamic systems). To demonstrate the implications of our approach, we build hardware implementations using 2-input XOR gates and single-bit memory elements (flip-flops). We also discuss how to choose the redundant implementation that uses the least number of 2-input XOR gates.

This chapter is organized as follows. In Section 4.2 we provide an introduction to LFSM's and in Section 4.3 we characterize appropriate redundant implementations. In Section 4.4 we discuss how our approach can be used, through a variety of examples.

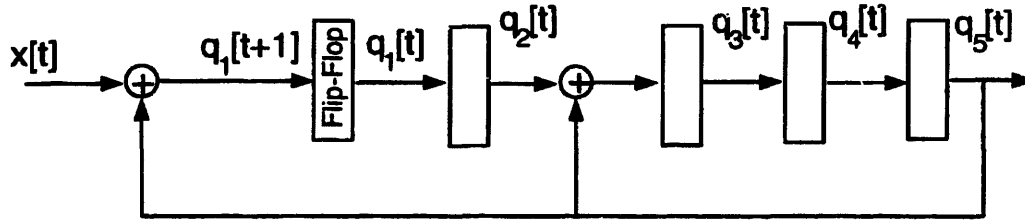


Figure 4-1: Example of a linear feedback shift register.

4.2 Linear Finite-State Machines

Linear finite-state machines (LFSM's) form a very general class of finite-state machines¹ with a variety of applications, [13, 46]. They include linear feedback shift registers [41, 70, 28, 29], sequence enumerators and random number generators [41], encoders and decoders for linear error-correcting codes [81, 11, 111], and cellular automata, [19, 20].

The state evolution of an LFSM \mathcal{S} is given by

$$\mathbf{q}_s[t+1] = \mathbf{A}\mathbf{q}_s[t] \oplus \mathbf{B}\mathbf{x}[t], \quad (4.1)$$

where t is the discrete-time index, $\mathbf{q}_s[t]$ is the *state vector* and $\mathbf{x}[t]$ is the *input vector*. We assume that $\mathbf{q}_s[\cdot]$ is d -dimensional, $\mathbf{x}[\cdot]$ is u -dimensional, and \mathbf{A} , \mathbf{B} are constant matrices of appropriate dimensions. All vectors and matrices have entries from $GF(2)$, the Galois field² of order 2, i.e., they are either “0” or “1” (more generally they can be drawn from any other finite field). In eq. (4.1) and for the rest of this chapter, matrix-vector multiplication and vector-vector addition are performed as usual except that element-wise addition and multiplication are taken modulo-2. Operation \oplus in (4.1) denotes vector addition modulo-2. Note that “-1” is the same as “+1” when performing addition and multiplication modulo-2; we will feel free to use both notations to give more insight regarding a check or a transformation that takes place.

¹A discussion on the power of LFSM's and related references can be found in [115].

²The finite field $GF(l)$ is the *unique* set of l elements GF together with two binary operations \oplus and \otimes such that

1. GF forms a group under \oplus with identity 0.
2. $GF - \{0\}$ forms a commutative group under \otimes with identity 1.
3. Operation \otimes distributes over \oplus , i.e., for all $f_1, f_2, f_3 \in GF$, $f_1 \otimes (f_2 \oplus f_3) = (f_1 \otimes f_2) \oplus (f_1 \otimes f_3)$.

The order l of a finite field has to be a prime number or a power of a prime number, [13, 46].

Example 4.1 The linear feedback shift register (LFSR) in Figure 4-1 is a simple example of an LFSM. It is implemented using single-bit memory elements (flip-flops) and 2-input XOR gates (a 2-input XOR gate performs modulo-2 addition on its binary inputs and is denoted by \oplus in the figure). The state evolution equation of the LFSR in Figure 4-1 is given by

$$\begin{aligned} \mathbf{q}_s[t+1] &= \mathbf{A}\mathbf{q}_s[t] \oplus \mathbf{b}x[t] \\ &= \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{q}_s[t] \oplus \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} x[t]. \end{aligned}$$

Note that when $x[\cdot] = 0$ and $\mathbf{q}_s[0] \neq \mathbf{0}$, the LFSR acts as an autonomous sequence generator. It goes through all non-zero states (essentially counting from 1 to 31). For example, if initialized at $\mathbf{q}_s[0] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}^T$, the LFSR goes through states $\mathbf{q}_s[1] = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix}^T$, $\mathbf{q}_s[2] = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix}^T$, ..., $\mathbf{q}_s[30] = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \end{bmatrix}^T$, $\mathbf{q}_s[31] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}^T$, and so forth. \square

Given a description of the state evolution of an LFSM as in the example above, there are a number of implementations with 2-input XOR gates and flip-flops. Just as in the case of LTI dynamic systems in Chapter 3 (see Section 3.4), we assume that each bit in the next-state vector $\mathbf{q}_s[t+1]$ is calculated using a *separate* set of 2-input XOR gates. This implies that a failure in a single XOR gate can corrupt at most one bit in vector $\mathbf{q}_s[t+1]$. We also assume that the calculation of each bit in $\mathbf{q}_s[t+1]$ is based on the bits of $\mathbf{q}_s[t]$ that are explicitly specified by the “1’s” in the matrix \mathbf{A} of the state evolution equation (e.g., the third bit of $\mathbf{q}_s[t+1]$ in Example 4.1 is calculated based on the second and fifth bits of $\mathbf{q}_s[t]$). Under these assumptions, we can focus on detecting/correcting single-bit errors because a single failure in an adder or in an XOR gate corrupts at most one bit in the state vector of a redundant implementation.

One can obtain an LFSM \mathcal{S}' (with d -dimensional state vector $\mathbf{q}'_s[t]$) that is *similar* to \mathcal{S}

in eq. (4.1) through a similarity transformation, [13, 46]:

$$\begin{aligned}\mathbf{q}'_s[t+1] &= \underbrace{(\mathbf{T}^{-1}\mathbf{A}\mathbf{T})}_{\mathbf{A}'}\mathbf{q}'_s[t] \oplus \underbrace{(\mathbf{T}^{-1}\mathbf{B})}_{\mathbf{B}'}\mathbf{x}[t] \\ &\equiv \mathbf{A}'\mathbf{q}'_s[t] \oplus \mathbf{B}'\mathbf{x}[t],\end{aligned}$$

where \mathbf{T} is an *invertible* $d \times d$ binary matrix such that $\mathbf{q}_s[t] = \mathbf{T}\mathbf{q}'_s[t]$. The initial conditions for the transformed LFSM can be obtained as $\mathbf{q}'_s[0] = \mathbf{T}^{-1}\mathbf{q}_s[0]$.

It can be shown that any LFSM with state evolution as in eq. (4.1) can be put via a similarity transformation in a form where the matrix \mathbf{A}' is in *classical canonical form*, [13]:

$$\mathbf{A}' = \begin{bmatrix} \mathbf{A}_1 & & & \\ & \mathbf{A}_2 & & \\ & & \ddots & \\ & & & \mathbf{A}_p \end{bmatrix},$$

where each \mathbf{A}_i ($1 \leq i \leq p$) is given by

$$\mathbf{A}_i = \begin{bmatrix} \mathbf{C}_{i_1} & & & \\ & \mathbf{C}_{i_2} & & \\ & & \ddots & \\ & & & \mathbf{C}_{i_q} \end{bmatrix}.$$

Each \mathbf{C}_i , ($1 \leq j \leq q$) looks like

$$\mathbf{C}_{i_j} = \begin{bmatrix} \mathbf{D}_{i_j} & \mathbf{E}_{i_j} & & & \\ & \mathbf{D}_{i_j} & \mathbf{E}_{i_j} & & \\ & & \mathbf{D}_{i_j} & \ddots & \\ & & & \ddots & \mathbf{E}_{i_j} \\ & & & & \mathbf{D}_{i_j} \end{bmatrix},$$

where \mathbf{D}_{i_j} and \mathbf{E}_{i_j} are given by

$$\mathbf{D}_{i,j} = \begin{bmatrix} 0 & 0 & \cdots & 0 & * \\ 1 & 0 & \cdots & 0 & * \\ 0 & 1 & \ddots & \vdots & * \\ \vdots & & \ddots & 0 & * \\ 0 & \cdots & 0 & 1 & * \end{bmatrix}, \quad \mathbf{E}_{i,j} = \begin{bmatrix} 0 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

Each $*$ symbol could be a “0” or a “1”. What is important about this form is that there are at most two “1’s” in each row of \mathbf{A}' ; this means that each bit in the next-state vector $\mathbf{q}'_s[t+1]$ can be generated based on at most two bits of the current state vector $\mathbf{q}'_s[t]$. This property of the classical canonical form is critical in our analysis in Chapter 6.

4.3 Characterization of Redundant Implementations

Our analysis in this section focuses on the structure of the redundant implementations and assumes that the error detection/correction procedure is fault-free³. In Chapter 6 we extend our approach to handle failures in the error-correcting mechanism.

We look for ways of embedding the given LFSM \mathcal{S} (with d state variables and state evolution as in eq. (4.1)) into a redundant LFSM \mathcal{H} with η state variables ($\eta \equiv d + s$, $s > 0$) and state evolution

$$\mathbf{q}_h[t+1] = \mathcal{A}\mathbf{q}_h[t] \oplus \mathcal{B}\mathbf{x}[t]. \quad (4.2)$$

The initial state $\mathbf{q}_h[0]$ and matrices \mathcal{A} , \mathcal{B} need to be chosen such that the unfailed state $\mathbf{q}_h[t]$ of \mathcal{H} at time step t provides complete information about $\mathbf{q}_s[t]$, the state of the original LFSM \mathcal{S} , through a decoding mapping, and vice-versa. We restrict ourselves to decoding and encoding techniques that are linear in $GF(2)$ (in general encoding and decoding need not be linear). Specifically, we assume that there exist

- a $d \times \eta$ binary *decoding* matrix \mathbf{L} such that $\mathbf{q}_s[t] = \mathbf{L}\mathbf{q}_h[t]$ for all t ,
- an $\eta \times d$ binary *encoding* matrix \mathbf{G} such that $\mathbf{q}_h[t] = \mathbf{G}\mathbf{q}_s[t]$ for all t .

³As mentioned in Chapter 3, the assumption that error detection/correction is fault-free is reasonable if the complexity of detection/correction is considerably less than the complexity of computing the state evolution of the system.

Under the above assumptions, the redundant machine \mathcal{H} concurrently simulates the original machine \mathcal{S} ($\mathbf{q}_s[t] = \mathbf{L}\mathbf{q}_h[t]$). Furthermore, there is a one-to-one correspondence between the states in \mathcal{S} and the states in \mathcal{H} (i.e., $\mathbf{q}_h[t] = \mathbf{G}\mathbf{q}_s[t]$ and $\mathbf{q}_s[t] = \mathbf{L}\mathbf{q}_h[t]$). It is easy to show that $\mathbf{L}\mathbf{G} = \mathbf{I}_d$ (where \mathbf{I}_d is the $d \times d$ identity matrix).

The redundant machine \mathcal{H} enforces an (η, d) linear code on the state of the original machine, [81, 11, 111]. An (η, d) linear code uses η bits to represent d bits of information and is defined in $GF(2)$ by an $\eta \times d$ generator matrix \mathbf{G} with full-column rank. The d dimensional vector \mathbf{q}_s is *uniquely* represented by the η dimensional vector (codeword) $\mathbf{q}_h = \mathbf{G}\mathbf{q}_s$. Error-detection is straightforward: under fault-free conditions, the redundant state vector must be in the column space of \mathbf{G} ; therefore, all we need to check is that at each time step t the redundant state $\mathbf{q}_h[t]$ lies in the column space of \mathbf{G} (in coding theory terminology, we need to check that $\mathbf{q}_h[t]$ is a codeword of the linear code that is generated by \mathbf{G} , [81, 11, 111]). Equivalently, we can check that $\mathbf{q}_h[t]$ is in the null space of an appropriate *parity check matrix* \mathbf{P} , so that $\mathbf{P}\mathbf{q}_h[t] = \mathbf{0}$. The parity check matrix has row rank $\eta - d \equiv s$ and satisfies $\mathbf{P}\mathbf{G} = \mathbf{0}$. Error-correction associates with each valid state in \mathcal{H} (of the form $\mathbf{G}\mathbf{q}_s[\cdot]$), a unique subset of invalid states that get corrected to that particular valid state⁴. Error-correction can be performed using any of the methods used in the communications setting (e.g., syndrome table decoding or iterative decoding, [81, 11, 111, 37]).

The following theorem provides a parameterization of all redundant implementations for a given LFSM under a given linear coding scheme:

Theorem 4.1 *In the setting described above, LFSM \mathcal{H} (of dimension $\eta \equiv d + s$, $s > 0$ and state evolution as in eq. (4.2)) is a redundant version of \mathcal{S} if and only if it is similar to a standard redundant LFSM \mathcal{H}_σ whose state evolution equation is given by*

$$\mathbf{q}_\sigma[t+1] = \begin{bmatrix} \mathbf{A} & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{A}_{22} \end{bmatrix} \mathbf{q}_\sigma[t] \oplus \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix} \mathbf{x}[t]. \quad (4.3)$$

Here, \mathbf{A} and \mathbf{B} are the matrices in eq. (4.1), \mathbf{A}_{22} is an $s \times s$ binary matrix that describes the dynamics of the redundant modes that have been added, and \mathbf{A}_{12} is a $d \times s$ binary matrix that

⁴This subset usually contains η -dimensional vectors with small *Hamming* distance from the associated valid codeword. Recall that the Hamming distance between two binary vectors $\mathbf{x} = (x_1, x_2, \dots, x_\eta)$ and $\mathbf{y} = (y_1, y_2, \dots, y_\eta)$ is the number of positions at which \mathbf{x} and \mathbf{y} differ, [81, 11, 111].

describes the coupling from the redundant to the non-redundant modes. Associated with this standard redundant LFSM is the standard decoding matrix $\mathbf{L}_\sigma = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \end{bmatrix}$, the standard encoding matrix $\mathbf{G}_\sigma = \begin{bmatrix} \mathbf{I}_d \\ \mathbf{0} \end{bmatrix}$ and the standard parity check matrix $\mathbf{P}_\sigma = \begin{bmatrix} \mathbf{0} & \mathbf{I}_s \end{bmatrix}$.

Proof: The proof is similar to the proof of Theorem 3.1 in Chapter 3 and we omit it. \square

Given a pair of encoding and decoding matrices \mathbf{L} and \mathbf{G} (they need to satisfy $\mathbf{L}\mathbf{G} = \mathbf{I}_d$), and an LFSM \mathcal{S} , Theorem 4.1 completely characterizes all possible redundant LFSM's \mathcal{H} . Since the choice of the binary matrices \mathbf{A}_{12} and \mathbf{A}_{22} is completely free, there are multiple redundant implementations of LFSM \mathcal{S} for the given \mathbf{L} and \mathbf{G} .

4.4 Examples of Fault-Tolerant Schemes

In this section we discuss examples of redundant implementations for LFSM's. Example 4.2 illustrates that, even for a simple checksum scheme, non-zero \mathbf{A}_{12} and/or \mathbf{A}_{22} can lead to designs that make better use of redundancy. Example 4.3 discusses some interesting trade-offs between computation and memory for redundant LFSM implementations in the context of convolutional encoders for communication systems. More generally, we discuss ways to explore different redundant implementations in order to minimize hardware overhead.

Example 4.2 Suppose that the original LFSM \mathcal{S} that we would like to protect is the linear feedback shift register shown in Figure 4-1. In order to detect a single failure in an XOR gate, we can use an extra “checksum” state variable (as was suggested for linear time-invariant dynamic systems in [50] — see the discussions in Example 3.2 and also in [61, 94]). The resulting redundant LFSM \mathcal{H} has six state variables and state evolution

$$\mathbf{q}_h[t+1] = \left[\begin{array}{c|c} \mathbf{A} & \mathbf{0} \\ \hline \mathbf{c}^T \mathbf{A} & \mathbf{0} \end{array} \right] \mathbf{q}_h[t] \oplus \left[\begin{array}{c} \mathbf{b} \\ \hline \mathbf{c}^T \mathbf{b} \end{array} \right] x[t],$$

where $\mathbf{c}^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}$, i.e.,

$$\mathbf{q}_h[t+1] = \left[\begin{array}{ccccc|c} 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 1 & 1 & 1 & 1 & 0 & 0 \end{array} \right] \mathbf{q}_h[t] \oplus \left[\begin{array}{c} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ \hline 1 \end{array} \right] x[t].$$

Under fault-free conditions, the added state variable is always the modulo-2 sum of all other state variables (which are the same as the original state variables in LFSM \mathcal{S}).

The above approach is easily seen to be consistent with our setup. Specifically, the encoding, decoding and parity check matrices are given by:

$$\mathbf{G} = \left[\begin{array}{c} \mathbf{I}_5 \\ \mathbf{c}^T \end{array} \right] = \left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 \end{array} \right], \quad \mathbf{L} = [\mathbf{I}_5 \mid \mathbf{0}] = \left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \end{array} \right],$$

$$\mathbf{P} = [-\mathbf{c}^T \mid 1] = [-1 \quad -1 \quad -1 \quad -1 \quad -1 \mid 1].$$

Using the transformation $\mathbf{q}_\sigma[t] = \mathcal{T} \mathbf{q}_h[t]$ where $\mathcal{T} = \left[\begin{array}{cc} \mathbf{I}_5 & \mathbf{0} \\ \mathbf{c}^T & 1 \end{array} \right]$, we see that \mathcal{H} is similar to a standard redundant LFSM \mathcal{H}_σ with state evolution given by

$$\mathbf{q}_\sigma[t+1] = \left[\begin{array}{cc} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & 0 \end{array} \right] \mathbf{q}_\sigma[t] \oplus \left[\begin{array}{c} \mathbf{b} \\ 0 \end{array} \right] x[t].$$

Note that both \mathbf{A}_{12} and \mathbf{A}_{22} are set to zero.

As stated earlier, there are multiple redundant implementations with the same encoding, decoding and parity check matrices. For the scenario described here, there are exactly 2^5 different LFSM's (we get a different system for each combination of choices for entries in matrices \mathbf{A}_{12} and \mathbf{A}_{22}). One such choice could have been to set

$$\mathbf{A}_{12} = \mathbf{0}, \quad \mathbf{A}_{22} = [1],$$

and use the same transformation ($\mathbf{q}_o[t] = \mathcal{T}\mathbf{q}_{h'}[t]$, $\mathcal{T} = \begin{bmatrix} \mathbf{I}_5 & \mathbf{0} \\ \mathbf{c}^T & 1 \end{bmatrix}$) to get a redundant LFSM \mathcal{H}' with state evolution equation

$$\mathbf{q}_{h'}[t+1] = \left[\begin{array}{c|c} \mathbf{A} & \mathbf{0} \\ \hline \mathbf{c}^T \mathbf{A} - \mathbf{A}_{22} \mathbf{c}^T & \mathbf{A}_{22} \end{array} \right] \mathbf{q}_{h'}[t] \oplus \left[\begin{array}{c} \mathbf{b} \\ \hline \mathbf{c}^T \mathbf{b} \end{array} \right] x[t],$$

or

$$\mathbf{q}_{h'}[t+1] = \left[\begin{array}{ccccc|c} 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 1 \end{array} \right] \mathbf{q}_{h'}[t] \oplus \left[\begin{array}{c} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ \hline 1 \end{array} \right] x[t].$$

Both redundant LFSM's \mathcal{H} and \mathcal{H}' have the same encoding, decoding and parity check matrices, and are able to concurrently detect single-bit errors in the redundant state vector (and therefore a failure in a single XOR gate, according to our assumptions about hardware implementation). Clearly, the complexity in \mathcal{H}' is lower than in \mathcal{H} . More generally, as illustrated in this example for the case of a non-zero \mathbf{A}_{22} , we obtain more efficient redundant implementations by exploiting the dynamics of the redundant modes (given by \mathbf{A}_{22}) and/or their coupling with the original system (given by \mathbf{A}_{12}). The gain is not only in terms of the hardware involved, but also in terms of minimizing the probability of failure (since XOR gates may fail). More generally, one of the problems in encoding the state of dynamic systems (in order to provide fault tolerance) has been the cost associated with generating the redundant bits, [75]. For example, in the original implementation of the checksum scheme, generating one additional bit costs more (in terms of 2-input XOR gates) than the linear feedback shift register altogether. \square

Example 4.3 At the top of Figure 4-2 we see an example of a rate 1/3 *convolutional encoder* (the code was taken from [111]). The encoder takes a sequence of bits $x[t]$ and encodes it into three output sequences ($y_1[t]$, $y_2[t]$ and $y_3[t]$) as shown in the figure. The encoding mechanism can be seen as an LFSM with state evolution equation given by

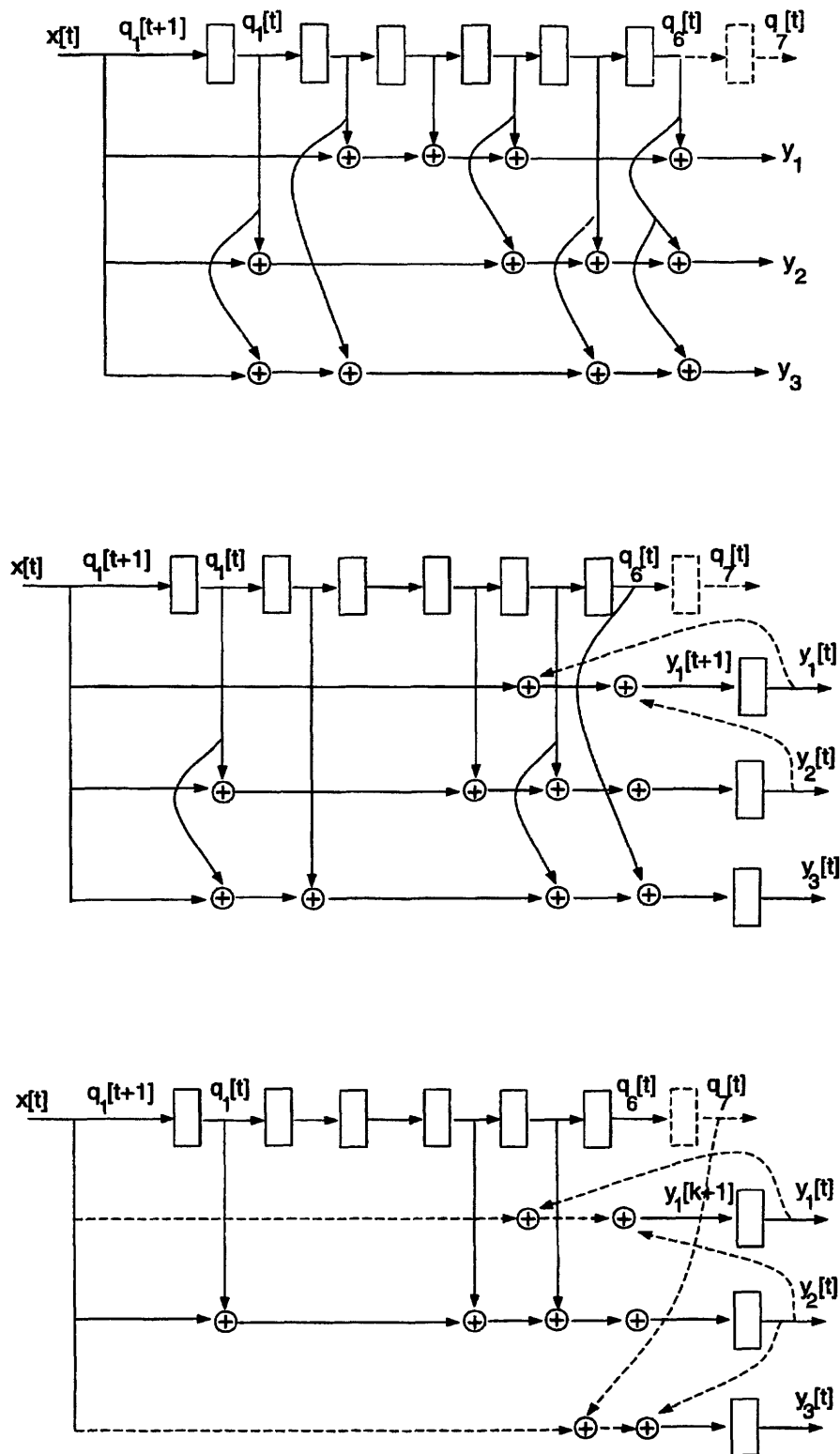


Figure 4-2: Three different implementations of a convolutional encoder.

$$\begin{aligned}
\mathbf{q}_s[t+1] &\equiv \begin{bmatrix} q_1[t+1] \\ q_2[t+1] \\ q_3[t+1] \\ q_4[t+1] \\ q_5[t+1] \\ q_6[t+1] \\ q_7[t+1] \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{q}_s[t] \oplus \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} x[t] \\
&\equiv \mathbf{A}\mathbf{q}_s[t] \oplus \mathbf{b}x[t]
\end{aligned}$$

and output⁵

$$\begin{aligned}
\mathbf{y}[t+1] &\equiv \begin{bmatrix} y_1[t+1] \\ y_2[t+1] \\ y_3[t+1] \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \mathbf{q}_s[t] \oplus \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} x[t] \\
&\equiv \mathbf{F}\mathbf{q}_s[t] \oplus \mathbf{d}x[t].
\end{aligned}$$

(Note that we have included $q_7[t]$ as a “dummy” state variable because we will use it in our analysis later on.)

Since at each time step t we calculate output values $y_1[t]$, $y_2[t]$ and $y_3[t]$, we can consider *saving* them in designated single-bit memory elements. If we do this, we have a redundant implementation of an LFSM with state evolution equation

$$\begin{aligned}
\mathbf{q}_h[t+1] &\equiv \begin{bmatrix} \mathbf{q}_s[t+1] \\ \mathbf{y}[t+1] \end{bmatrix} = \left[\begin{array}{c|c} \mathbf{A} & \mathbf{0} \\ \mathbf{F} & \mathbf{0} \end{array} \right] \mathbf{q}_h[t] \oplus \left[\begin{array}{c} \mathbf{b} \\ \mathbf{d} \end{array} \right] x[t] \\
&= \left[\begin{array}{ccccccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{array} \right] \mathbf{q}_h[t] \oplus \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \hline 1 \\ 1 \\ 1 \end{bmatrix} x[t],
\end{aligned}$$

and whose encoding and decoding matrices are given by:

⁵What we denote by $\mathbf{y}[t+1]$ usually gets denoted by $\mathbf{y}[t]$.

$$\mathbf{L} = [\mathbf{I}_7 \mid \mathbf{0}] , \quad \mathbf{G} = \left[\begin{array}{c|cccccc} & \mathbf{I}_7 & & & & & \\ \hline 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 \end{array} \right] .$$

Therefore, just as we did in Example 4.2, we can use non-zero redundant dynamics ($\mathbf{A}_{22} \neq \mathbf{0}$) and/or coupling ($\mathbf{A}_{12} \neq \mathbf{0}$) in order to reduce the number of 2-input XOR gates that are used. The encoder in the middle of Figure 4-2 is the result of such a minimization with the restriction that state variable $q_7[\cdot]$ not be used; this reduces the number of 2-input XOR gates by two. If we allow the use of $q_7[\cdot]$ as shown at the bottom of Figure 4-2, we can reduce the number of XOR gates even more.

Therefore, we see an interesting tradeoff between single-bit memory elements (flip-flops) and computation (2-input XOR gates) in the context of redundant implementations of LFSM's. The more state variables we remember the easier it is to generate the desired output. If we remember more outputs and states (e.g., if we add a state variable $q_8[t]$ and previous outputs $y_1[t-1]$, $y_2[t-1]$, $y_3[t-1]$), we can do even better. \square

Next, we describe how to systematically minimize the number of 2-input XOR gates in a redundant implementation.

Construction of Redundant LFSM's Using Systematic Linear Codes

For a given LFSM we can use Theorem 4.1 to construct all redundant implementations that are encoded according to a given linear code. In fact, if the linear code is systematic⁶, one can algorithmically find the redundant LFSM that uses minimal hardware overhead (i.e., minimal number of 2-input XOR gates).

Problem Formulation: Let \mathcal{S} be an LFSM with d state variables and state evolution as in eq. (4.1). Given encoding, decoding and parity check matrices

$$\mathbf{G} = \left[\begin{array}{c} \mathbf{I}_d \\ \mathbf{C} \end{array} \right] , \quad \mathbf{L} = [\mathbf{I}_d \mid \mathbf{0}] , \quad \mathbf{P} = [\mathbf{C} \mid \mathbf{I}_s]$$

⁶An (η, d) systematic linear code is one whose generator matrix is an $\eta \times d$ binary matrix of the form $\mathbf{G} = \left[\begin{array}{c} \mathbf{I}_d \\ \mathbf{C} \end{array} \right]$. Any linear code can be transformed into an equivalent systematic linear code, [111].

(where \mathbf{C} is an $s \times d$ binary matrix), construct the redundant LFSM \mathcal{H} (of dimension $\eta = d + s, s > 0$ and state evolution as in eq. (4.2)) that uses the *minimum* number of 2-input XOR gates.

Solution: From Theorem 4.1, we know that any appropriate redundant implementation will be similar to a standard LFSM \mathcal{H}_σ . Specifically, there exists an $\eta \times \eta$ matrix \mathcal{T} such that

$$\mathcal{A} = \mathcal{T}^{-1} \begin{bmatrix} \mathbf{A} & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{A}_{22} \end{bmatrix} \mathcal{T}, \quad \mathcal{B} = \mathcal{T}^{-1} \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix},$$

where the choices for \mathbf{A}_{12} and \mathbf{A}_{22} are arbitrary.

Moreover, we know that

$$\mathbf{G} = \mathcal{T}^{-1} \mathbf{G}_\sigma, \quad \mathbf{L} = \mathbf{L}_\sigma \mathcal{T}, \quad \mathbf{P} = \mathbf{P}_\sigma \mathcal{T},$$

which establish that \mathcal{T} is given by

$$\mathcal{T} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \\ \mathbf{C} & \mathbf{I}_s \end{bmatrix}.$$

(One easily checks that $\mathcal{T}^{-1} = \mathcal{T}$ over $GF(2)$, which is consistent with the choice of \mathbf{G} .)

Therefore, we completely know matrices \mathcal{A} and \mathcal{B} in terms of \mathbf{A}_{12} and \mathbf{A}_{22} :

$$\begin{aligned} \mathcal{A} &= \mathcal{T}^{-1} \begin{bmatrix} \mathbf{A} & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{A}_{22} \end{bmatrix} \mathcal{T} = \left[\begin{array}{c|c} \mathbf{A} \oplus \mathbf{A}_{12}\mathbf{C} & \mathbf{A}_{12} \\ \hline \mathbf{CA} \oplus \mathbf{CA}_{12}\mathbf{C} \oplus \mathbf{A}_{22}\mathbf{C} & \mathbf{CA}_{12} \oplus \mathbf{A}_{22} \end{array} \right], \\ \mathcal{B} &= \mathcal{T}^{-1} \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{B} \\ \mathbf{CB} \end{bmatrix}. \end{aligned}$$

In order to find the system with the minimal number of 2-input XOR gates, we need to choose \mathbf{A}_{12} and \mathbf{A}_{22} so that we minimize⁷ the number of “1’s” in \mathcal{A} . For example, we can search through all $2^{\eta \times s}$ possibilities (each entry can be either a “0” or a “1”) to find the

⁷Assuming that each row will have at least one “1”.

one that minimizes the number of “1’s” in \mathcal{A} . Next, we discuss a more efficient approach.

Minimization Algorithm:

1. Ignore the bottom s rows of \mathcal{A} (we will soon show why we can do this) and optimize the cost in the top d rows. Each row of matrix \mathbf{A}_{12} can be optimized independently from the other rows (clearly the j th row of matrix \mathbf{A}_{12} does not influence the structure of the other rows of \mathcal{A}). An exhaustive search of all possibilities in each row will look through 2^s different cases. Thus, the minimization for the top d rows will take $d2^s$ steps.
2. Having chosen the entries of \mathbf{A}_{12} , proceed in the exact same way for the last s rows of \mathcal{A} (once \mathbf{A}_{12} is known, the problem has the same structure as for the top d rows). Exhaustive search for each row will search 2^s cases; the total cases needed will be $s2^s$.

The algorithm above searches through a total of $\eta 2^s = (d + s)2^s$ cases instead of $2^{7 \times s}$.

The only issue that remains to be resolved is, of course, whether choosing \mathbf{A}_{12} first (based only on the top d rows of matrix \mathcal{A}) is actually optimal. We will show that this is the case by contradiction: suppose that we have chosen \mathbf{A}_{12} (as in Step 1 of the algorithm), but that there exists a matrix $\mathbf{A}'_{12} \neq \mathbf{A}_{12}$, which together with a choice of \mathbf{A}'_{22} , minimizes the number of “1’s” in \mathcal{A} . Let $\mathbf{A}_{22} = \mathbf{A}'_{22} \oplus \mathbf{CA}'_{12} \oplus \mathbf{CA}_{12}$; matrix \mathcal{A} is then given by

$$\begin{aligned} \mathcal{A} &= \left[\begin{array}{c|c} \mathbf{A} \oplus \mathbf{A}_{12}\mathbf{C} & \mathbf{A}_{12} \\ \hline \mathbf{CA} \oplus \mathbf{CA}_{12}\mathbf{C} \oplus \mathbf{A}_{22}\mathbf{C} & \mathbf{CA}_{12} \oplus \mathbf{A}_{22} \end{array} \right] \\ &= \left[\begin{array}{c|c} \mathbf{A} \oplus \mathbf{A}_{12}\mathbf{C} & \mathbf{A}_{12} \\ \hline \mathbf{CA} \oplus \mathbf{CA}'_{12}\mathbf{C} \oplus \mathbf{A}'_{22}\mathbf{C} & \mathbf{CA}'_{12} \oplus \mathbf{A}'_{22} \end{array} \right]. \end{aligned}$$

Clearly, this choice of \mathbf{A}_{22} has the same effect in the bottom s rows as the choice \mathbf{A}'_{12} and \mathbf{A}'_{22} . Since by assumption \mathbf{A}_{12} was a better choice in minimizing the number of “1’s” in the top d rows, we have a contradiction: choices \mathbf{A}'_{12} and \mathbf{A}'_{22} are suboptimal (they are worse than choices \mathbf{A}_{12} and \mathbf{A}_{22}). \square

Example 4.4 Consider the autonomous LFSM with state evolution $\mathbf{q}_s[t+1] = \mathbf{A}\mathbf{q}_s[t]$ and matrix \mathbf{A} given by

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

This LFSM goes through all non-zero 9-bit sequences, essentially counting from 1 to $2^9 - 1$ (because its feedback corresponds to a primitive polynomial, [111, 41]). We want to construct a redundant machine that uses four additional state variables and has encoding matrix $\mathbf{G} = \begin{bmatrix} \mathbf{I}_9 \\ \mathbf{C} \end{bmatrix}$, decoding matrix $\mathbf{L} = \begin{bmatrix} \mathbf{I}_9 & \mathbf{0} \end{bmatrix}$ and parity check matrix $\mathbf{P} = \begin{bmatrix} \mathbf{C} & \mathbf{I}_4 \end{bmatrix}$, where \mathbf{C} is given by

$$\mathbf{C} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Note that this choice of \mathbf{P} allows single-error correction because all of its columns are different.

If we use the minimization algorithm described earlier, we find that the choice that minimizes the number of XOR gates is

$$\mathbf{A}_{12} = \mathbf{0}, \quad \mathbf{A}_{22} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

Matrix \mathbf{A} for the resulting LFSM is given by

$$\begin{aligned}
A &= \left[\begin{array}{c|c} A \oplus A_{12}C & A_{12} \\ \hline CA \oplus CA_{12}C \oplus A_{22}C & CA_{12} \oplus A_{22} \end{array} \right] \\
&= \left[\begin{array}{c|c} \begin{array}{cccccccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array} & \begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array} \end{array} \right] \\
&\quad \left[\begin{array}{c|c} \begin{array}{cccccccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{array} & \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array} \end{array} \right]
\end{aligned}$$

It requires only nine 2-input XOR gates (as opposed to sixteen gates required by the realization that sets A_{12} and A_{22} to zero). Note that the original machine uses a single XOR gate. \square

4.5 Summary

In this section we obtained a characterization of all redundant implementations for a given LFSM under linear encoding and decoding schemes. Based on this characterization, we presented a variety of redundant implementations that can be used by our two-stage approach for fault tolerance. Future work can focus on describing in more detail ways to minimize the cost of redundant implementations by exploiting redundant dynamics and coupling. It would also be interesting to study whether the two-stage approach to fault tolerance can employ convolutional rather than block coding techniques. Note that so far we have assumed that error-correction is fault-free. In Chapter 6 we relax this assumption by allowing failures in the error detecting/correcting mechanism and by discussing methods to handle them effectively and efficiently.

Chapter 5

Failure Monitoring in Discrete Event Systems Using Redundant Petri Net Implementations

5.1 Introduction

In this chapter we apply our methodology for fault tolerance in dynamic systems to Petri net models of discrete event systems (DES's). This results in techniques for monitoring failures and other activity in such systems. Our approach replaces the original Petri net with a *redundant* one (with more places, tokens and/or transitions) in a way that preserves the state, evolution and properties of the original Petri net in some encoded form. We systematically develop schemes for monitoring failures in a DES by focusing on the class of *separate* redundant Petri net implementations. These implementations retain the functionality of the original Petri net, but use additional places and tokens in order to impose invariant conditions; by performing *linear* consistency checks, our monitoring schemes are able to locate and identify failures in the overall system. The constructions that we propose are attractive because they automatically point out the additional connections that are necessary in order to allow failure monitoring, and may not require explicit acknowledgments from each activity. Furthermore, they can be made robust to failures and can be adapted to incorporate configuration changes in the original system or to impose restrictions in the

information that is available to the monitors. For example, we can design our schemes so that they do not require information about system activity that is unavailable or gets corrupted. We also discuss how to construct non-separate redundant Petri net implementations and analyze briefly some of their potential advantages. The resulting monitoring schemes are simple and straightforward to design. Future work should examine designing and optimizing monitoring schemes that achieve the desired objective while minimizing the cost associated with them (e.g., the number of communication links required, the number of acknowledgments, the size of monitor, and others).

The chapter is organized as follows. In Sections 5.2 and 5.3 we provide a brief introduction to Petri net modeling of DES's and discuss the types of failures that we will be protecting against. In Section 5.4 we construct monitoring schemes using separate redundant Petri net implementations. In Section 5.5 we discuss non-separate redundant Petri net implementations and analyze some of their potential advantages. In Section 5.6 we describe how our redundant implementations can be used to facilitate control or to detect "illegal" or unmodeled behavior in a DES.

5.2 Petri Net Models of Discrete Event Systems

Petri nets are a graphical and mathematical model for a variety of information and processing systems, [73]. Due to their power and flexibility, Petri nets are particularly relevant to the study of concurrent, asynchronous, distributed, nondeterministic, and/or stochastic systems, [7, 18]. They are used to model communication protocols, manufacturing systems [30], or more general DES's, [17]. The extended spectrum of applications, their size and distributed nature, and the diverse implementations involved in modern Petri nets necessitate elaborate control and failure detection/recovery mechanisms.

A Petri net \mathcal{S} is represented by a directed, bipartite graph with two kinds of nodes: *places* (denoted by $\{p_1, p_2, \dots, p_d\}$ and drawn as circles) and *transitions* (denoted by $\{t_1, t_2, \dots, t_u\}$ and drawn as rectangles). Weighted directed arcs connect transitions to places and vice-versa (but there are no connections from a place to a place or from a transition to a transition). The arc weights have to be nonnegative integers; we use b_{ij}^- to denote the

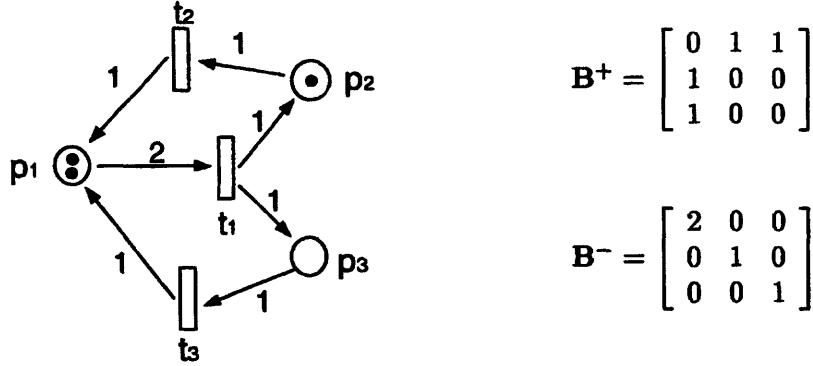


Figure 5-1: Example of a Petri net with three places and three transitions.

weight¹ of the arc from place p_i to transition t_j and b_{ij}^+ to denote the weight of the arc from transition t_j to place p_i . The graph shown in Figure 5-1 is an example of a Petri net with $d = 3$ and $u = 3$; its three places are denoted by p_1 , p_2 and p_3 , and its three transitions by t_1 , t_2 and t_3 (arcs with zero weight are not drawn).

Depending on the system modeled by the Petri net, input places can be interpreted as preconditions, input data/signals, resources, or buffers; transitions can be regarded as events, computation steps, tasks, or processors; output places can represent postconditions, output data/signals, conclusions, or buffers. Each place functions as a *token holder*. Tokens are drawn as black dots and represent resources that are available at different parts of the system. The number of tokens in a place *cannot* be negative. At any given time instant t , the *marking* (state) of the Petri net is given by the number of tokens at each of its places; for the Petri net in the figure the marking (at time instant 0) is given by $\mathbf{q}_s[0] = \begin{bmatrix} 2 & 1 & 0 \end{bmatrix}^T$.

Transitions model *events* that take place and cause the rearrangement, generation or disappearance of tokens. Transition t_j is *enabled* (i.e., it is allowed to take place) only if each of its input places p_i has at least b_{ij}^- tokens (where, as explained before, b_{ij}^- is the weight of the arc from place p_i to transition t_j). When transition t_j takes place (we say that transition t_j *fires*), it removes b_{ij}^- tokens from each input place p_i and adds b_{ij}^+ tokens to each output place p_i . In our example in Figure 5-1 transitions t_1 and t_2 are enabled but transition t_3 is not. If transition t_1 fires, it removes 2 tokens from its input place p_1 and

¹In this analysis we assume that there is only one arc from place p_i to transition t_j ; otherwise b_{ij}^- can be the aggregate weight of all such transitions.

adds 1 token each to its output places p_2 and p_3 ; the corresponding state of the Petri net (at the next time instant) will be $\mathbf{q}_s[1] = \begin{bmatrix} 0 & 2 & 1 \end{bmatrix}^T$.

If we define $\mathbf{B}^- = [b_{ij}^-]$ (respectively $\mathbf{B}^+ = [b_{ij}^+]$) to be the $d \times u$ matrix with b_{ij}^- (respectively b_{ij}^+) at its i th row, j th column position, the state evolution of a Petri net can be represented by the following equation:

$$\mathbf{q}_s[t+1] = \mathbf{q}_s[t] + (\mathbf{B}^+ - \mathbf{B}^-)\mathbf{x}[t] \quad (5.1)$$

$$= \mathbf{q}_s[t] + \mathbf{B}\mathbf{x}[t], \quad (5.2)$$

where $\mathbf{B} \equiv \mathbf{B}^+ - \mathbf{B}^-$ (in Figure 5-1 we show the corresponding \mathbf{B}^+ and \mathbf{B}^- for that Petri net). The input $\mathbf{x}[t]$ in the above description is n -dimensional and restricted to have exactly one non-zero entry with value “1”. When $\mathbf{x}[t] = \mathbf{x}_j = \begin{bmatrix} 0 & \dots & 1 & \dots & 0 \end{bmatrix}^T$ (the “1” being at the j th position), transition t_j fires (j is in $\{1, 2, \dots, u\}$). Note that transition t_j is enabled at time instant t if and only if $\mathbf{q}_s[t] \geq \mathbf{B}^-(:, j)$ (where $\mathbf{B}^-(:, j)$ denotes the j th column of \mathbf{B}^- and the inequality is taken element-wise). A *pure* Petri net is one in which no place serves as both an input and an output for the same transition (i.e., only one of b_{ij}^+ and b_{ij}^- can be non-zero). The Petri net in Figure 5-1 (with the indicated \mathbf{B}^+ and \mathbf{B}^- matrices) is a pure Petri net. Matrix \mathbf{B} has integer entries; its transpose is known as the *incidence matrix*, [73].

Discrete event systems are often modeled as Petri nets. In the following example we present the Petri net version of the popular “cat-and-mouse” problem, originally introduced by Ramadge and Wonham in the setting of supervisory control [86], and described as a Petri net in [114]. References [86, 114] were concerned with controlling the doors in the maze so that the two animals are never in the same room together². In Sections 5.4 and 5.5 we discuss how to perform failure detection and identification in such systems; in Section 5.6 we discuss how to detect ongoing “illegal” activity.

Example 5.1 We are given the maze of five rooms shown in Figure 5-2; a cat and a mouse circulate in this maze, with the cat moving from room to room through *unidirectional* doors $\{c_1, c_2, \dots, c_8\}$ and the mouse through unidirectional doors $\{m_1, m_2, \dots, m_6\}$. The Petri net

²Only a subset of the doors may be controllable; the task becomes challenging because we wish to allow maximum freedom in the movement of the two animals (while avoiding their entrance into the same room).

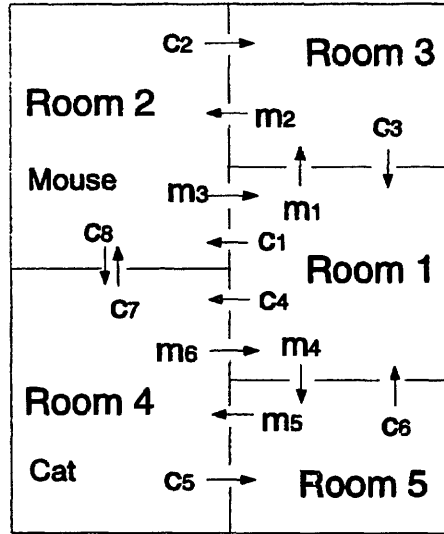


Figure 5-2: Cat-and-mouse maze.

model is based on two separate subnets, one dealing with the cat's position and movements and the other dealing with the mouse's position and movements. Each subnet has five places, corresponding to the five rooms in the maze. A token in a certain place indicates that the mouse (or the cat) is in the corresponding room. Transitions model the movements of the two animals between different rooms (as allowed by the structure of the maze in Figure 5-2). The subnet that deals with the mouse has a marking with five variables, exactly one of which has the value "1" (the rest are set to "0"). The state evolution for this subnet is given by eqs. (5.1) and (5.2) with

$$B^+ = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad B^- = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

For example, state $q_s[t] = [0 \ 1 \ 0 \ 0 \ 0]^T$ indicates that at time instant t the mouse is in room 2. Transition t_3 takes place when the mouse moves from room 2 to room 1 through door m_3 ; this causes the new state to be $q_s[t+1] = [1 \ 0 \ 0 \ 0 \ 0]^T$. In [114] the two subnets associated with the mouse and cat movements were combined in order to obtain

an overall Petri net, based on which was constructed a *linear* controller that achieves the desired objective (i.e., disallows the presence of the cat and the mouse in the same room while permitting maximum freedom in their movement within the maze). In Section 5.6 we use this Petri net representation in order to construct monitors that detect and identify “illegal” (or unmodeled) activity in the system. \square

5.3 Error Model

In complex Petri nets with a large number of places and transitions, there is a high possibility of system breakdown due to malfunctions of hardware, software or other components. It is therefore essential that one design systems with the ability to detect, locate and correct any failures that may occur. In this section we discuss the error models that will be used in our failure detection and identification schemes for Petri nets. As mentioned in Chapter 1, an effective error model needs to capture the underlying failure in an efficient manner. Clearly, this will depend significantly on the particular application and the actual implementation; this is the price we pay for the generality of the error model. Since failures depend on the actual implementation (which varies considerably depending on the application), we will consider three different error models.

- A *transition failure* models a failure that occurs while performing a certain transition. We say that transition t_j *fails to execute its postconditions* if no tokens are deposited to its output places (even though the correct number of tokens from the input places have been used). Similarly, we say that transition t_j *fails to execute its preconditions* if the tokens that are supposed to be removed from the input places of the faulty transition are not removed (even though the correct number of tokens are deposited at the corresponding output places). In terms of the state evolution in eq. (5.1), a failure at transition t_j corresponds to transition t_j firing, but its preconditions (given by the j th column of B^- , $B^-(:,j)$), or its postconditions (given by $B^+(:,j)$) not taking effect³.

³A failure in which *both* the preconditions and the postconditions are not executed is indistinguishable from the transition not taking place at all. The motivation for the transition failure error model came from [35], although the failures mentioned there are captured best by *place failures*.

- A *place failure* models a failure that corrupts the number of tokens in a *single* place of the Petri net. In terms of eq. (5.1), a place failure at time instant t causes the value of a single variable in the d -dimensional state $\mathbf{q}_s[t]$ to be incorrect. This error model is suitable for Petri nets that represent computational systems or finite-state machines (e.g., single-bit errors corrupt a single place in the Petri net); it has appeared in earlier work that dealt with failure-detection in pure Petri nets, [97, 98].
- The *additive failure model* is based on explicitly enumerating all failures that we would like to be able to detect or protect against. The error is then modeled by its *additive* effect on the state $\mathbf{q}_s[t]$ of the Petri net. In particular, if failure $f(i)$ takes place at time instant t , then $\mathbf{q}_{f(i)}[t] = \mathbf{q}_s[t] + \mathbf{e}_{f(i)}$ where $\mathbf{q}_{f(i)}[t]$ is the faulty state of the Petri net and $\mathbf{e}_{f(i)}$ is the *additive* effect of failure $f(i)$. If we can find *a priori* the additive effect $\mathbf{e}_{f(i)}$ for each failure $f(i)$ that we would like to protect against, then we can define a $d \times l$ *failure matrix* $E = \left[\begin{array}{c|c|c|c} \mathbf{e}_{f(1)} & \mathbf{e}_{f(2)} & \cdots & \mathbf{e}_{f(l)} \end{array} \right]$, where l is the total number of different failures expected. Based on the matrix E , we can construct redundant implementations to protect our Petri net.

Note that the additive error model captures both transition and place failures:

- If transition t_j fails to execute its preconditions, then $\mathbf{e}_{t_j}^- = \mathbf{B}^-(:, j)$, whereas if t_j fails to execute its postconditions, then $\mathbf{e}_{t_j}^+ = -\mathbf{B}^+(:, j)$.
- The corruption of the number of tokens in place p_i is captured by the additive error array $\mathbf{e}_{p_i} = c \times \left[\begin{array}{cccc} 0 & \cdots & 1 & \cdots & 0 \end{array} \right]^T$, where c is an integer that denotes the number of tokens that have been added and the only non-zero entry of the array $\left[\begin{array}{cccc} 0 & \cdots & 1 & \cdots & 0 \end{array} \right]^T$ appears at the i th position.

The additive error model can also capture the effects of multiple *independent* additive failures (i.e., failures whose additive effect does not depend on whether other failures have taken place or not). For example, a precondition failure at transition t_j and an *independent* failure at place p_i will result in the additive error array $\mathbf{e}_{t_j}^- + \mathbf{e}_{p_i}$.

Example 5.2 Consider the Petri net in Figure 5-3. It could be the model of a distributed processing network or of a flexible manufacturing system. Transition t_2 models a process

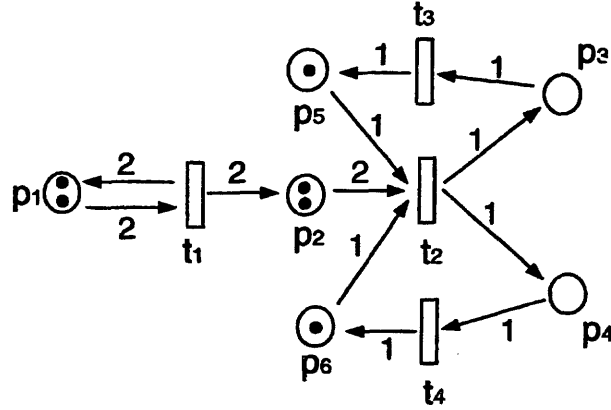


Figure 5-3: Petri net model of a distributed processing system.

that takes as input two data packets (or two raw products) from place p_2 and produces two different data packets (or intermediate products), each of which gets deposited to places p_3 and p_4 . Processes t_3 and t_4 take their corresponding input packets (from places p_3 and p_4 respectively) to produce the final data packets (or final products). Note that processes t_3 and t_4 can take effect concurrently. Once done, they return separate acknowledgments to places p_5 and p_6 so that process t_2 can be enabled again. Transition t_1 models the external input to the system and is always enabled. The state of the Petri net shown in Figure 5-3 is given by $\mathbf{q}_s[0] = \begin{bmatrix} 2 & 2 & 0 & 0 & 1 & 1 \end{bmatrix}^T$; only transitions t_1 and t_2 are enabled.

If the process modeled by transition t_2 fails to execute its postconditions, tokens will be removed from input places p_2 , p_5 and p_6 , but no tokens will be deposited at output places p_3 and p_4 . The faulty state of the Petri net will be $\mathbf{q}_f[1] = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$.

If process t_2 fails to execute its preconditions, then tokens will appear at the output places p_3 and p_4 but no tokens will be removed from the input places p_2 , p_5 and p_6 . The faulty state of the Petri net will be $\mathbf{q}_f[1] = \begin{bmatrix} 2 & 2 & 1 & 1 & 1 & 1 \end{bmatrix}^T$.

If process t_2 executes correctly but there is a failure at place p_4 , then the resulting state will be of the form $\mathbf{q}_f[1] = \begin{bmatrix} 2 & 0 & 1 & 1+c & 0 & 0 \end{bmatrix}^T$; the number of tokens at place p_4 has been corrupted by c . \square

Example 5.3 The Petri net in Figure 5-4 denotes a simple digital system. Places correspond to memory locations that hold a single bit with value "0" or "1". In this particular

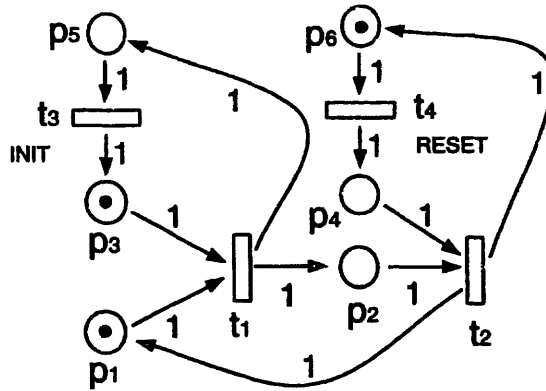


Figure 5-4: Petri net model of a digital system.

example, p_1 and p_2 represent the two states of the system⁴, places p_3 and p_4 are placeholders for the INIT and RESET inputs, and places p_5 and p_6 ensure that these inputs are only applied when appropriate. Transitions correspond to logic gates that produce the next state of the system based on the current state and input. More specifically, each transition produces two outputs: the first output has value “1” and activates the necessary memory locations for the next state; the second output has value “0” and deactivates the memory locations associated with the previous state (and input). We assume that the output of the system is based on its state (p_1 or p_2) and we do not show it here.

A failure in a particular memory location causes a “0” to become “1” or vice-versa. The effect can be represented by a place failure. A gate failure at a transition is more complicated and may need to be represented by multiple place failures or a combination of transition and place failures. □

5.4 Monitoring Schemes Using Separate Redundant Implementations

5.4.1 Separate Redundant Petri Net Implementations

We begin our study of *redundant Petri net implementations* with a special case in which the original Petri net remains unchanged. Carefully chosen additional functionality allows

⁴The system is in state p_i when there is a token in place p_i . The approach is reminiscent of *one-hot coding* where each state is represented by its own memory location.

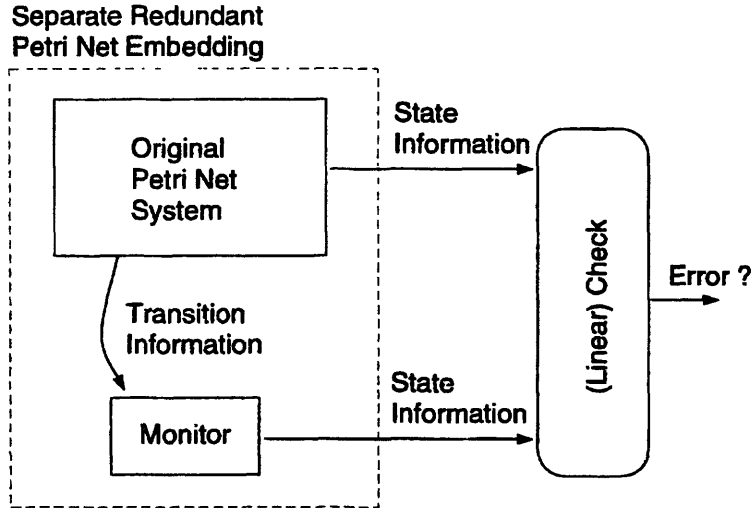


Figure 5-5: Concurrent monitoring scheme for a Petri net.

us to build a monitoring scheme as shown in Figure 5-5: by performing a check on the combined state of the original system and the added *monitor*, the detecting mechanism is able to locate and identify failures in the overall system. The state of the monitor is changed according to the transition activity in the original system, and captures exactly the information that is necessary in order to concurrently detect and identify failures.

Our monitoring schemes are constructed using linear checks. This makes them easily adaptable to changes in the configuration or in the initial marking of the given Petri net; furthermore, our schemes can be applied to systems where certain information is unavailable (e.g., when *no* connections are available from a particular place or transition). We give examples of how these ideas could be pursued but leave a detailed study to future work. The alternative to our construction could be an analysis that is based on locating an invalid state and inferring the exact failure that caused the Petri net to reach this invalid state. Our approach avoids this complicated reachability analysis⁵, automatically points out which additional connections are necessary and results in simple monitors.

Definition 5.1 A separate redundant implementation of Petri net \mathcal{S} (with d places, u transitions, marking $\mathbf{q}_s[\cdot]$ and state evolution as in eq. (5.1)) is a Petri net \mathcal{H} (with $\eta \equiv d+u$

⁵A nice discussion on state reachability in Petri nets can be found in [91].

places, $s > 0$, and u transitions) that has state evolution

$$\begin{aligned} \mathbf{q}_h[t+1] &= \mathbf{q}_h[t] + \mathbf{B}^+ \mathbf{x}[t] - \mathbf{B}^- \mathbf{x}[t] \\ &= \mathbf{q}_h[t] + \begin{bmatrix} \mathbf{B}^+ \\ \mathbf{X}^+ \end{bmatrix} \mathbf{x}[t] - \begin{bmatrix} \mathbf{B}^- \\ \mathbf{X}^- \end{bmatrix} \mathbf{x}[t], \end{aligned} \quad (5.3)$$

and whose state is given by⁶

$$\mathbf{q}_h[t] = \underbrace{\begin{bmatrix} \mathbf{I}_d \\ \mathbf{C} \end{bmatrix}}_{\mathbf{G}} \mathbf{q}_s[t],$$

for all time instants t . We require that for any initial marking (state) $\mathbf{q}_s[0]$ for \mathcal{S} , Petri net \mathcal{H} (with initial state $\mathbf{q}_h[0] = \mathbf{G}\mathbf{q}_s[0]$) admits all firing transition sequences that are allowed in \mathcal{S} (under initial state $\mathbf{q}_s[0]$).

Note that the functionality of Petri net \mathcal{S} remains intact within the separate redundant Petri net implementation \mathcal{H} . Matrix \mathbf{G} is referred to as the *encoding* matrix. All valid states $\mathbf{q}_h[t]$ in \mathcal{H} have to lie within the column space of \mathbf{G} ; furthermore, there exists a parity check matrix $\mathbf{P} = \begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix}$ such that $\mathbf{P}\mathbf{q}_h[t] = \mathbf{0}$ for all t (under fault-free conditions). Since \mathcal{H} is a Petri net, matrices \mathbf{X}^+ and \mathbf{X}^- , and state $\mathbf{q}_h[t]$ (for all t) have nonnegative integer entries. The following theorem characterizes separate redundant Petri net implementations and leads to systematic ways of constructing them:

Theorem 5.1 *Consider the setting described above. Petri net \mathcal{H} is a separate redundant implementation of Petri net \mathcal{S} if and only if \mathbf{C} is a matrix with nonnegative integer entries and*

$$\mathbf{X}^+ = \mathbf{C}\mathbf{B}^+ - \mathbf{D}, \quad \mathbf{X}^- = \mathbf{C}\mathbf{B}^- - \mathbf{D},$$

where \mathbf{D} is any $s \times u$ matrix with nonnegative integer entries such that $\mathbf{D} \leq \min(\mathbf{C}\mathbf{B}^+, \mathbf{C}\mathbf{B}^-)$ (operations \leq and \min are taken element-wise).

⁶More generally the state of a separate redundant Petri net implementation can be given by $\mathbf{q}_h[t] = \begin{bmatrix} \mathbf{q}_s[t] \\ \zeta(\mathbf{q}_s[t]) \end{bmatrix}$ for an appropriate function ζ .

Proof: (\Rightarrow) The state $\mathbf{q}_h[0] = \mathbf{G}\mathbf{q}_s[0] = \begin{bmatrix} \mathbf{I}_d \\ \mathbf{C} \end{bmatrix} \mathbf{q}_s[0]$ needs to have nonnegative integer entries for all valid $\mathbf{q}_s[0]$ (a valid initial state for \mathcal{S} is any marking $\mathbf{q}_s[0]$ with nonnegative integer entries). Clearly, a necessary (and sufficient) condition is that \mathbf{C} is a matrix with nonnegative integer entries.

If we combine the state evolution of the redundant and original Petri nets (in eqs. (5.3) and (5.1) respectively), we see that

$$\begin{aligned} \mathbf{G}\mathbf{q}_s[t+1] \equiv \mathbf{q}_h[t+1] &= \mathbf{q}_h[t] + \begin{bmatrix} \mathbf{B}^+ \\ \mathbf{X}^+ \end{bmatrix} \mathbf{x}[t] - \begin{bmatrix} \mathbf{B}^- \\ \mathbf{X}^- \end{bmatrix} \mathbf{x}[t] \\ &\Rightarrow \\ \begin{bmatrix} \mathbf{I}_d \\ \mathbf{C} \end{bmatrix} \mathbf{q}_s[t+1] &= \begin{bmatrix} \mathbf{I}_d \\ \mathbf{C} \end{bmatrix} \mathbf{q}_s[t] + \begin{bmatrix} \mathbf{B}^+ \\ \mathbf{X}^+ \end{bmatrix} \mathbf{x}[t] - \begin{bmatrix} \mathbf{B}^- \\ \mathbf{X}^- \end{bmatrix} \mathbf{x}[t]. \end{aligned}$$

Since any transition t_j can be enabled (e.g., by choosing $\mathbf{q}_s[0] \geq \mathbf{B}^-(:, j)$), we conclude that

$$\mathbf{X}^+ - \mathbf{X}^- = \mathbf{C}(\mathbf{B}^+ - \mathbf{B}^-).$$

Without loss of generality we can set $\mathbf{X}^+ = \mathbf{C}\mathbf{B}^+ - \mathbf{D}$ and $\mathbf{X}^- = \mathbf{C}\mathbf{B}^- - \mathbf{D}$ for some matrix \mathbf{D} with integer entries. In order for Petri net \mathcal{H} with initial marking $\mathbf{q}_h[0] = \begin{bmatrix} \mathbf{q}_s^T[0] & (\mathbf{C}\mathbf{q}_s[0])^T \end{bmatrix}^T$ (where $\mathbf{q}_s[0]$ is *any* initial state for \mathcal{S}) to admit all firing transition sequences that are allowed in \mathcal{S} under initial state $\mathbf{q}_s[0]$, we need \mathbf{D} to have nonnegative integer entries. The proof follows easily by contradiction: suppose \mathbf{D} has a negative entry in its j th column; choose $\mathbf{q}_s[0] = \mathbf{B}^-(:, j)$; then transition t_j can be fired in \mathcal{S} but cannot be fired in \mathcal{H} because

$$\begin{aligned} \mathbf{C}\mathbf{q}_s[0] &= \mathbf{C}\mathbf{B}^-(:, j) \\ &< \mathbf{C}\mathbf{B}^-(:, j) - \mathbf{D}(:, j) \\ &= \mathbf{X}^-(:, j). \end{aligned}$$

The requirement that $\mathbf{D} \leq \min(\mathbf{C}\mathbf{B}^+, \mathbf{C}\mathbf{B}^-)$ follows from \mathbf{X}^+ and \mathbf{X}^- being matrices with nonnegative integer entries.

(\Leftarrow) The converse direction follows easily. The only challenge is to show that if \mathbf{D} is chosen to have nonnegative entries, all transitions that are enabled in \mathcal{S} at time instant t under state $\mathbf{q}_s[t]$ are also enabled in \mathcal{H} under state $\mathbf{q}_h[t] = \begin{bmatrix} \mathbf{q}_s^T[t] & (\mathbf{C}\mathbf{q}_s[t])^T \end{bmatrix}^T$. For this, note that if \mathbf{D} has nonnegative entries, then

$$\begin{aligned} \mathbf{q}_s[t] \geq \mathbf{B}^-(:,j) &\Rightarrow \mathbf{G}\mathbf{q}_s[t] \geq \mathbf{G}\mathbf{B}^-(:,j) \\ &\Rightarrow \mathbf{q}_h[t] \geq \mathbf{G}\mathbf{B}^-(:,j) \\ &\Rightarrow \mathbf{q}_h[t] \geq \mathbf{G}\mathbf{B}^-(:,j) - \begin{bmatrix} \mathbf{0} \\ \mathbf{D}(:,j) \end{bmatrix} \\ &\Rightarrow \mathbf{q}_h[t] \geq \mathbf{B}^-(:,j). \end{aligned}$$

(Remember that matrices \mathbf{G} , \mathbf{B}^+ , \mathbf{B}^- and \mathbf{D} have nonnegative integer entries.) We conclude that if transition t_j is enabled in \mathcal{S} ($\mathbf{q}_s[t] \geq \mathbf{B}^-(:,j)$), then it is also enabled in \mathcal{H} ($\mathbf{q}_h[t] \geq \mathbf{B}^-(:,j)$). \square

5.4.2 Failure Detection and Identification

Given a Petri net \mathcal{S} with state evolution as in eq. (5.1), we can use a separate redundant implementation \mathcal{H} as described in Theorem 5.1 to monitor transition and place failures. The invariant conditions imposed by the construction of our separate implementations can be checked by verifying that $\begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix} \mathbf{q}_h[t]$ is equal to $\mathbf{0}$. The s additional places in \mathcal{H} function as *checkpoint places* and can either be distributed in the Petri net system or be part of a centralized monitor⁷.

Transition Failures: Suppose that at time instant $t-1$ transition t_j fires (that is, $\mathbf{x}[t-1] = \mathbf{x}_j$). If, due to a failure, the *postconditions* of transition t_j are not executed, the erroneous state at time instant t will be

$$\mathbf{q}_f[t] = \mathbf{q}_h[t] - \mathbf{B}^+(:,j) = \mathbf{q}_h[t] - \mathbf{B}^+\mathbf{x}_j$$

(where $\mathbf{q}_h[t]$ is the state that would have been reached under fault-free conditions). The

⁷Some of the constraints that we developed in the previous section can be dropped if we adopt the view in [98], and treat additional places only as *test places*, i.e., allow them to have a negative number of tokens. In such case, \mathbf{C} and \mathbf{D} are not restricted to have nonnegative entries.

error syndrome will be

$$\begin{aligned}
Pq_f[t] &= P(q_h[t] - \begin{bmatrix} B^+ \\ CB^+ - D \end{bmatrix} x_j) \\
&= Pq_h[t] - P \begin{bmatrix} B^+ \\ CB^+ - D \end{bmatrix} x_j \\
&= \mathbf{0} - \begin{bmatrix} -C & I_s \end{bmatrix} \begin{bmatrix} B^+ \\ CB^+ - D \end{bmatrix} x_j \\
&= -(-CB^+ + CB^+ - D) x_j \\
&= Dx_j \equiv D(:, j) .
\end{aligned}$$

If the *preconditions* of transition t_j are not executed, the erroneous state will be

$$q_f[t] = q_h[t] + B^-(:, j) = q_h[t] + B^- x_j ;$$

the error syndrome can be calculated similarly as

$$Pq_f[t] = -Dx_j \equiv -D(:, j) .$$

If we choose all columns of D to be distinct, we will be able to detect and identify all single-transition failures. Depending on the sign, we can also describe whether preconditions or postconditions were not executed. In fact, given enough redundancy we will be able to identify multiple transition failures (for example, we can ensure that the columns of D linearly independent).

Example 5.4 Consider the Petri net in Figure 5-1 with the indicated B^+ and B^- matrices. We will use one additional place ($s = 1$); in order to concurrently detect and identify transition failures, we need to ensure that the columns of matrix D are distinct (the choice of C does not matter in this case). We set $D = \begin{bmatrix} 3 & 2 & 1 \end{bmatrix}$, $C = \begin{bmatrix} 2 & 2 & 1 \end{bmatrix}$ and obtain the separate redundant Petri net implementation of Figure 5-6 (the additional connections are shown with dotted lines).

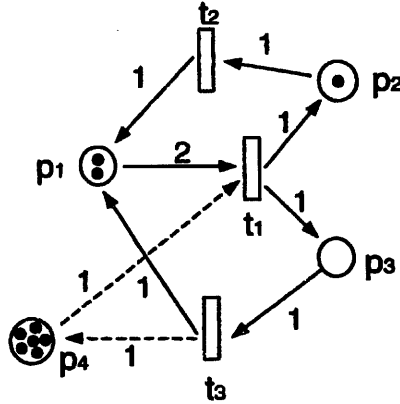


Figure 5-6: Example of a separate redundant Petri net implementation that identifies single-transition failures in the Petri net of Figure 5-1.

Matrices B^+ and B^- are given by

$$B^+ = \left[\frac{B^+}{CB^+ - D} \right] = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B^- = \left[\frac{B^-}{CB^- - D} \right] = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

The parity check, performed concurrently by the checking mechanism, is given by

$$\left[-C \mid I_1 \right] \mathbf{q}_h[t] = \left[-2 \quad -2 \quad -1 \mid 1 \right] \mathbf{q}_h[t].$$

If the parity check is -3 (respectively -2 , -1), then transition t_1 (respectively t_2 , t_3) has failed to perform its preconditions. If the parity check is 3 (respectively 2 , 1), then transition t_1 (respectively t_2 , t_3) has failed to perform its postconditions.

The additional place p_4 is part of the monitoring mechanism: it receives information about the activity in the original Petri net (which transitions fire or complete, etc.) and appropriately updates its tokens. The linear checker (not shown in Figure 5-6) detects and identifies failures by evaluating a checksum on the state of the overall (redundant) system. Note that the monitoring mechanism in Figure 5-6 does not use any information about transition t_2 . More generally, explicit connections from each transition to the monitoring mechanism may not be required; in fact, an interesting direction to pursue in future work is

to investigate whether this monitoring scheme can be systematically adapted to handle cases where certain connections are not permitted (i.e., where information about a transition or a place is not available). \square

Place Failures: If, due to a failure, the number of tokens in place p_i is increased by c , the faulty state is given by

$$\mathbf{q}_f[t] = \mathbf{q}_h[t] + \mathbf{e}_{p_i}$$

where \mathbf{e}_{p_i} is an η -dimensional array with a unique non-zero entry at its i th position, i.e., $\mathbf{e}_{p_i} = c \times \begin{bmatrix} 0 & \dots & 1 & \dots & 0 \end{bmatrix}^T$ (with the "1" at the i th position). In this case, the parity check will be

$$\begin{aligned} \mathbf{P}\mathbf{q}_f[t] &= \mathbf{P}(\mathbf{q}_h[t] + \mathbf{e}_{p_i}) \\ &= \mathbf{P}\mathbf{q}_h[t] + \mathbf{P}\mathbf{e}_{p_i} \\ &= \mathbf{0} + \mathbf{P}\mathbf{e}_{p_i} \\ &= c \times \mathbf{P}(:, i) . \end{aligned}$$

If we choose \mathbf{C} so that columns of $\mathbf{P} \equiv \begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix}$ are not rational multiples of each other, then we can detect and identify single-place failures⁸.

Example 5.5 In order to concurrently detect and identify single-place failures in the Petri net of Figure 5-1, we will use two additional places ($s = 2$) and choose $\mathbf{C} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 1 \end{bmatrix}$ (so that the columns of the parity check matrix $\mathbf{P} = \begin{bmatrix} -\mathbf{C} & \mathbf{I}_2 \end{bmatrix}$ are not multiples of each other). Our choice for \mathbf{D} is not critical in the identification of place failures and in this case we set⁹ $\mathbf{D} = \begin{bmatrix} 2 & 1 & 1 \\ 2 & 1 & 1 \end{bmatrix}$. We then obtain the separate redundant implementation shown in Figure 5-7 (the additional connections are shown with dotted lines).

⁸We need to make sure that for all pairs of columns of \mathbf{P} there do not exist non-zero integers α, β such that $\alpha \times \mathbf{P}(:, i) = \beta \times \mathbf{P}(:, j)$, $i \neq j$.

⁹This choice actually minimizes the number of additional connections (for the given choice of \mathbf{C}).

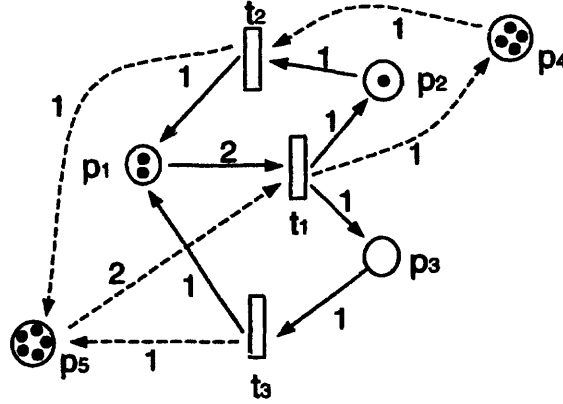


Figure 5-7: Example of a separate redundant Petri net implementation that identifies single-place failures in the Petri net of Figure 5-1.

Matrices B^+ and B^- are given by

$$B^+ = \left[\frac{B^+}{CB^+ - D} \right] = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \quad B^- = \left[\frac{B^-}{CB^- - D} \right] = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 2 & 0 & 0 \end{bmatrix}.$$

The parity check is performed through

$$\left[-C \mid I_2 \right] \mathbf{q}_h[t] = \begin{bmatrix} -1 & -2 & -1 \\ -2 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{q}_h[t].$$

If the result is a multiple of $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ (respectively $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$), then place p_1 (respectively p_2 , p_3 , p_4 , p_5) has failed.

In a monitoring scheme, the additional places (p_4 and p_5 in this example) may be parts of a centralized monitoring mechanism, so it may be possible in certain cases to assume that they are fault-free. Just as in Example 5.4, the checking mechanism (not shown in Figure 5-7) detects and identifies place failures by evaluating a checksum on the state of the overall Petri net. \square

If \mathbf{C} and \mathbf{D} are chosen properly, we can actually perform detection and identification of both place and transition failures. Note that matrices \mathbf{C} and \mathbf{D} can be chosen almost independently (subject to the constraints analyzed above¹⁰). The following example illustrates how this can be done.

Example 5.6 Identification of a single-transition failure or a single-place failure (but not of both occurring together) can be achieved in the Petri net of Figure 5-1 using two additional places ($s = 2$). Let $\mathbf{C} = \begin{bmatrix} 3 & 2 & 3 \\ 2 & 3 & 3 \end{bmatrix}$ and $\mathbf{D} = \begin{bmatrix} 5 & 2 & 3 \\ 4 & 1 & 1 \end{bmatrix}$. With these choices, matrices \mathcal{B}^+ and \mathcal{B}^- are given by

$$\mathcal{B}^+ = \left[\frac{\mathbf{B}^+}{\mathbf{CB}^+ - \mathbf{D}} \right] = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix}, \quad \mathcal{B}^- = \left[\frac{\mathbf{B}^-}{\mathbf{CB}^- - \mathbf{D}} \right] = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 2 & 2 \end{bmatrix}.$$

The parity check is performed through

$$\left[-\mathbf{C} \mid \mathbf{I}_2 \right] \mathbf{q}_h[t] = \begin{bmatrix} -3 & -2 & -3 & 1 & 0 \\ -2 & -3 & -3 & 0 & 1 \end{bmatrix} \mathbf{q}_h[t].$$

If the parity check is a multiple of $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$ (respectively $\begin{bmatrix} 2 \\ 3 \end{bmatrix}$, $\begin{bmatrix} 3 \\ 3 \end{bmatrix}$, $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$), then

there is a place failure in p_1 (respectively p_2 , p_3 , p_4 , p_5). If the parity check is $\begin{bmatrix} 5 \\ 4 \end{bmatrix}$

(respectively $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $\begin{bmatrix} 3 \\ 1 \end{bmatrix}$), then transition t_1 (respectively t_2 , t_3) has failed to perform

its postconditions. If the parity check is $\begin{bmatrix} -5 \\ -4 \end{bmatrix}$ (respectively $\begin{bmatrix} -2 \\ -1 \end{bmatrix}$, $\begin{bmatrix} -3 \\ -1 \end{bmatrix}$), then transition t_1 (respectively t_2 , t_3) has failed to perform its preconditions.

¹⁰Matrix \mathbf{D} has to satisfy $\mathbf{D} \leq \min(\mathbf{CB}^+, \mathbf{CB}^-)$, but we can always multiply matrix \mathbf{C} by an integer constant to increase the possibilities for entries in \mathbf{D} .

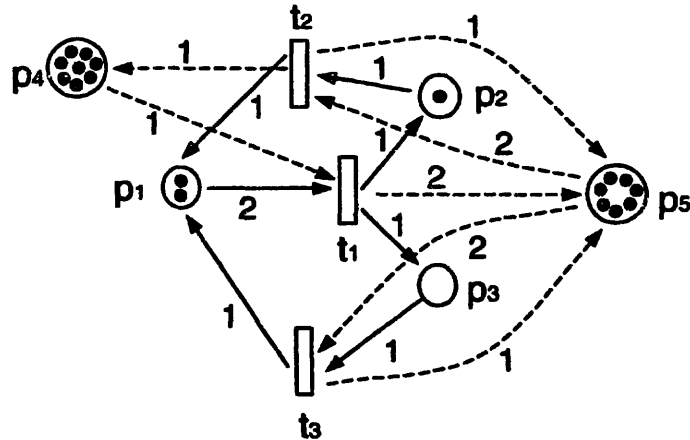


Figure 5-8: Example of a separate redundant Petri net implementation that identifies single-transition or single-place failures in the Petri net of Figure 5-1.

The resulting redundant Petri net implementation is shown in Figure 5-8 (the additional connections are shown with dotted lines; the linear checker is not shown in the figure). \square

The graphical interpretation of the monitoring scheme in the above examples is straightforward: we add s places and connect them to the transitions of the original Petri net. The added places could be part of a centralized controller or could be distributed in the system. The tokens associated with the additional connections and places can be regarded as simple acknowledgment messages. The weights of the additional connections are given by the matrices $CB^+ - D$ and $CB^- - D$. The choice of matrix C specifies detection and identification for place failures, whereas the choice of D determines detection and identification for transition failures. Coding techniques or simple linear algebra can be used to guide the choice of C or D . To detect single-place failures, we need to ensure that the columns of matrix C are not multiples of each other (this is what guided our choice of C in Examples 5.5 and 5.6). Similarly, matrices D in Examples 5.4 and 5.6 were chosen so that their columns are not the same (they are allowed to be multiples of each other).

The above discussion clearly demonstrates that there are many choices for matrices C and D for given failure detection and identification requirements. One interesting future research direction is to develop criteria for choosing among these various possibilities. Depending on the underlying system, plausible objectives could be to minimize the size of the monitor (number of additional places), the number of additional connections (from the

original system to the additional places), and/or the number of tokens involved. Once these criteria are well-understood, it would be beneficial to develop algorithmic techniques and automatic tools that allow us to systematically choose \mathbf{C} and \mathbf{D} so as to satisfy any or all of these criteria.

The additional places in our monitoring schemes (e.g., places p_4 and p_5 in Examples 5.5 and 5.6) may be parts of a centralized monitoring mechanism, so it may be reasonable in certain cases to assume that they are fault-free. Nevertheless, our scheme is capable of handling failures in any of the places in the Petri net embedding, including the added ones. The checking mechanism (not shown in any of the figures in Examples 5.4, 5.5 and 5.6) detects and identifies place failures by evaluating a checksum on the state of the overall Petri net embedding. Our implicit assumption has been that no failure takes place during this checksum calculation; it would be interesting to investigate ways to relax this assumption.

Note that, if we restrict ourselves to *pure* Petri nets, then we do not have a choice for \mathbf{D} . More specifically, we need to ensure that the resulting Petri net is pure, which means that $\mathbf{D} = \min(\mathbf{CB}^+, \mathbf{CB}^-)$. In such cases we may lose the ability to detect transition failures (we may attempt to treat them as *multiple* place failures). In this restricted case we recover the results in [97, 98]: given a pure Petri net S as in eq. (5.2), we can construct a *pure* Petri net embedding with state evolution

$$\mathbf{q}_h[t+1] = \mathbf{q}_h[t] + \begin{bmatrix} \mathbf{B} \\ \mathbf{CB} \end{bmatrix} \mathbf{x}[t]$$

for a matrix \mathbf{C} with nonnegative integer entries.

The distance measure adopted in [97] suggests that the redundant Petri net should guard against place failures (corruption of the number of tokens in individual places). The examples in [97] include a discussion of codes in finite fields and Petri nets in which addition is performed modulo some integer. (For instance, modulo-2 addition matches well with Petri net systems in which places are implemented using binary memory elements. In this case, by choosing $\mathbf{P} \equiv \begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix}$ to be the (systematic) parity check matrix of a Hamming code [111], one can achieve single-place failure detection and identification.) Our approach is more general than in [97, 98], and is well-suited to failure detection and identification schemes for DES's.

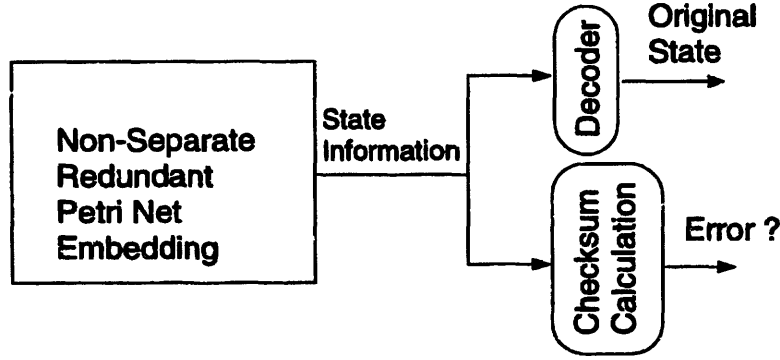


Figure 5-9: Concurrent monitoring using a non-separate Petri net implementation.

5.5 Monitoring Schemes Using Non-Separate Redundant Implementations

5.5.1 Non-Separate Redundant Petri Net Implementations

In this section we characterize non-separate redundant Petri net implementations and use this characterization to systematically construct more general monitoring schemes as shown in Figure 5-9: the state of the redundant Petri net implementation is encoded in a form that allows us to recover the state of the original Petri net *and* perform failure detection and identification.

Let S be a Petri net with d places, u transitions and state evolution as given in eqs. (5.1) and (5.2); let $\mathbf{q}_s[0]$ be *any* initial state $\mathbf{q}_s[0] \geq \mathbf{0}$ and $\mathcal{X} = \{\mathbf{x}[0], \mathbf{x}[1], \dots\}$ be *any* admissible (*legal*) firing sequence under this initial state.

Definition 5.2 Let \mathcal{H} be a Petri net with $\eta \equiv d + s$ places, $u + v$ transitions (where s and v are positive integers), initial state $\mathbf{q}_h[0]$ and state evolution equation

$$\begin{aligned} \mathbf{q}_h[t+1] &= \mathbf{q}_h[t] + \mathbf{B}^+ \mathbf{z}[t] - \mathbf{B}^- \mathbf{z}[t] \\ &= \mathbf{q}_h[t] + (\mathbf{B}^+ - \mathbf{B}^-) \mathbf{z}[t]. \end{aligned} \tag{5.4}$$

Petri net \mathcal{H} is a redundant implementation for S if it concurrently simulates S in the following sense: there exist

1. *an input encoding mapping $\xi : \mathbf{x}[t] \mapsto \mathbf{z}[t]$,*

2. a state decoding mapping $\ell : \mathbf{q}_h[t] \mapsto \mathbf{q}_s[t]$, and

3. a state encoding mapping $g : \mathbf{q}_s[t] \mapsto \mathbf{q}_h[t]$,

such that for any initial state $\mathbf{q}_s[0]$ in \mathcal{S} and any admissible firing sequence \mathcal{X} (for $\mathbf{q}_s[0]$), and for $\mathbf{z}[t] = \xi(\mathbf{x}[t])$, $\mathbf{q}_s[t] = \ell(\mathbf{q}_h[t])$ and $\mathbf{q}_h[t] = g(\mathbf{q}_s[t])$ for all time instants $t \geq 0$.

As defined above, a redundant implementation \mathcal{H} is a Petri net that after proper initialization (i.e., $\mathbf{q}_h[0] = g(\mathbf{q}_s[0])$) has the ability to *admit* any firing sequence \mathcal{X} that is admissible by the original Petri net \mathcal{S} (under initial state $\mathbf{q}_s[0]$). The state of the original Petri net at time instant t is specified by the state of the redundant implementation and vice-versa (through mappings ℓ and g). Note that, regardless of the initial state $\mathbf{q}_s[0]$ and the firing sequence \mathcal{X} , the state $\mathbf{q}_h[t]$ of the redundant implementation always lies in a subset of the redundant state space (namely the image of $\mathbf{q}_s[\cdot]$ under the mapping g). The v additional transitions represent new activity that can be used for failure-recovery.

For the rest of this section we focus on a special class of non-separate redundant implementations, where encoding and decoding can be performed through appropriate encoding and decoding matrices. Specifically, we consider the case where there exists a $d \times \eta$ decoding matrix \mathbf{L} and an $\eta \times d$ encoding matrix \mathbf{G} such that, under any initial state $\mathbf{q}_s[0]$ and *any* admissible sequence of inputs $\mathcal{X} = \{\mathbf{x}[0], \mathbf{x}[1], \dots\}$,

$$\mathbf{q}_s[t] = \mathbf{L}\mathbf{q}_h[t], \text{ and}$$

$$\mathbf{q}_h[t] = \mathbf{G}\mathbf{q}_s[t]$$

for all time instants $t \geq 0$. Furthermore, we will assume that $v = 0$ and (without loss of generality) treat ξ as the identity mapping (we can always permute the columns of \mathbf{B}^+ and \mathbf{B}^- in eq. (5.1)). The state evolution equation of a non-separate redundant Petri net implementation is then given by

$$\mathbf{q}_h[t+1] = \mathbf{q}_h[t] + \mathbf{B}^+\mathbf{x}[t] - \mathbf{B}^-\mathbf{x}[t] \quad (5.5)$$

$$= \mathbf{q}_h[t] + \mathbf{B}\mathbf{x}[t], \quad (5.6)$$

where $\mathbf{B} \equiv \mathbf{B}^+ - \mathbf{B}^-$.

The additional structure that is enforced through a redundant Petri net implementation of the form described above can be used for failure detection and identification. In order to systematically construct redundant implementations, we need to provide a common starting point. The following theorem characterizes redundant Petri net implementations in terms of a similarity transformation and a standard redundant Petri net.

Theorem 5.2 *A Petri net \mathcal{H} with $\eta \equiv d + s$ places, u transitions and state evolution as in eqs. (5.5) and (5.6) is a redundant Petri net implementation for \mathcal{S} (with state evolution as in eqs. (5.1) and (5.2)) only if it is similar (in the usual sense of change of basis in the state space, see Chapter 3) to a standard redundant Petri net implementation \mathcal{H}_σ whose state evolution equation is given by*

$$\begin{aligned} \mathbf{q}_\sigma[t+1] &= \mathbf{q}_\sigma[t] + \begin{bmatrix} \mathbf{B}^+ - \mathbf{B}^- \\ \mathbf{0} \end{bmatrix} \mathbf{x}[t] \\ &= \mathbf{q}_\sigma[t] + \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix} \mathbf{x}[t]. \end{aligned} \quad (5.7)$$

Here, \mathbf{B}^+ , \mathbf{B}^- and $\mathbf{B} \equiv \mathbf{B}^+ - \mathbf{B}^-$ are the matrices in eqs. (5.1) and (5.2). Associated with the standard redundant Petri net implementation is the standard decoding matrix $\mathbf{L}_\sigma = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \end{bmatrix}$ and the standard encoding matrix $\mathbf{G}_\sigma = \begin{bmatrix} \mathbf{I}_d \\ \mathbf{0} \end{bmatrix}$. Note that the standard redundant Petri net implementation is a pure Petri net.

Proof: Clearly, $\mathbf{L}\mathbf{G}\mathbf{q}_s[\cdot] = \mathbf{L}\mathbf{q}_h[\cdot] = \mathbf{q}_s[\cdot]$. Since all initial states are possible in \mathcal{S} ($\mathbf{q}_s[0]$ can be any array with nonnegative integer entries), we conclude that $\mathbf{L}\mathbf{G} = \mathbf{I}_d$. In particular, \mathbf{L} is full-row rank, \mathbf{G} is full-column rank and there exists an $\eta \times \eta$ matrix \mathcal{T} such that $\mathbf{L}\mathcal{T} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \end{bmatrix}$ and $\mathcal{T}^{-1}\mathbf{G} = \begin{bmatrix} \mathbf{I}_d \\ \mathbf{0} \end{bmatrix}$. By employing the similarity transformation $\mathbf{q}'_h[t] = \mathcal{T}\mathbf{q}_h[t]$, we obtain a similar system \mathcal{H}' whose state evolution is given by

$$\begin{aligned} \mathbf{q}'_h[t+1] &= (\mathcal{T}^{-1}\mathbf{I}_\eta\mathcal{T})\mathbf{q}'_h[t] + (\mathcal{T}^{-1}\mathbf{B})\mathbf{x}[t] \\ &\equiv \mathbf{q}'_h[t] + \mathbf{B}'\mathbf{x}[t], \end{aligned}$$

and has decoding and encoding matrices $L' = LT = \begin{bmatrix} I_d & 0 \end{bmatrix}$, and $G' = T^{-1}G = \begin{bmatrix} I_d \\ 0 \end{bmatrix}$.

For all time instants t , $q'_h[t] = G'q_s[t] = \begin{bmatrix} q_s[t] \\ 0 \end{bmatrix}$; by combining the state evolution equations of the original Petri net and the redundant system, we see that

$$\begin{aligned} q'_h[t+1] &= q'_h[t] + B'x[t] \\ &\Rightarrow \\ \begin{bmatrix} q_s[t] + Bx[t] \\ 0 \end{bmatrix} &= \begin{bmatrix} q_s[t] \\ 0 \end{bmatrix} + \begin{bmatrix} B'_1 \\ B'_2 \end{bmatrix} x[t]. \end{aligned}$$

The above equations hold for all initial conditions $q_s[0]$; since all transitions are enabled under some initial condition $q_s[0]$, we see that $B'_1 = B$ and $B'_2 = 0$.

If we regard the system \mathcal{H}' as a pure Petri net, we see that any transition enabled in \mathcal{S} is also enabled in \mathcal{H}' . Therefore, \mathcal{H}' is a redundant Petri net implementation. In fact, it is the standard redundant Petri net implementation \mathcal{H}_σ with the decoding and encoding matrices presented in the theorem. \square

Theorem 5.2 provides a characterization of the class of redundant Petri net implementations for the given Petri net \mathcal{S} and is a convenient starting point for systematically constructing such implementations. The invariant conditions that are imposed by the added redundancy on the standard Petri net \mathcal{H}_σ are easily identified: they are summarized by the *parity check* $P_\sigma q_\sigma[\cdot]$, where $P_\sigma = \begin{bmatrix} 0 & I_s \end{bmatrix}$ is the parity check matrix.

We now produce the converse to Theorem 5.2, which leads to the systematic construction of redundant Petri net implementations.

Theorem 5.3 *Let \mathcal{S} be a Petri net with d places, u transitions and state evolution as given in eqs. (5.1) and (5.2). A Petri net \mathcal{H} with $\eta \equiv d+s$ places, u transitions and state evolution as in eqs. (5.5) and (5.6) is a redundant Petri net implementation of \mathcal{S} if:*

- *It is similar to a standard redundant Petri net implementation \mathcal{H}_σ (with state evolution equation as in (5.7)) through an $\eta \times \eta$ invertible matrix T , whose first d columns consist of nonnegative integer entries. (The encoding, decoding and parity check matrices of*

the Petri net implementation \mathcal{H} are then given by $\mathbf{L} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \end{bmatrix} \mathcal{T}$, $\mathbf{G} = \mathcal{T}^{-1} \begin{bmatrix} \mathbf{I}_d \\ \mathbf{0} \end{bmatrix}$

and $\mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{I}_s \end{bmatrix} \mathcal{T}$.)

- Matrices \mathbf{B}^+ and \mathbf{B}^- are given by

$$\begin{aligned} \mathbf{B}^+ &= \mathcal{T}^{-1} \begin{bmatrix} \mathbf{B}^+ \\ \mathbf{0} \end{bmatrix} - \mathcal{D} = \mathbf{G}\mathbf{B}^+ - \mathcal{D}, \\ \mathbf{B}^- &= \mathcal{T}^{-1} \begin{bmatrix} \mathbf{B}^- \\ \mathbf{0} \end{bmatrix} - \mathcal{D} = \mathbf{G}\mathbf{B}^- - \mathcal{D}, \end{aligned}$$

where \mathcal{D} is an $\eta \times u$ matrix with nonnegative integer entries. Note that \mathcal{D} has to be chosen so that the entries of \mathbf{B}^+ and \mathbf{B}^- are nonnegative, i.e., $\mathcal{D} \leq \min(\mathbf{G}\mathbf{B}^+, \mathbf{G}\mathbf{B}^-)$.

Proof: We know from Theorem 5.2 that any redundant Petri net implementation \mathcal{H} as in eqs. (5.5) and (5.6) can be obtained through an *appropriate* similarity transformation $\mathcal{T}\mathbf{q}_h[t] = \mathbf{q}_s[t]$ of the standard redundant implementation \mathcal{H}_s in eq. (5.7). In the process of constructing \mathcal{H} from \mathcal{H}_s , we need to ensure that \mathcal{H} is a valid redundant Petri net implementation of \mathcal{S} , i.e., we need to meet the following requirements:

1. Given any initial condition $\mathbf{q}_s[0]$ ($\mathbf{q}_s[0]$ has nonnegative integer entries), the marking $\mathbf{q}_h[0] = \mathcal{T}^{-1}\mathbf{G}_s\mathbf{q}_s[0] = \mathcal{T}^{-1} \begin{bmatrix} \mathbf{I}_d \\ \mathbf{0} \end{bmatrix} \mathbf{q}_s[0]$ should have nonnegative integer entries.
2. Matrices \mathbf{B}^+ and \mathbf{B}^- should have *nonnegative* integer entries.
3. The set of transitions enabled in \mathcal{S} at any time instant t should be a subset of the set of transitions enabled in \mathcal{H} (so that under any initial condition $\mathbf{q}_s[0]$, a firing sequence \mathcal{X} that is admissible (legal) in \mathcal{S} is also admissible in \mathcal{H}).

The first condition has to be satisfied for any array $\mathbf{q}_s[0]$ with nonnegative integer entries. It is therefore necessary and sufficient that the first d columns of \mathcal{T}^{-1} have nonnegative integer entries. This also ensures that the matrix difference

$$\mathbf{B}^+ - \mathbf{B}^- = \mathcal{T}^{-1} \begin{bmatrix} \mathbf{B}^+ - \mathbf{B}^- \\ \mathbf{0} \end{bmatrix} = \mathcal{T}^{-1} \begin{bmatrix} \mathbf{I}_d \\ \mathbf{0} \end{bmatrix} (\mathbf{B}^+ - \mathbf{B}^-) \equiv \mathbf{G}(\mathbf{B}^+ - \mathbf{B}^-)$$

consists of integer entries. Without loss of generality we let

$$\begin{aligned} B^+ &= GB^+ - \mathcal{D}, \\ B^- &= GB^- - \mathcal{D}, \end{aligned}$$

where the entries of \mathcal{D} are integers chosen so that B^+ and B^- have nonnegative entries (i.e., it is necessary that $\mathcal{D} \leq GB^+$ and $\mathcal{D} \leq GB^-$).

We now check the last condition: transition t_j is enabled in the original Petri net \mathcal{S} at time instant t if and only if $\mathbf{q}_s[t] \geq B^-(:, j)$. If \mathcal{D} has nonnegative entries, then

$$\begin{aligned} \mathbf{q}_s[t] \geq B^-\mathbf{x}_j &\Rightarrow G\mathbf{q}_s[t] \geq GB^-\mathbf{x}_j \\ &\Rightarrow \mathbf{q}_h[t] \geq GB^-\mathbf{x}_j \\ &\Rightarrow \mathbf{q}_h[t] \geq (GB^- - \mathcal{D})\mathbf{x}_j \\ &\Rightarrow \mathbf{q}_h[t] \geq B^-\mathbf{x}_j, \end{aligned}$$

where $B^-(:, j) \equiv B^-\mathbf{x}_j$ (recall that $\mathbf{q}_s[t]$, B^- , G and \mathcal{D} have *nonnegative* integer entries). Therefore, if transition t_j is enabled in the original Petri net \mathcal{S} , it is also enabled in \mathcal{H} (transition t_j is enabled in \mathcal{H} if and only if $\mathbf{q}_h[t] \geq B^-(:, j)$). It is not hard to see that it is also *necessary* for \mathcal{D} to have nonnegative integer entries (otherwise we can find a counterexample by appropriately choosing the initial condition $\mathbf{q}_s[0]$). \square

The following lemma is derived easily from Theorem 5.3 and simplifies the construction of redundant Petri net implementations:

Lemma 5.1 *Let \mathcal{S} be a Petri net with d places, u transitions and state evolution as given in eqs. (5.1) and (5.2). A Petri net \mathcal{H} with $\eta \equiv d + s$ ($s > 0$) places, u transitions and state evolution as in eqs. (5.5) and (5.6) is a redundant implementation of \mathcal{S} if:*

- Matrices B^+ and B^- have nonnegative integer entries given by

$$\begin{aligned} B^+ &= GB^+ - \mathcal{D}, \\ B^- &= GB^- - \mathcal{D}, \end{aligned}$$

where \mathbf{G} is a full-column rank $\eta \times d$ matrix with nonnegative integer entries and \mathcal{D} is an $\eta \times u$ matrix with nonnegative integer entries.

Note that no other restrictions are placed on a redundant Petri net implementation. For example, the entries of the decoding matrix \mathbf{L} and the parity check matrix \mathbf{P} can be negative and/or rational.

5.5.2 Failure Detection and Identification

Separate redundant Petri net implementations are required whenever the structure of the original Petri net cannot be changed. In cases where we have the flexibility to restructure the original Petri net, we can look for *non-separate* redundant Petri net implementations which may have additional features (e.g., use fewer tokens, connections, or places than separate implementations of the same order). This is useful for Petri nets that model digital controllers or computational systems. In essence, non-separate redundant implementations permit fault tolerance considerations during the *design* of the overall (redundant) Petri net.

We will be using non-separate redundant Petri net implementations of the form in Theorem 5.3. Note that the invariance conditions imposed by these constructions can be checked by the parity matrix $\mathbf{P} = \mathbf{P}_\sigma \mathcal{T} = \begin{bmatrix} \mathbf{0} & \mathbf{I}_s \end{bmatrix} \mathcal{T}$.

Transition Failures: Suppose we use a non-separate redundant Petri net implementation to detect and identify transition failures. If transition t_j fires at time instant $t - 1$ (i.e., $\mathbf{x}[t - 1] = \mathbf{x}_j$) but fails to execute its *postconditions*, the erroneous state will be

$$\mathbf{q}_f[t] = \mathbf{q}_h[t] - \mathbf{B}^+(\cdot, j) = \mathbf{q}_h[t] - (\mathbf{GB}^+ - \mathcal{D}) \mathbf{x}_j .$$

The error syndrome is

$$\begin{aligned} \mathbf{P} \mathbf{q}_f[t] &= \mathbf{P}(\mathbf{q}_h[t] - (\mathbf{GB}^+ - \mathcal{D}) \mathbf{x}_j) \\ &= \mathbf{0} - \mathbf{P}(\mathbf{GB}^+ - \mathcal{D}) \mathbf{x}_j \\ &= -(\mathbf{P}_\sigma \mathcal{T})(\mathcal{T}^{-1} \mathbf{G}_\sigma \mathbf{B}^+ - \mathcal{D}) \mathbf{x}_j \\ &= \mathbf{P}_\sigma \mathcal{T} \mathcal{D} \mathbf{x}_j \equiv \mathbf{P} \mathcal{D} \mathbf{x}_j . \end{aligned}$$

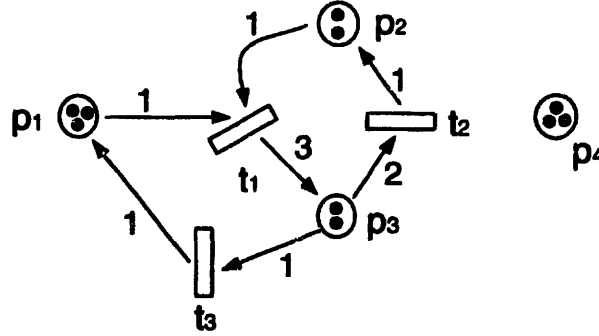


Figure 5-10: Example of a non-separate redundant Petri net implementation that identifies single-transition failures in the Petri net of Figure 5-1.

If the preconditions of transition t_j are not executed, the erroneous state will be

$$\mathbf{q}_f[t] = \mathbf{q}_h[t] + \mathcal{B}^-(\cdot, j) = \mathbf{q}_h[t] + (\mathbf{GB}^- - \mathcal{D}) \mathbf{x}_j$$

and the error syndrome is $\mathbf{P}\mathbf{q}_f[t] = -\mathbf{PD}\mathbf{x}_j$ (calculated similarly).

If we ensure that the columns of matrix \mathbf{PD} are all distinct, we can detect and identify all single-transition failures. Depending on the sign, we can also decide whether postconditions or preconditions were not executed. Note that in the non-separate case the syndromes are linear combinations of columns of \mathcal{D} .

Example 5.7 The Petri net in Figure 5-10 is a non-separate redundant implementation of the Petri net in Figure 5-1. It uses one additional place ($s = 1$); place p_4 , however, is disconnected from the rest of the network and can be treated as a constant. The scheme can detect and identify single-transition failures.

The transformation matrix \mathcal{T}^{-1} and matrix \mathcal{D} that we used in this example are

$$\mathcal{T}^{-1} = \begin{bmatrix} 1 & 1 & 0 & -1 \\ 1 & 0 & 1 & 2 \\ 0 & 2 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \quad \mathcal{D} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 2 & 1 & 1 \end{bmatrix}.$$

They result in matrices $\mathbf{G} = \mathcal{T}^{-1}\mathbf{G}_\sigma$, $\mathbf{B}^+ = \mathbf{G}\mathbf{B}^+ - \mathcal{D}$ and $\mathbf{B}^- = \mathbf{G}\mathbf{B}^- - \mathcal{D}$ as follows:

$$\mathbf{G} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{B}^+ = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 3 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B}^- = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

The decoding matrix $\mathbf{L} = \mathbf{L}_\sigma \mathcal{T}$ and the parity check matrix $\mathbf{P} = \mathbf{P}_\sigma \mathcal{T}$ are given by

$$\mathbf{L} = \begin{bmatrix} 1 & 1 & 0 & -1 \\ 1 & 1 & 1 & -2 \\ -3 & -4 & -2 & 7 \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} 1 & 2 & 1 & -3 \end{bmatrix}.$$

If the parity check $\mathbf{P}\mathbf{q}_h[t]$ is -3 (respectively -2 , -1), then transition t_1 (respectively t_2 , t_3) has failed to execute its postconditions. If the check is 3 (respectively 2 , 1), then transition t_1 (respectively t_2 , t_3) has failed to execute its preconditions. \square

Note that the failure-identification schemes in Figures 5-6 and 5-10 are both able to detect single-transition failures. The scheme in Figure 5-10, however, requires fewer connections (only 7 are required as opposed to 9 in the scheme of Figure 5-6) and fewer places (only 3 as opposed to 4). If desirable, non-separate redundant Petri net implementations can be used to optimize other quantities (e.g., minimize the sum of weights between connections or minimize the number of tokens that reside in a place). This may be a significant advantage when places, connections, or tokens are hard or expensive to establish.

Place Failures: Suppose we use a non-separate redundant Petri net implementation to protect against place failures. If, due to a failure, the number of tokens in place p_i is increased by c , the faulty state is given by $\mathbf{q}_f[t] = \mathbf{q}_h[t] + \mathbf{e}_{p_i}$ (where \mathbf{e}_{p_i} is an η -dimensional array with a unique non-zero entry at its i th position, $\mathbf{e}_{p_i} = c \times \begin{bmatrix} 0 & \dots & 1 & \dots & 0 \end{bmatrix}^T$). The parity check will then be

$$\begin{aligned} \mathbf{P}\mathbf{q}_f[t] &= \mathbf{P}(\mathbf{q}_h[t] + \mathbf{e}_{p_i}) \\ &= \mathbf{0} + \mathbf{P}\mathbf{e}_{p_i} \\ &= c \times \mathbf{P}(:, i). \end{aligned}$$

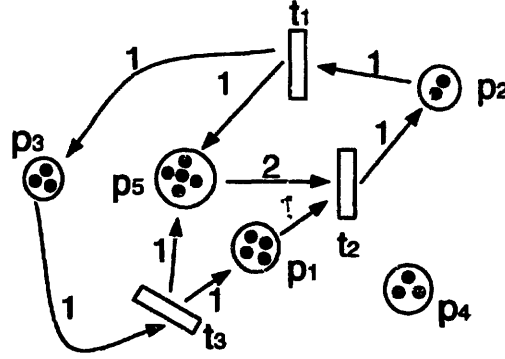


Figure 5-11: Example of a non-separate redundant Petri net implementation that identifies single-place failures in the Petri net of Figure 5-1.

We can detect single-place failures if all columns of the matrix $\mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{I}_s \end{bmatrix} \mathcal{T}$ are non-zero. If the columns of \mathbf{P} are not rational multiples of each other, then we can detect and identify single-place failures.

Example 5.8 In Figure 5-11 we show a non-separate redundant implementation of the Petri net in Figure 5-1. The implementation uses two additional places ($s = 2$) and is able to identify single-place failures. Note that place p_4 essentially acts as a constant.

The transformation matrix \mathcal{T}^{-1} and matrix \mathcal{D} that were used, as well as matrices \mathcal{B}^+ and \mathcal{B}^- , are given by

$$\mathcal{T}^{-1} = \begin{bmatrix} 1 & 2 & 0 & -1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 3 & 0 & 0 & 1 \end{bmatrix}, \quad \mathcal{D} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & 1 \\ 2 & 1 & 1 \\ 2 & 1 & 1 \\ 2 & 1 & 0 \end{bmatrix}, \quad \mathcal{B}^+ = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \quad \mathcal{B}^- = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 2 & 0 \end{bmatrix}.$$

The parity check matrix is

$$\mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{I}_2 \end{bmatrix} \mathcal{T} = \frac{1}{4} \times \begin{bmatrix} -3 & 1 & -1 & 1 & 2 \\ -1 & 3 & 1 & -5 & 2 \end{bmatrix}.$$

□

Note that the syndromes for transition and place failures in non-separate Petri net

embeddings are more complicated than the syndromes in separate embeddings. At the same time, however, we are given some additional flexibility (design parameters), which we can use to construct “optimal” embeddings, that is, embeddings that maintain the desired monitoring capabilities while minimizing certain quantities of interest (such as tokens, connections or places). We will not address such optimization questions; the examples that we presented in this section illustrated some interesting questions that may be posed in the future.

5.6 Applications in Control

A DES is usually monitored through a separate control mechanism that takes appropriate actions based on observations about the state and activity in the system. Control strategies (such as enabling or disabling transitions and external inputs) are often based on the Petri net that models the DES of interest, [72, 114, 71]. In this section we demonstrate that redundant Petri net implementations can facilitate the task of the controller by monitoring active transitions and by identifying “illegal” (unmodeled) transitions. One of the biggest advantages of our approach is that it can be combined with failure detection and identification, and perform monitoring despite incomplete or erroneous information.

5.6.1 Monitoring Active Transitions

In order to time decisions appropriately, the controller of a DES may need to identify ongoing activity in the system. For example, the controller may have to identify all *active*¹¹ transitions or it may need to detect (two or more) transitions that have fired simultaneously. If we use the techniques of Section 5.4, we can construct separate redundant Petri net implementations that allow the controller to detect and locate active transitions by looking at the state of the redundant implementation. The following example illustrates this idea.

Example 5.9 We revisit the Petri net of Figure 5-3 which models a distributed processing network. If we add one extra place ($s = 1$) and use matrices $C = \begin{bmatrix} 1 & 1 & 3 & 2 & 3 & 1 \end{bmatrix}$ and

¹¹We define an active transition as a transition that has not completed yet: it has used all tokens at its input places but has not returned any tokens at its output places. If we use the terminology of the transition failure model in Section 5.3, we can say that active transitions are the ones that have not completed their postconditions.

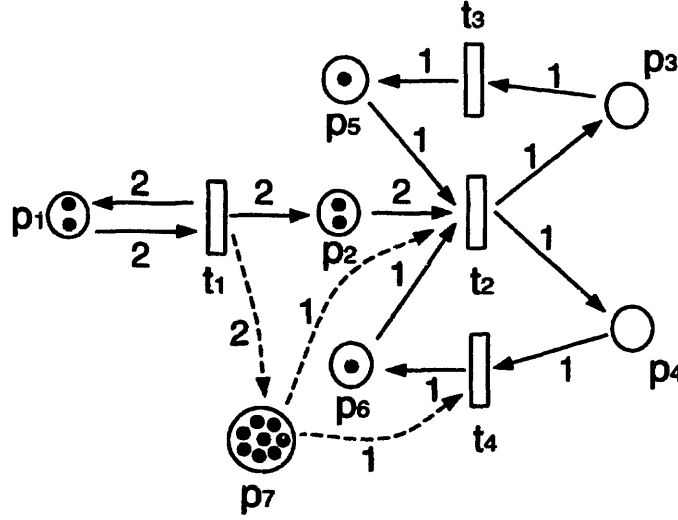


Figure 5-12: Example of a separate redundant Petri net implementation that enhances control of the Petri net of Figure 5-3.

$D = \begin{bmatrix} 2 & 5 & 3 & 1 \end{bmatrix}$, we obtain the separate redundant Petri net implementation shown in Figure 5-12.

At any given time instant t , the controller of the redundant Petri net in Figure 5-12 can determine if a transition is under execution by observing the overall state $\mathbf{q}_h[t]$ of the system and by performing the parity check

$$\begin{bmatrix} -C & I_1 \end{bmatrix} \mathbf{q}_h[t] = \begin{bmatrix} -1 & -1 & -3 & -2 & -3 & -1 & 1 \end{bmatrix} \mathbf{q}_h[t].$$

If the result is 2 (respectively 5, 3, 1), then transition t_1 (respectively t_2 , t_3 , t_4) is under execution. Note that in order to identify whether multiple transitions are under execution, we need to use additional places ($s > 1$).

The interpretation of the additional place in this example is straightforward. Place p_7 acts as a place-holder for special tokens (*acknowledgments*): it receives 2 (respectively 1) such tokens whenever transition t_1 (respectively t_4) is completed; it provides 1 token in order to enable transition t_2 . Explicit acknowledgments about the initiation and completion of each transition are avoided. Furthermore, by adding enough extra places, we can make the above monitoring scheme robust to incomplete or erroneous information (as in the case when a certain place fails to submit the correct number of tokens). \square

5.6.2 Detecting Illegal Transitions

The occurrence of illegal or unmodeled activity in a DES can lead to complete control failure. In this section we use separate redundant Petri net implementations to detect and identify illegal transitions in DES's. We assume that the system modeled by the Petri net is "observed" through two different mechanisms: (i) *place sensors* that provide information about the number of tokens in each place, and (ii) *transition sensors* that indicate when each transition fires. We now discuss how to obtain a separate redundant Petri net implementation to detect discrepancies in the information provided by these two sets of sensors in order to pinpoint illegal behavior.

Suppose that the DES of interest is modeled by a Petri net with state evolution equation

$$\mathbf{q}_s[t+1] = \mathbf{q}_s[t] + \left[\mathbf{B}^+ \mid \mathbf{B}_u^+ \right] \mathbf{x}[t] - \left[\mathbf{B}^- \mid \mathbf{B}_u^- \right] \mathbf{x}[t],$$

where the columns of \mathbf{B}_u^+ and \mathbf{B}_u^- model the postconditions and preconditions of illegal transitions. We will construct a separate redundant implementation of the legal part of the network. The overall system will then have a state evolution equation given by

$$\mathbf{q}_h[t+1] \equiv \begin{bmatrix} \mathbf{q}_{h1}[t+1] \\ \mathbf{q}_{h2}[t+1] \end{bmatrix} = \mathbf{q}_h[t] + \begin{bmatrix} \mathbf{B}^+ & \mathbf{B}_u^+ \\ \mathbf{CB}^+ - \mathbf{D} & \mathbf{0} \end{bmatrix} \mathbf{x}[t] - \begin{bmatrix} \mathbf{B}^- & \mathbf{B}_u^- \\ \mathbf{CB}^- - \mathbf{D} & \mathbf{0} \end{bmatrix} \mathbf{x}[t].$$

Our goal is to choose \mathbf{C} and \mathbf{D} so that we can detect illegal behavior. Information about the state of the upper part of the redundant implementation (with state evolution $\mathbf{q}_{h1}[t+1] = \mathbf{q}_{h1}[t] + \left[\mathbf{B}^+ \mid \mathbf{B}_u^+ \right] \mathbf{x}[t] - \left[\mathbf{B}^- \mid \mathbf{B}_u^- \right] \mathbf{x}[t]$) will be provided to the controller by the place sensors. The effect of illegal transitions will be captured in this part of the redundant implementation by the changes in the number of tokens in the affected places. The additional places (with state evolution $\mathbf{q}_{h2}[t+1] = \mathbf{q}_{h2}[t] + \left[\mathbf{CB}^+ - \mathbf{D} \mid \mathbf{0} \right] \mathbf{x}[t] - \left[\mathbf{CB}^- - \mathbf{D} \mid \mathbf{0} \right] \mathbf{x}[t]$) are internal to the controller and act only as test places¹². Once the number of tokens in these test places is initialized appropriately (i.e., $\mathbf{q}_{h2}[0] = \mathbf{C}\mathbf{q}_{h1}[0]$), the controller removes or adds tokens to these places based on which (legal) transitions take place. Therefore, the information about the bottom part of the system is provided by the

¹²Test places cannot inhibit transitions and can have a negative number of tokens. A connection from a test place p_i to transition t_j simply indicates that the number of tokens in p_i will decrease when t_j fires.

transition sensors.

If an illegal transition fires at time instant t , the illegal state $\mathbf{q}_f[t]$ of the redundant implementation is given by

$$\mathbf{q}_f[t] = \mathbf{q}_h[t] + \begin{bmatrix} \mathbf{B}_u^+ \\ \mathbf{0} \end{bmatrix} \mathbf{x}_u[t] - \begin{bmatrix} \mathbf{B}_u^- \\ \mathbf{0} \end{bmatrix} \mathbf{x}_u[t] \equiv \mathbf{q}_h[t] + \begin{bmatrix} \mathbf{B}_u \\ \mathbf{0} \end{bmatrix} \mathbf{x}_u[t],$$

where $\mathbf{B}_u \equiv \mathbf{B}_u^+ - \mathbf{B}_u^-$ and $\mathbf{x}_u[t]$ denotes an array with all zero entries, except an entry that is “1” and indicates the illegal transition that fired. If we perform the parity check $\mathbf{P}\mathbf{q}_f[t]$, we get

$$\begin{aligned} \mathbf{P}\mathbf{q}_f[t] &= \begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix} \mathbf{q}_f[t] \\ &= \begin{bmatrix} -\mathbf{C} & \mathbf{I}_s \end{bmatrix} \left(\mathbf{q}_h[t] + \begin{bmatrix} \mathbf{B}_u \\ \mathbf{0} \end{bmatrix} \mathbf{x}_u[t] \right) \\ &= -\mathbf{C}\mathbf{B}_u\mathbf{x}_u[t]. \end{aligned}$$

Therefore, we can identify which illegal transition has fired if all columns of $\mathbf{C}\mathbf{B}_u$ are unique.

Example 5.10 The controller of the maze in Figure 5-2 obtains information about the state of the system through a set of detectors. More specifically, each room is equipped with a “mouse sensor” that indicates whether the mouse is in that room. In addition, “door sensors” get activated whenever the mouse goes through the corresponding door.

Suppose that due to a bad choice of materials, the maze of Figure 5-2 is built in a way that allows the mouse to dig a tunnel connecting rooms 1 and 5 and a tunnel connecting rooms 1 and 4. This leads to the following set of illegal (i.e., non-door) transitions in the network:

$$\mathbf{B}_u = \begin{bmatrix} 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 1 & 0 & 0 \end{bmatrix}.$$

In order to detect the existence of such tunnels, we can use a redundant Petri net implementation with one additional place ($s = 1$), $C = \begin{bmatrix} 1 & 1 & 1 & 2 & 3 \end{bmatrix}$ and $D = \begin{bmatrix} 1 & 1 & 1 & 1 & 2 & 1 \end{bmatrix}$. The resulting redundant matrices B^+ and B^- are given by

$$B^+ = \left[\begin{array}{c|c} B^+ & B_u^+ \\ \hline CB^+ - D & 0 \end{array} \right] = \left[\begin{array}{cccccc|cccc} 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right],$$

$$B^- = \left[\begin{array}{c|c} B^- & B_u^- \\ \hline CB^- - D & 0 \end{array} \right] = \left[\begin{array}{cccccc|cccc} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{array} \right].$$

The upper part of the network is observed through the place ("mouse") sensors. The number of tokens in the additional place is updated based on information from the transition ("door") sensors (it receives 2 tokens when transition t_3 fires; it loses 1 token each time transition t_4 or t_5 fires). The parity check is given by

$$\begin{bmatrix} -1 & -1 & -1 & -3 & -2 & -1 & 1 \end{bmatrix} \mathbf{q}_h[t]$$

and is 0 if no illegal activity has taken place. It is 2 (respectively -2 , 1 , -1) if illegal transition $B_u(:, 1)$ (respectively $B_u(:, 2)$, $B_u(:, 3)$, $B_u(:, 4)$) has taken place. Clearly, in order to be able to detect the existence of a tunnel in the maze, we only need to use three door sensors. \square

5.7 Summary

In this chapter we have presented techniques that systematically incorporate redundancy into a given Petri net. The result is a methodology for monitoring failures or facilitating control of underlying DES's. We defined and characterized classes of redundant Petri net implementations, which we then used for systematically constructing failure detection and identification schemes for a variety of failures. Our approach extended the results on fault-tolerant Petri nets in [97, 98] by introducing non-separate redundant implementations, by allowing the study of non-pure Petri nets and by applying the methods to the development of monitoring schemes for DES's. Our monitors detect and identify failures by using simple linear checks; furthermore, they can be made robust to erroneous or incomplete information and do not need to be re-constructed when the initial state of the Petri net changes.

Future work needs to better characterize the cost of our schemes in terms of an appropriate set of measures (e.g., the total number of links and/or the number of tokens). Once these measures are well understood, we need to *systematically* construct monitoring schemes with minimal cost (for example, choose linear codes and weights that minimize the number of connections). Along the same lines, we should investigate the constraints that are required to develop hierarchical/distributed monitoring schemes for complex Petri nets. Another interesting future extension is to study examples where a subset of the transitions is uncontrollable and/or unobservable (i.e., when links to or from the transition are not possible, [71]), or cases where the number of tokens in a subset of the places is unavailable. The main problem is to algorithmically devise schemes that have the desirable error-detecting capabilities while minimizing the associated "cost". In order to achieve these objectives, we may want to relax some of our assumptions: for example, if we drop the requirement that failure detection/identification is concurrent, we can build monitors that are matched to the dynamic nature of the underlying Petri net and the types of failures expected. These monitors will perform "non-concurrent" failure identification and will also have the ability to handle failures in the error-correcting mechanism. Finally, in order to achieve error-correction we need to introduce *failure-recovery transitions* (which will serve as correcting actions when particular failures are detected).

Chapter 6

Unreliable Error-Correction

6.1 Introduction

In this chapter we take a more “classical” view of fault tolerance (as studied, for example, in [107, 112, 95, 96, 33]) and construct reliable dynamic systems exclusively out of unreliable components, including unreliable components in the error-correcting mechanism. As we add redundancy into our system, we allow the probability of failure in each component to remain constant¹. We assume that all components that we use suffer *transient* failures with constant probability *independently* between different components and *independently* between different time steps. Since our systems evolve with time according to their internal state, we need to deal with the effects of *error propagation*.

This chapter is organized as follows. In Section 6.2 we state the problem and our assumptions. In Section 6.3 we introduce and analyze a *distributed voting scheme* that is a generalization of modular redundancy, employing multiple unreliable system replicas and multiple unreliable voters. We show that by increasing the amount of redundancy (system replicas and voters) modestly, one can significantly extend the time interval for which a fault-tolerant implementation will operate “correctly”. More specifically, increasing the amount of redundancy (systems and voters) by a constant amount, allows one to double the number of time steps for which the fault-tolerant implementation will operate within a specified probability of failure. In Section 6.4 we combine our distributed voting scheme

¹We are essentially adopting the second of the two different approaches to fault tolerance discussed in Chapter 1.

with linear codes that can be corrected with low complexity. This allows us to obtain interconnections of *identical* linear finite-state machines that operate in parallel on *distinct* input streams and use only a *constant* amount of redundant hardware per machine to achieve arbitrarily small probability of failure over any specified time interval. Equivalently, given a specified probability of failure, one can achieve “correctness” for any given, finite number of time steps using a constant amount of redundancy per system. Along the way we make connections with previous work on reliable computational circuits, stable memories, and graph and coding theory.

6.2 Problem Statement

In a dynamic system an incorrect transition to the next state at a particular time step will not only affect the output at that time step, but will typically also affect the state and therefore the output of the system at later time steps. In Chapters 2 through 5 we added structured redundancy into our dynamic systems, and then performed error detection and correction by identifying violations of the state constraints and taking appropriate correcting actions. This approach works nicely if the error-correcting mechanism is fault-free. If we are to allow failures in *all* components of the system, however, we need to ensure that failures in the error-correcting mechanism will not be devastating. To realize the severity of the problem, recall the toy example that we introduced in Chapter 1: assume that we have a dynamic system (e.g., a finite-state machine) in which the probability of making a transition to an incorrect next state (on any input) is p_s (independently between time steps). Clearly, the probability that the system follows the correct state trajectory for L consecutive time steps is $(1 - p_s)^L$, and goes to zero exponentially with L . One solution is to use modular redundancy with feedback (as in Figure 1-2): at the end of each time step, a voter decides what the correct state is and feeds this corrected state back to *all* systems. If the voter feeds back an incorrect state with probability p_v , this approach will *not* work: after L time steps, the probability that the system has followed the correct state trajectory is at best² $(1 - p_v)^L$ and again goes down exponentially with L . The problem is

²This bound ignores the (rare) possibility that a failure in the voter may result in feeding back the correct state (because the majority of the systems are in an incorrect state).

that failures in the voter (or more generally in the error-correcting mechanism) corrupt the overall redundant system state and cause error propagation. Therefore, given unreliable systems *and* unreliable error-correction, we need to use a different approach in order to guarantee reliable state evolution for a larger number of time steps.

The first question that we ask in this chapter is the following: given unreliable systems and unreliable voters (more generally given unreliable components), is there a way to guarantee the operation of a dynamic system for an arbitrarily large (but finite) number of time steps? Furthermore, what is the tradeoff between the amount of redundant hardware and the associated probability of failure or the number of time steps for which the system is required to operate reliably? Our approach will use the scheme shown in Figure 1-3 but will allow failures in both the redundant implementation and the error-correcting mechanism. Clearly, since all components of this construction are allowed to fail, the system will not necessarily be in the correct state at the end of a particular time step. What we hope for, however, is for its state to be within a set of states that *correspond* to the correct one: in other words, if a fault-free error corrector/decoder was available, then we would be able to obtain the correct state from the possibly corrupted state of the redundant system. This situation is shown in Figure 6-1: at the end of each time step, the system is within a set of states that could be corrected/decoded to the actual state (in which the underlying system would be, had there been no failures). Even when the decoding mechanism is not fault-free, our approach is still desirable because it guarantees that the probability of a decoding failure will not increase with time in an unacceptable fashion. As long as the redundant state is within the set of states that represent the actual (underlying) state, the decoding at each time step will be incorrect with a *fixed* probability, which depends only on the reliability of the decoding mechanism and does not diminish as the dynamic system evolves in time. Our method guarantees that the probability of incorrect state evolution during a certain time interval is much smaller in the redundant dynamic system than in the original one.

6.3 Distributed Voting Scheme

The problem in the modular redundancy scheme of Figure 1-2 is that a voter failure corrupts the states of *all* system replicas. This results in an *overall failure*, i.e., a situation where

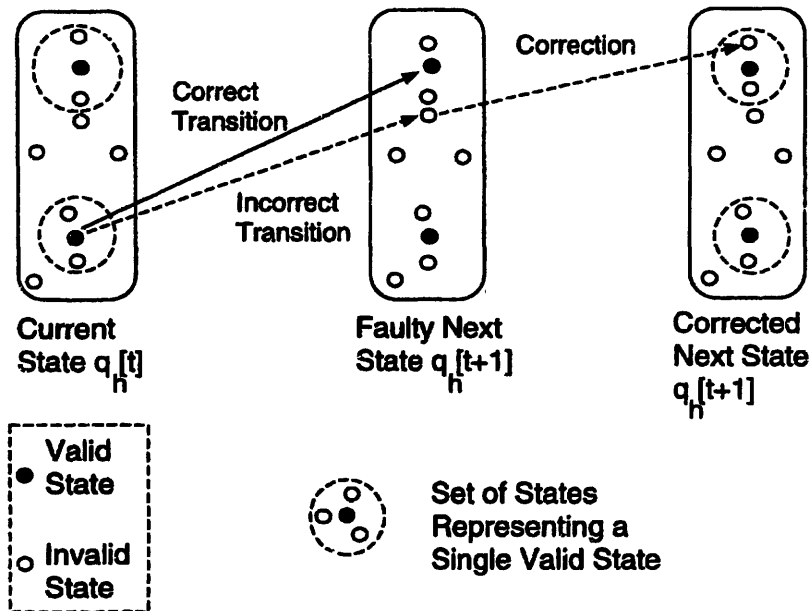


Figure 6-1: Reliable state evolution using unreliable error-correction.

the state of our redundant implementation does not correctly represent the state of the underlying dynamic system (e.g., if the majority of the systems agrees on an incorrect state, then even a *fault-free* voter would not be able to recover the correct state of the underlying dynamic system). To avoid this situation, we need to ensure that failures in the error-correcting mechanism do not have such devastating consequences. One way to achieve this is to have several voters and to perform the error-correction in a *distributed* fashion, as shown in Figure 6-2. The arrangement in Figure 6-2 uses n system replicas and n voters. All n replicas are initialized at the same state and receive the same inputs. Each voter receives "ballots" from all systems and feeds back a correction to only *one* of the systems. This way, a failure in a single voter only corrupts *one* of the system replicas and not all of them.

Notice that the redundant implementation of Figure 6-2 operates correctly as long as more than half of the n systems are in the correct state, since a fault-free voter is then able to recover the correct state of the underlying system. Also notice that as long as half or more of the systems are in the correct state all voters ideally feed back the correct state, unless there is a voter failure. Therefore, a failure in a particular voter or a particular system will be corrected at future time steps with high probability if at least half of the

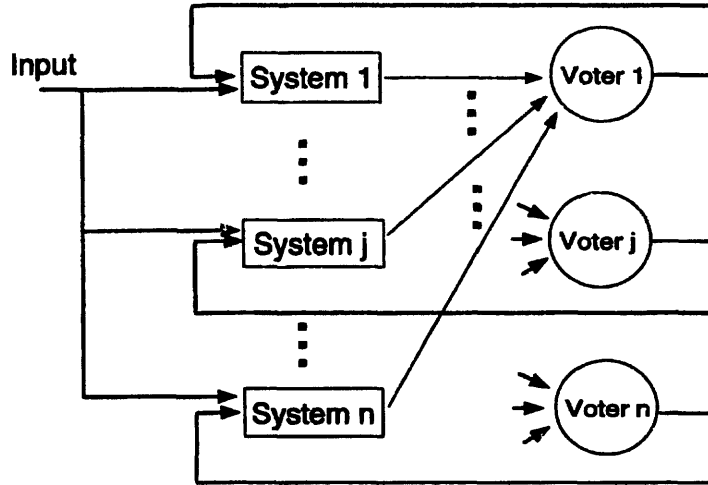


Figure 6-2: Modular redundancy using a distributed voting scheme.

other systems end up in the correct state. We will say that an *overall failure* happens when half or more of the systems are in a corrupted state³. Our goal will be to ensure that, with high probability, the fault-tolerant implementation will operate for a finite number of time steps with no overall failure, i.e., with at least $\lceil \frac{n}{2} \rceil$ systems in the correct state at any given time step. Note that it is *not* necessary for each of these $\lceil \frac{n}{2} \rceil$ systems to remain in the correct state for all consecutive time steps.

Theorem 6.1 *Suppose that each system takes a transition to an incorrect state with probability p_s and each voter feeds back an incorrect state with probability p_v (independently between systems, voters and time steps). Then the probability of an overall failure at or before time step L (starting at time step 0) can be bounded as follows:*

$$\Pr[\text{overall failure at or before time step } L] \leq L \sum_{i=\lceil n/2 \rceil}^n \binom{n}{i} p^i (1-p)^{n-i},$$

where $p \equiv p_v + (1 - p_v)p_s$. This probability goes down exponentially with the number of systems n if and only if $p < \frac{1}{2}$.

Proof: Given that there is no overall failure at time step $\tau-1$, the probability that system j

³This definition of an overall failure is actually conservative because the overall redundant implementation may perform as expected even if more than half of the systems are in an incorrect state. What we really need is for the *majority* of the systems to be in the correct state.

ends up in an incorrect state at time step τ is bounded by the probability that either its voter fails or its voter does not fail, but system j itself transitions to an incorrect state, i.e.,

$$\Pr[\text{system } j \text{ in incorrect state at } \tau \mid \text{no overall failure at } \tau-1] \leq p_v + (1 - p_v)p_s \equiv p.$$

Given no overall failure at time step $\tau-1$, the probability of an overall failure at time step τ is the probability that half or more of the n system replicas fail:

$$\Pr[\text{overall failure at } \tau \mid \text{no overall failure at } \tau-1] \leq \sum_{i=\lfloor n/2 \rfloor}^n \binom{n}{i} p^i (1-p)^{n-i}.$$

Using the above expression, we can bound the probability of an overall failure at or before a certain time step L using the union bound:

$$\Pr[\text{overall failure at or before } L] \leq L \sum_{i=\lfloor n/2 \rfloor}^n \binom{n}{i} p^i (1-p)^{n-i}.$$

Note that the bound on the probability of failure increases linearly with the number of time steps (because of the union bound). The bound goes down exponentially with n if and only if p is less than $\frac{1}{2}$; we can see this using the Sterling approximation and the results on p. 531 of [38] and assuming $p < \frac{1}{2}$, we see that

$$\binom{n}{n/2} p^{n/2} (1-p)^{n/2} \leq \sum_{i=n/2}^n \binom{n}{i} p^i (1-p)^{n-i} \leq \frac{1-p}{1-2p} \binom{n}{n/2} p^{n/2} (1-p)^{n/2}$$

(where for simplicity we have taken n to be even). Since

$$\sqrt{\frac{1}{2n}} 2^n \leq \binom{n}{n/2} \leq \sqrt{\frac{2}{\pi n}} 2^n,$$

we conclude that $\sum_{i=n/2}^n \binom{n}{i} p^i (1-p)^{n-i}$ will decrease exponentially with n if and only if $p(1-p) < \frac{1}{4}$ (i.e., if and only if p is less than $\frac{1}{2}$). \square

A potential problem with the arrangement in Figure 6-2 is the fact that, as we increase

n , the complexity of each voter (and therefore p_v) increases⁴. If this causes p to increase beyond $\frac{1}{2}$, we can no longer guarantee exponential decay of the probability of an overall failure (note that we can ensure that $p < \frac{1}{2}$ as long as $p_v < \frac{\frac{1}{2}-p_s}{1-p_s}$). In the next section we consider an arrangement in which the number of inputs to each voter is *fixed*, which means that the voter complexity and p_v remain constant as we increase the number of systems and voters. More generally, it would be interesting to explore the performance of schemes where, for example, we fix or bound the number of inputs to each voter. Such situations have been studied in the context of reliable computational circuits and stable memories⁵.

Another concern about the approach in Figure 6-2 is that, in order to construct dynamic systems that fail with an acceptably small probability of failure during any (finite) time interval, we may need to increase the hardware in our redundant implementations unacceptably. More specifically, if we double the number of time steps, the bound in Theorem 6.1 suggests that we may need to increase the number of system replications by a constant amount (in order to keep the probability of an overall failure at the same level).

To make an analogy with the communications problem of digital transmission through an unreliable link, what we have shown in this section is very similar to what can be achieved *without* the coding techniques that Shannon suggested in [95, 96]. More specifically, in the communications setting we can make the probability of a transmission error as small as we want by arbitrarily replicating (retransmitting) the bit we want to transmit, but at the cost of correspondingly reducing the rate at which information is transmitted. If, however, we are willing to transmit k bits as a block, then we can use coding techniques to achieve an arbitrarily small probability of transmission error with a *constant* amount of redundancy per bit⁶. In the next section we transfer this coding paradigm to our computational setting. Specifically, we show that for *identical* linear finite-state machines that operate in parallel on *distinct* input sequences, one can design a scheme that requires only a *constant* amount

⁴A nice discussion and further pointers on the hardware complexity of voting mechanisms can be found in [48]; more general voting algorithms and complexity issues are discussed in [79].

⁵In [69, 82] an (m, k, α, β) *compressor* graph is defined as a bipartite multigraph with m input nodes and m output nodes such that the following property is satisfied: for every set A that contains at most αm input nodes, the set of output nodes that are connected to at least $k/2$ inputs in A contains at most βm elements. We can use such multigraphs to generalize the distributed voting scheme that we discussed in this section.

⁶This is achieved by encoding k information bits into $n > k$ bits, transmitting these n bits through the channel, receiving n (possibly corrupted) bits, performing error-correction and finally decoding the n bits into the original k bits.

of redundancy per machine/sequence to achieve arbitrarily small probability of failure over any finite time interval.

6.4 Reliable Linear Finite-State Machines Using Constant Redundancy

In this section we combine a variation of the distributed voting scheme of the previous section with linear coding techniques. We obtain interconnections of identical linear finite-state machines (LFSM's) that operate in parallel on *distinct* input streams and require only a *constant* amount of redundancy *per machine* to achieve an arbitrarily small probability of failure over a specified (finite) time interval. The codes that we use are low-density parity check codes (see [37, 68, 99, 101]); error-correction is of low complexity⁷ and can be performed using majority voters and XOR gates. The voters that we will be using vote on $J-1$ bits (where J is a constant). Throughout this section we will assume that these $(J-1)$ -bit voters fail with probability p_v and the 2-input XOR gates fail with probability p_x . For simplicity, we will also assume that no failures take place in the single-bit memory elements (flip-flops), although our approach can also handle this type of failure.

Before we describe our construction, we provide an introduction to low-density parity check codes [37], and discuss how they have been used to construct *stable memories*, [104, 106, 105].

6.4.1 Low-Density Parity Check Codes and Stable Memories

An (n, k) low-density parity check (LDPC) code is a linear code that represents k bits of information using n total bits, [37]. Just like any linear code, an LDPC code has an $n \times k$ generator matrix \mathbf{G} with full-column rank; the additional requirement is that the code has a parity check matrix \mathbf{P} that (is generally sparse and) has exactly K "1's" in each row and J "1's" in each column. It can be easily shown that the ratio $\frac{n}{K}$ has to be an integer and that \mathbf{P} has dimension $\frac{nJ}{K} \times n$, [37]. Each bit in a codeword is involved in J parity checks, and each of these J parity checks involves $K-1$ other bits. Note that the rows of \mathbf{P} are allowed

⁷We use codes with low decoding complexity because failures in the correcting mechanism could otherwise become the bottleneck of our fault-tolerant scheme.

to be linearly dependent (i.e., \mathbf{P} can have more than $n - k$ rows) and that the generator matrix \mathbf{G} of an LDPC code is not necessarily sparse.

In his seminal thesis [37] Gallager studied ways to construct and decode LDPC codes. In particular, he constructed sequences of (n, k) LDPC codes for fixed J and K with *rate* $\frac{k}{n} \geq 1 - \frac{J}{K}$. Gallager suggested and analyzed the performance of simple iterative procedures for correcting erroneous bits in corrupted codewords; we summarize these procedures below.

Iterative Decoding For each bit:

1. evaluate the J associated parity checks (since each column of \mathbf{P} has exactly J “1’s”);
2. if more than half of the J parity checks for a particular bit are unsatisfied, flip the value of that bit; do this for all bits concurrently;
3. iterate;

In order to analytically evaluate the performance of this iterative scheme, Gallager slightly modified his approach:

Modified Iterative Decoding Replace each bit d_i with J *bit-copies* $\{d_i^1, d_i^2, \dots, d_i^J\}$ (all bit-copies are initially the same); obtain new estimates of each of these copies (i.e., J new estimates $\{\hat{d}_i^1, \hat{d}_i^2, \dots, \hat{d}_i^J\}$) by

1. evaluating $J-1$ parity checks for each bit-copy (each time step excluding one *different* parity check from the original set of J checks);
2. flipping the value of a particular bit-copy if half or more of the $J-1$ parity checks are unsatisfied;
3. iterating.

A hardware implementation of Gallager’s modified iterative decoding scheme can be seen in Figure 6-3. Initially, we start with J copies of an (n, k) codeword (i.e., a total of Jn bits). During each iteration, each bit-copy is corrected using an error-correcting mechanism of the form shown in the figure: there are a total of $J-1$ parity checks, each evaluated via $K-1$ 2-input XOR gates; the output of each voter is “1” if half or more of the $J-1$ parity checks are non-zero. The correction is accomplished by XOR-ing the output of the voter with the previous value of the bit-copy.

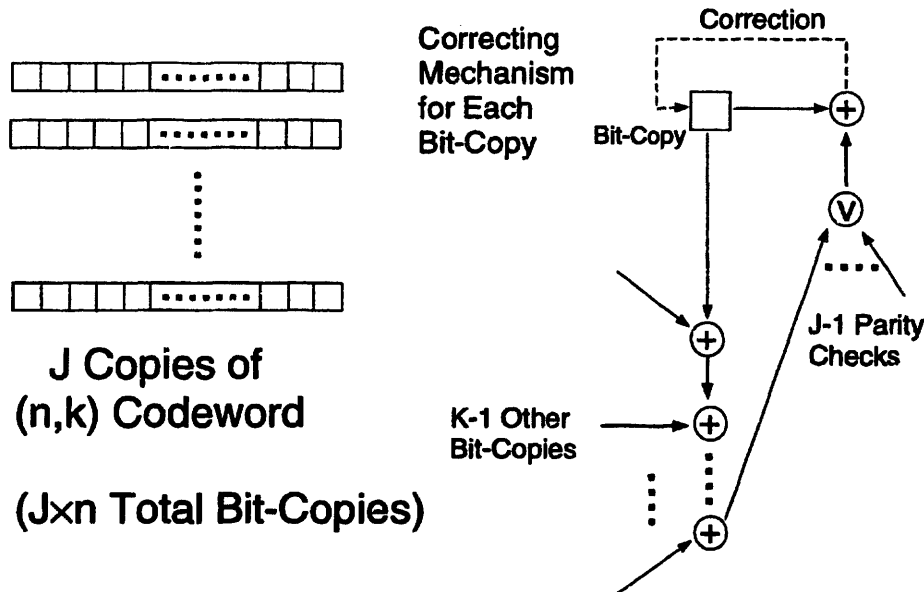


Figure 6-3: Hardware implementation of the modified iterative decoding scheme for LDPC codes.

Note that in the modified iterative decoding scheme, each parity check requires $K - 1$ input bits (other than the bit-copy we are trying to estimate). Since each of these input bits has J different copies, we have some flexibility in terms of which particular copy we use when obtaining an estimate for copy d_i^j of bit d_i ($1 \leq j \leq J$). If we are careful enough in how we choose among these J bit-copies, we can guarantee that the number of *independent iterations* will be non-zero (i.e., the number of independent iterations is the number of iterations for which no decision about any bit-copy is based on a previous estimate of this same bit-copy). In particular, when estimating a copy of bit d_i using an estimate of bit d_j , we should use the bit-copy of d_j that disregarded the parity check involving d_i (otherwise, the estimate of d_i would immediately depend upon its previous estimate).

The number of independent iterations, which we denote by m , is important because during these first m iterations, the probability of error in an estimate for a *particular* bit-copy can be calculated easily using independence. It is not hard to show that the number of independent iterations for *any* LDPC code is bounded by $m < \frac{\log n}{\log[(K-1)(J-1)]}$; in his thesis Gallager suggested a procedure for constructing sequences of (n, k) LDPC codes with fixed J, K (i.e., with parity check matrices that have J "1's" in each row and K "1's" in each column) such that $\frac{k}{n} \geq 1 - \frac{J}{K}$ and with the number of independent iterations m bounded

by

$$m + 1 > \frac{\log n + \log \frac{KJ-K-J}{2K}}{2 \log[(K-1)(J-1)]} > m.$$

Building on Gallager's work, Taylor considered the following problem in [106]: suppose that we have unreliable memory elements (flip-flops) which can store a single bit ("0" or "1") but may fail *independently* during each time step. More specifically, a bit stored in an unreliable flip-flop may get flipped with probability p_c during each time step. Taylor used (n, k) LDPC codes to construct reliable (or *stable*) memory arrays out of unreliable flip-flops: a reliable memory array uses n flip-flops to store k bits of information. At the end of each time step an *unreliable* error-correcting mechanism re-establishes the correct state in the memory array. (Note that if error-correction is fault-free, the problem reduces to the problem of communicating through a sequence of identical unreliable channels, while performing error detection/correction at the end of each transmission through a channel: the first node transmits an (n, k) codeword to the second node via an unreliable communication link; after performing error detection and correction, the second node transmits the corrected (ideally the same) codeword to the third node, and so forth.) The memory scheme performs reliably for L time steps if at *any* time step τ ($0 \leq \tau \leq L$) the k information bits can be *deduced* from the n memory bits. This means that the codeword stored in the memory at time step τ is within the set of n -bit sequences that get decoded to the originally stored codeword (i.e., if we employ *fault-free* iterative decoding at the end of time step τ , we will obtain the codeword that was stored in the memory array at time step 0).

Taylor used LDPC codes and Gallager's modified iterative procedure to build a correcting mechanism out of 2-input XOR gates and $(J-1)$ -bit voters that may fail (i.e., output an incorrect bit) with probabilities p_x and p_v respectively. Note that since he was using the modified iterative decoding scheme, there were J estimates associated with each bit, each of which was corrected based on a simple circuit that involved one $(J-1)$ -bit voter and $1 + (K-1)(J-1)$ 2-input XOR gates (as shown in Figure 6-3). Taylor constructed reliable memory arrays using (n, k) LDPC codes (with $\frac{k}{n} \geq 1 - \frac{J}{K}$, $J < K$) such that the probability of a failure increases linearly with the number of time steps τ and decreases polynomially with k (i.e., the probability of failure is $O(k^{-\beta})$ for a *positive* constant β). Therefore, by

increasing k we can make the probability of failure arbitrarily small while keeping $\frac{k}{n} \geq 1 - \frac{J}{K}$ (i.e., the redundancy per bit remains below a constant). Note that Taylor's construction of reliable memory arrays uses Jn voters, Jn flip-flops and $Jn[1 + (J - 1)(K - 1)]$ 2-input XOR gates; since $\frac{n}{k} \leq \frac{1}{1 - J/K}$, the overhead per bit (in terms of overall flip-flops, XOR gates and voters) remains below a constant as k and n increase. Taylor also showed that one can reliably perform the XOR operation on k pairs of bits by performing component-wise XOR-ing on two (n, k) codewords. In fact, he showed that one can reliably perform a sequence of τ such component-wise XOR operations, [105]. His results for general computation (i.e., component-wise binary operations other than XOR) were in error; see the discussions in [100, 83].

Here, we extend Taylor's techniques to handle LFSM's.

6.4.2 Reliable Linear Finite-State Machines

Without loss of generality, we assume the LFSM that we are trying to protect has a d -dimensional state that evolves according to

$$\mathbf{q}[t + 1] = \mathbf{A}_c \mathbf{q}[t] \oplus \mathbf{b}x[t], \quad (6.1)$$

where \mathbf{A}_c is a $d \times d$ matrix in classical canonical form (see the discussion in Section 4.2). For simplicity, we also assume that the LFSM has a single input. Note that any such LFSM can be implemented using XOR gates and flip-flops as outlined in Chapter 4. In such implementations, each bit in the next-state vector $\mathbf{q}[t + 1]$ can be generated using at most two bits from $\mathbf{q}[t]$ and at most two 2-input XOR gates (this is due to the structure of matrix \mathbf{A}_c).

We will take k such LFSM's that run in parallel (each with possibly *different* initial state and *different* input streams) and use an LDPC scheme to protect their evolution in time (which constitutes computation on their input streams). What we have are k parallel *instantiations* of the system in eq. (6.1):

$$\begin{bmatrix} \mathbf{q}_1[t + 1] & \cdots & \mathbf{q}_k[t + 1] \end{bmatrix} = \mathbf{A}_c \begin{bmatrix} \mathbf{q}_1[t] & \cdots & \mathbf{q}_k[t] \end{bmatrix} \oplus \mathbf{b} \begin{bmatrix} x_1[t] & \cdots & x_k[t] \end{bmatrix}.$$

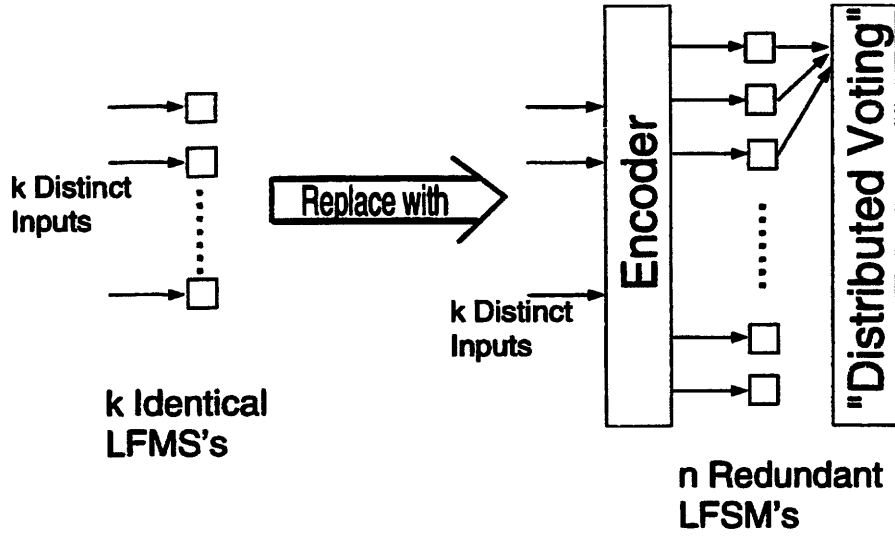


Figure 6-4: Replacing k LFSM's with n redundant LFSM's.

Let \mathbf{G} be the $n \times k$ encoding matrix of a linear code. If we post-multiply both sides of the above equation by \mathbf{G}^T , we get the following n encoded parallel instantiations:

$$\begin{aligned}
 \begin{bmatrix} \mathbf{q}_1[t+1] & \cdots & \mathbf{q}_k[t+1] \end{bmatrix} \mathbf{G}^T &= \left(\mathbf{A}_c \begin{bmatrix} \mathbf{q}_1[t] & \cdots & \mathbf{q}_k[t] \end{bmatrix} \right) \mathbf{G}^T \oplus \\
 &\oplus \left(\mathbf{b} \begin{bmatrix} x_1[t] & \cdots & x_k[t] \end{bmatrix} \right) \mathbf{G}^T \\
 &= \mathbf{A}_c \left(\begin{bmatrix} \mathbf{q}_1[t] & \cdots & \mathbf{q}_k[t] \end{bmatrix} \mathbf{G}^T \right) \oplus \\
 &\oplus \mathbf{b} \left(\begin{bmatrix} x_1[t] & \cdots & x_k[t] \end{bmatrix} \mathbf{G}^T \right),
 \end{aligned}$$

or equivalently

$$\begin{bmatrix} \xi_1[t+1] & \cdots & \xi_n[t+1] \end{bmatrix} = \mathbf{A}_c \begin{bmatrix} \xi_1[t] & \cdots & \xi_n[t] \end{bmatrix} \oplus \underbrace{\mathbf{b} \left(\begin{bmatrix} x_1[t] & \cdots & x_k[t] \end{bmatrix} \mathbf{G}^T \right)}_{e(x_1[t], \dots, x_k[t])} \quad (6.2)$$

where

$$\begin{bmatrix} \xi_1[\tau] & \cdots & \xi_n[\tau] \end{bmatrix} \equiv \begin{bmatrix} \mathbf{q}_1[\tau] & \cdots & \mathbf{q}_k[\tau] \end{bmatrix} \mathbf{G}^T. \quad (6.3)$$

Effectively, we have n LFSM's with state evolution of the form of eq. (6.1), performing k *different* encoded instantiations of the system in eq. (6.1). As shown in Figure 6-4, we

have replaced the operation of k identical LFSM's acting on distinct input streams by n redundant LFSM's acting on encoded versions of the k original inputs (encoded according to an (n, k) linear code with generator matrix G). We will use a *separate* set of flip-flops and XOR gates to implement each of the n redundant systems, and we assume for simplicity that the flip-flops are reliable. We also assume that encoding is performed instantaneously and is fault-free. (The latter assumption can be relaxed; the real issue with the encoding mechanism is its time and hardware complexity — see the discussion in Section 6.4.3.) At each time step, we will supply encoded inputs to these n systems and allow each one to evolve to its corresponding (and possibly erroneous) next state. At the end of the time step we will correct errors in the new states of the n systems by performing error-correction on d codewords from our (n, k) code, with the i th codeword obtained by collecting the i th bit from each of the n state vectors.

If error-correction was fault-free, we could invoke⁸ Shannon's result and ensure that the condition in eq. (6.3) will be satisfied with high probability (at least for a specified, finite number of time steps). By increasing both k and n , we would be able to make the probability of "error per time step" (i.e., the probability of an overall failure at a particular time step given *no* corruption at the previous time step, denoted by $\text{Pr}[\text{error per time step}]$) arbitrarily small. Then, using the union bound, we could conclude that the probability of an overall failure over L consecutive time steps is bounded as follows:

$$\text{Pr}[\text{overall failure at or before time step } L] \leq L \text{Pr}[\text{error per time step}].$$

In order to allow failures in the error-correcting mechanism, we use the following approach: we employ LDPC codes and perform error-correction in each bit using an unreliable error-correcting mechanism (like the one in Figure 6-3). Note that the error-correcting mechanism for each bit is implemented using *separate* sets of unreliable XOR gates and

⁸This argument can be made more precise by looking at the probability of error per bit during each time step. Assuming that there are no corruptions in any of the n state vectors at the beginning of a given time step, we can easily bound the probability of an error in each bit of the n next-state vectors (based on the number of XOR operations that are involved). If this bit-error probability is less than $\frac{1}{2}$ and if we ensure that errors among different bits are independent (which will certainly be the case if we use separate XOR gates to generate each bit), then our problem essentially reduces to an unreliable communication problem. The key is that by assuming fault-free error-correction, we are ensuring that at the beginning of each time step the overall redundant state will be correct (unless, of course, failures during state evolution have resulted in an overall failure and have caused the correcting mechanism to recover to an incorrect state).

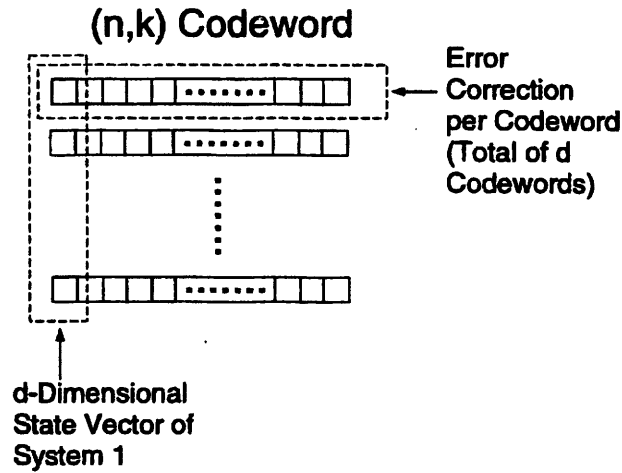


Figure 6-5: Encoded implementation of k LFSM's using n redundant LFSM's.

unreliable voters (so that a single failure in a component corrupts a single bit). Also note that, following Taylor's scheme, we actually need to have J replicas of each of the n redundant systems (a total of Jn systems). At the beginning of each time step, we allow these Jn systems to evolve to a (possibly corrupted) next state; at the end of the time step, we perform error detection and correction using *one* iteration of the modified iterative decoding scheme.

Once we allow failures in the error-correcting mechanism, we can no longer guarantee the invariant condition in eq. (6.3). However, given that there is no *overall failure*, the overall redundant state (i.e., the state of all Jn systems) at a certain time step correctly represents the state of the k underlying systems⁹. This means that with a fault-free iterative decoder we would be able to correctly recover the state of the k underlying redundant systems.

Before stating our main theorem, we summarize the setting in which it holds:

Consider k distinct instantiations of an LFSM with state evolution as in eq. (6.1), each instantiation with its own initial state and distinct input sequence. Embed these k instantiations into n redundant LFSM's (also with state evolution as in eq. (6.1)) using the approach described above. Each of these n redundant systems needs to be properly initialized (so that

⁹The overall state is an nd binary vector that represents kd bits of information. The n redundant systems perform without an overall failure for L time steps if their overall state at time step τ ($0 \leq \tau \leq L$) is within the set of nd vectors that correspond to the actual kd bits of information at that particular time step. In other words, if we had a fault-free (iterative) decoder, we would be able to obtain the correct states of the k underlying systems.

eq. (6.3) is satisfied for $\tau = 0$) and needs to be supplied with an appropriate input (encoded according to an (n, k) LDPC code). Furthermore, each of the n redundant systems needs to have J realizations (i.e., there is a total of Jn systems). A system realization uses its own set of reliable flip-flops and unreliable 2-input XOR gates. At the beginning of a time step, we allow all Jn redundant systems to evolve to a (possibly corrupted) next state. At the end of the time step, we use Gallager's modified iterative decoding scheme to correct any errors that may have taken place (each bit-copy is corrected using a separate set of $1 + (J-1)(K-1)$ unreliable 2-input XOR gates and one unreliable $(J-1)$ -bit voter).

Theorem 6.2 Assume that the 2-input XOR gates fail with probability p_x and the $(J-1)$ -bit voters fail with probability p_v . Let J be a fixed even integer greater than 4, let K be an integer greater than J , and let p be such that

$$p > \binom{J-1}{J/2} [(K-1)(2p+3p_x)]^{J/2} + p_v + p_x .$$

Then there exists a sequence of (n, k) LDPC codes (with $\frac{k}{n} \geq 1 - \frac{J}{K}$, with K "1's" in each row and J "1's" in each column of their parity check matrices) such that the probability of an overall failure at or before time step L is bounded above as follows:

$$\Pr[\text{overall failure at or before time step } L] < LdCk^{-\beta} ,$$

where β and C are constants given by

$$\beta = - \frac{\log \left\{ (J-1)(K-1) \binom{J-2}{J/2-1} [(K-1)(2p+3p_x)]^{J/2-1} \right\}}{2 \log[(J-1)(K-1)]} - 3 ,$$

$$C = \frac{J}{(1-J/K)^3} (2p+3p_x) \left[\frac{1}{2K} - \frac{1}{2J(K-1)} \right]^{-(\beta+3)} .$$

The code redundancy is $\frac{n}{k} \leq \frac{1}{1-J/K}$ and the hardware used (including the error-correcting mechanism) is bounded above by $\frac{Jd(3+(J-1)(K-1))}{1-J/K}$ XOR gates and by $\frac{Jd}{1-J/K}$ voters per system (where d is the system dimension).

J	K	p_x	p_v	p	β	C	Bound on $\Pr[\text{overall failure}]$
6	7	10^{-8}	10^{-8}	2.1×10^{-8}	0.5535	3.67	1
6	7	10^{-9}	10^{-9}	2.1×10^{-9}	1.2305	2.54	0.0062×10^{-4}
6	8	10^{-9}	10^{-9}	2.1×10^{-9}	0.9821	0.39	0.0519
6	9	10^{-9}	10^{-9}	2.1×10^{-9}	0.7836	0.14	0.4572
6	10	10^{-9}	10^{-9}	2.1×10^{-9}	0.6201	0.07	1
8	9	10^{-7}	10^{-7}	2.1×10^{-7}	0.6340	258	1
8	9	10^{-8}	10^{-8}	2.1×10^{-8}	1.4920	351	0.0126
8	9	10^{-9}	10^{-9}	2.1×10^{-9}	2.3501	477	1.69×10^{-8}

Table 6.1: Typical values for p , β and C given J , K , p_x and p_v . The bound on the probability of overall failure is shown for $d = 10$, $k = 10^7$ and $L = 10^5$.

Some illustrative values of β and C , and possible p , given p_x and p_v , can be seen in Table 6.1; the failure probabilities shown are for $d = 10$, $k = 10^7$ and $L = 10^5$.

Proof: The proof follows similar steps as the proofs in [106, 105]. In what follows, we give an overview of the proof. We describe it in more detail in Appendix B.

The state of the overall redundant implementation at a given time step τ (i.e., the states of the n redundant systems created by the embedding in eq. (6.2)) are fully captured by d codewords $C_i[t]$ of an (n, k) LDPC code ($1 \leq i \leq d$). More specifically, the state evolution equation of the n systems can be written as

$$\begin{bmatrix} C_1[t+1] \\ C_2[t+1] \\ \vdots \\ C_d[t+1] \end{bmatrix} = \mathbf{A}_c \begin{bmatrix} C_1[t] \\ C_2[t] \\ \vdots \\ C_d[t] \end{bmatrix} \oplus \mathbf{b} \left[\mathcal{X}[t] \right],$$

where $\mathcal{X}[t] = \begin{bmatrix} x_1[t] & x_2[t] & \dots & x_k[t] \end{bmatrix} \mathbf{G}^T$ is the encoding of the k inputs at time step t and \mathbf{A}_c , \mathbf{b} are the matrices in the state evolution equation (6.1).

Taylor showed that adding any two (n, k) codewords modulo-2 can be done reliably using LDPC codes and modified iterative decoding. Furthermore, he showed that we can reliably perform a *sequence* of L such additions by performing a component-wise XOR operation (using an array of n 2-input XOR gates) followed by one iteration of Gallager's modified

scheme (see the mechanism in Figure 6-3). More specifically, Taylor showed that

$$\Pr[\text{overall failure in a sequence of } L \text{ modulo-2 additions}] < LC'k^{-\beta'}$$

for constants C' and β' that depend on the probabilities of failure of the XOR gates and the voters, and on the parameters of the LDPC codes used.

From the discussion above, we see that we can use Taylor's scheme to perform error-correction in the d codewords from the (n, k) code. This requires, of course, that we maintain J copies of each codeword (a total of Jd codewords). During each time step, the overall redundant implementation calculates its new state (Jd new codewords) by adding modulo-2 the corresponding codewords of the current state; this is then followed by one iteration of error-correction based on Gallager's modified scheme.

Since matrix A_c is in canonical form, the computation of each codeword in the next overall state is based on at most two codewords of the current state (plus the input modulo-2). So over L time steps we essentially have d sequences of additions modulo-2 in the form that Taylor considered and which he showed can be protected efficiently via LDPC coding. Using the union bound, we conclude that

$$\Pr[\text{overall failure at or before time step } L] < LdCk^{-\beta}.$$

Note that the input is also something that we have to consider (and one of the reasons that our constants differ slightly from Taylor's) but it is not critical in the proof since the inputs involve no memory or error propagation.

The proof is discussed in more detail in Appendix B. □

6.4.3 Further Issues

Before we close, let us discuss a few issues regarding our approach:

- The bound on the probability of failure that we obtained in Theorem 6.2 goes down *polynomially* with the number of systems (not exponentially, as is the case for the distributed voting scheme and for Shannon's approach). It would be interesting to investigate whether this polynomial decay can be improved while maintaining constant

redundancy per system. Since error propagation does not necessarily constitute a failure, the bounds we have obtained are probably too conservative¹⁰ and it may be worthwhile exploring how they can be relaxed. Another interesting future direction would be to see whether other codes could be used.

- For simplicity, we have assumed that the only components that fail in our systems are the XOR gates. Our scheme can easily be adjusted to handle *transient* failures in other components (namely connections and/or flip-flops). We can also handle LFSM's with multiple inputs or LFSM's with the same dynamics (same A_c matrix) but different b vectors. Finding ways to handle permanent failures is also a very promising direction, but it would probably require the use of different techniques.
- We have not dealt in our proof with the encoding function or the cost associated with it. Encoding an LDPC code can be done in a straightforward way using matrix G : each of the n encoded bits can be generated using at most k information bits and at most $k-1$ 2-input XOR gates. Of course, the problem is that in our approach we let n and k increase. One possible solution is to encode using a binary tree of depth $\log k$ where each node performs a component-wise 2-input XOR operation. This requires $O(nk)$ 2-input XOR gates and $O(\log k)$ time; it can be done reliably using unreliable XOR gates if at the end of each stage of the tree evaluation we include one correcting iteration. This, however, means that our computation will be slowed down by $O(\log k)$ steps¹¹. Since our input encoding mechanism does not have to be "perfect" (in our proof we assumed so for simplicity), we could potentially exploit this flexibility to design encoding schemes that are perhaps faster or use less hardware than conventional ones; this could be a worthwhile future direction.
- The k LFSM instantiations that we are simultaneously protecting in Theorem 6.2 can be viewed as a larger original system S with state $\mathbf{q}_s[t] = \begin{bmatrix} \mathbf{q}_1^T[t] & \mathbf{q}_2^T[t] & \cdots & \mathbf{q}_k^T[t] \end{bmatrix}^T$,

¹⁰For example, Taylor's results were improved in [60, 32].

¹¹Similar slowdown is exhibited in Spielman's approach in [100].

input $\mathbf{x}[t] = \begin{bmatrix} x_1[t] & \dots & x_k[t] \end{bmatrix}$, and state evolution

$$\mathbf{q}_s[t+1] = \begin{bmatrix} \mathbf{A}_c & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_c & \mathbf{0} & & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_c & & \mathbf{0} \\ \vdots & & & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{A}_c \end{bmatrix} \mathbf{q}_s[t] \oplus \begin{bmatrix} \mathbf{b} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{b} & \mathbf{0} & & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{b} & & \mathbf{0} \\ \vdots & & & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{b} \end{bmatrix} \mathbf{x}[t],$$

Each \mathbf{A}_c is a $d \times d$ matrix and there are k such matrices along the diagonal of the overall system matrix; the structure of the input matrix is similar. The state vector $\mathbf{q}_s[\cdot]$ is a dk -dimensional vector that represents the states of all distinct instantiations of the system in eq. (6.1).

The overall *redundant* implementation that we construct is a system in the same form except that it has n matrices \mathbf{A}_c along the diagonal of its system matrix. Clearly, the redundant dynamics (in the sense of Chapter 4) of the overall redundant system are *not* zero. In fact, they are chosen to be exactly the same as the dynamics of the original systems. This allows us to create the redundant bits in a manner that does not require an excessive number of XOR gates. Generating the redundant bits using zero redundant dynamics would actually create problems in terms of the number of XOR gates that are needed, and would invalidate this approach¹². Therefore, this particular case illustrates a successful example of generating the redundant bits in an efficient manner.

- In a memoryless binary symmetric channel, a bit (“0” or “1”) is provided as input at the transmitting end; the output bit (at the receiving end) is the same as the input bit with probability $1 - p$ and flipped with probability p (p is known as the channel crossover probability). Errors between successive uses of the channel are independent. Shannon studied ways to encode k input bits into n redundant bits in order to achieve low probability of failure during transmissions. He showed that the probability of error can be made arbitrarily low using coding techniques, as long as

¹²This is actually one of the problems that Taylor had in [105]; see the discussions in [100, 83].

the rate $R = \frac{k}{n}$ of the code is less than the capacity of the channel, evaluated as $C = 1 + p \log p + (1 - p) \log(1 - p)$ (for the binary symmetric channel). Moreover, for rates R greater than C the probability of error per bit in the transmitted sequence can be arbitrarily large.

In our case, we have looked at embeddings of k distinct instantiations of a particular LFSM into n redundant systems, each of which is implemented using unreliable components. We have shown that, given certain conditions on the probabilities of component failures, there exist LDPC codes that will allow us to use n LFSM's to implement k identical LFSM's (that nevertheless operate on distinct input streams) and, *with non-zero "rate,"* achieve arbitrarily low probability of failure during any specified time interval. (In our context, "rate" means the amount of redundant hardware that is required *per machine instantiation*.) Specifically, by increasing n and k while keeping $\frac{k}{n} \geq 1 - \frac{J}{K}$, we can make the probability of an overall failure arbitrarily small. We have *not* shown that there is an upper bound on $\frac{k}{n}$ (which might then be called the *computational capacity*). It would be interesting to see if this can be done. We would probably have to consider a different approach since our goal would be to prove that for rates above a certain constant the probability of failure gets arbitrarily large, regardless of the coding/correcting scheme that we use.

- Ways of constructing fault-tolerant systems out of unreliable components have also been developed by Spielman in [100], and by Gács in [36]. In [100] the approach was for parallel systems that run on k "fine-grained" processors for L time steps. Spielman showed that the probability of error can go down as $O(Le^{-k^{1/4}})$ but the amount of redundancy is $O(k \log k)$ (i.e., $O(\log k)$ processors per system). Spielman also introduced the concept of *slowdown* due to the redundant implementation. Gács studied fault-tolerant cellular automata in [36], mostly in the context of stable memories. He employed cellular automata so that the cost/complexity of connectivity between different parts of the redundant implementation remain constant as the amount of redundancy increases. It would be interesting to investigate further how our methods relate to those of Spielman and Gács, and how these approaches may be generalized to arbitrary systems.

- Another interesting possibility is to apply our approach to signal processing or other special-purpose systems, such as linear digital filters. The latter generalization would have to deal with linear time-invariant systems operating on a larger (possibly infinite) set of elements and not on elements drawn from a finite field.

6.5 Summary

In this chapter we dealt with the problem of systematically providing fault tolerance to a dynamic system that is made of unreliable components, including unreliable components in the error-correcting mechanism. We considered components that suffer *transient* failures with constant probability. Initially, we employed modular redundancy with a distributed voting scheme, demonstrating that by increasing the amount of redundancy one can make the (bound on the) probability of failure during any given (finite) time interval as small as desired. We also constructed fault-tolerant LFSM's using failure-prone XOR gates and voters. More specifically, using linear codes with low decoding complexity, we obtained arrangements of n redundant LFSM's which reliably and efficiently perform the function of k identical LFSM's that are driven by distinct input sequences. At a *constant* cost per system we were able to guarantee any target probability of failure for a specified (finite) number of time steps. Our methods for error detection/correction are very simple and allow us to make connections with techniques used in graph and coding theory. They also demonstrate efficient ways to construct reliable dynamic systems exclusively using unreliable components (at least for LFSM's).

There are a number of interesting future directions, particularly in terms of applying our approach to more general dynamic systems, exploring further the use of alternative coding schemes with low decoding complexity (such as convolutional codes), and improving the bounds on the probability of failure. There are also interesting theoretical questions regarding how one can define the computational capacity of unreliable LFSM's and whether group machines are amenable to similar coding schemes as LFSM's. Since one concern about our approach is the increasing number of connections, it may be worthwhile exploring whether we can design dynamic systems that limit connections to neighboring elements (much like Gács approach in [36]). On the practical side, since our approach allows us to

construct dynamic systems largely out of unreliable components, it would be interesting to build and test such systems, either using silicon-based manufacturing technology (in order to evaluate any gains in terms of speed, power dissipation and reliability) or using novel technologies.

Chapter 7

Conclusions and Future Directions

In this thesis we developed a framework for protecting dynamic systems against failures that affect their state transition mechanism. Depending on the actual system implementation, these failures could be due to hardware malfunctions, programming errors, incorrect initialization, and so forth. Our approach replaces the original system with a larger, redundant dynamic system which is designed and implemented in a way that preserves the state evolution of the original system *and* imposes constraints on the set of states that are reachable under fault-free conditions. During each time step, the redundant system evolves to a possibly corrupted next state; then, a separate mechanism performs error detection and correction by identifying violations of the enforced state constraints and by taking appropriate correcting action. The redundant implementation together with the correcting mechanism become the fault-tolerant version of the original system.

The thesis systematically studied this two-stage approach to fault tolerance and demonstrated its potential and effectiveness. We directly applied coding techniques to dynamic systems and pointed out two issues of great importance: (i) *redundant dynamics* that can be used to efficiently/reliably enforce state constraints (for example, in order to build redundant implementations that require less hardware), and (ii) *error propagation* that is caused by failures in the error-correcting mechanism and complicates the task of maintaining the correct operation of a dynamic system for long time intervals. This second issue raises questions regarding the possibility and cost of constructing reliable dynamic systems using unreliable components.

We initially adopted the traditional assumption that the error detecting/correcting mechanism is fault-free, and focused on issues related exclusively to redundant dynamic systems. The overarching goal of Chapters 2 through 5 was to systematically develop schemes that can detect, identify and correct a fixed number of failures. We showed that a number of redundant implementations are possible under a particular error detection/correction scheme, and we precisely characterized these different redundant implementations for a variety of dynamic systems. Our techniques not only gave us insight and systematic ways of parameterizing redundant implementations, but also resulted in new schemes for fault tolerance, including parity check schemes with memory and reconfiguration methodologies. The specific classes of systems we considered are reviewed below.

Algebraic machines: Adopting a purely algebraic approach, we studied fault tolerance in group/semigroup machines and finite semiautomata. We showed how algebraic homomorphisms can be used to obtain redundant implementations, and exploited algebraic structure in order to facilitate error detection and correction. We made connections with algebraic machine decomposition and studied in more detail the construction of (separate and non-separate) redundant implementations. In particular, we demonstrated that certain decompositions for redundant implementations may be inappropriate because they detect failures unrelated to the system functionality we want to protect.

Further work on the role of machine decomposition and the use of non-separate redundancy schemes could bring our algebraic approach closer to making explicit connections with hardware (which is something we did not attempt in Chapter 2). For example, one possibility would be to study digital implementations that are based on failure-prone AND and OR gates and flip-flops. Another interesting future direction is to investigate whether fault tolerance can be achieved through a combination of our techniques and the techniques for dynamical systems and codes over finite abelian groups, [34, 16].

Linear time-invariant (LTI) dynamic systems: We constructed fault-tolerant LTI dynamic systems using *linear* codes. More specifically, given an LTI dynamic system that we needed to protect against state transition failures, we applied linear block-coding

techniques and obtained a complete characterization of the class of appropriate redundant implementations. Our approach generalized previous work based on modular redundancy or simple checksum schemes by allowing redundant modes to be non-zero and/or be coupled with the original system.

We made explicit connections with hardware and constructed our redundant LTI dynamic systems out of delays, adders and gain elements. Our design ensured that failures in a single component (adder or multiplier) corrupted only a single state variable. This allowed us to make use of linear codes to protect our systems against a fixed number of errors in the redundant state variables. Our examples exploited the possibilities that exist when constructing redundant implementations (e.g., in order to minimize hardware, to verify parity checks periodically, or to perform reconfiguration when dealing with permanent failures).

Future work can focus on applying these ideas to LTI dynamic systems of interest (e.g., linear filters in digital signal processing applications). We should also focus on systematically studying optimization criteria (e.g., minimizing hardware redundancy) and on finding the “best” redundant implementation for a particular decoding/encoding scheme. One can also study how these ideas generalize to nonlinear and/or time-varying systems.

Linear finite-state machines (LFSM's): We followed an approach similar to the one for LTI dynamic systems and showed that, under a particular linear block-coding scheme, there are many ways of constructing redundant implementations for an LFSM. We exploited this flexibility in order to minimize the number of XOR gates that are required in a redundant implementation. It would be interesting to see whether these techniques can be generalized to other “linear” systems, such as linear systems over rings or semirings (e.g., max-plus systems).

Petri nets: When we applied our approach to Petri net models of discrete event systems (DES's), we obtained monitoring schemes for complex networked systems, such as manufacturing systems, network and communication protocols, or power systems. The algebraic construction resembles that used in our other classes of dynamic systems: we are essentially trying to build a redundant Petri net with a state (marking) that

is restricted in a way that allows us to identify failures by detecting violations of the enforced state constraints. The tradeoffs and objectives involved in DES's, however, may be completely different. For example, the goal may be to avoid complicated reachability analysis, or to minimize the size of the monitor, or to construct systems that require minimal communication overhead.

The monitoring mechanisms that are obtained using our approach are simple and allow us to easily choose additional places, connections and weights so that monitoring of both transition and place failures can be verified by weighted checksums on the overall state of the redundant Petri net. The monitoring schemes can be designed to operate reliably despite erroneous/incomplete information from certain places, or to minimize the communication and hardware overhead that occurs due to the additional connections and acknowledgments. The thesis did not address such optimization issues explicitly, but this is a very interesting direction for future research.

Other future extensions include the investigation of additional error models and the development of hierarchical or distributed error detection and correction schemes. Another interesting possibility would be the development of robust control schemes for DES's that are observed at a central controller through a network of remote and unreliable sensors. In such cases we can use Petri net embeddings to design redundant sensor networks and to devise control strategies that achieve the desired objectives, despite the presence of uncertainty (in the sensors themselves or in the information received by the controller). The development of more general redundant embeddings and the explicit study of examples where a subset of the transitions is uncontrollable and/or unobservable (i.e., where links to or from these transitions are not possible) is also a very appealing direction. Finally, in order to achieve error-correction we should introduce *failure recovery transitions* that can serve as correcting mechanisms when failures are detected.

In the thesis we also relaxed the assumption that error-correction be fault-free, and looked at ways to cope with failures in the error-correcting mechanism. We considered transient failures that generate transitions to incorrect states at a particular time step, but whose physical causes disappear at later time steps. Transient failures in the error

corrector propagate in time, due to the dynamic evolution of our systems (the new state is partially based on the internal state of our systems); this results in a serious increase in the probability of failure, particularly in the case of dynamic systems that operate over long time intervals.

In order to handle error propagation effectively, we introduced modular redundancy schemes that use multiple system replicas and voters. By increasing the amount of redundancy, we can construct redundant implementations that operate under a specified (low) level of failure probability for any specified finite time interval. (Equivalently, we can construct redundant implementations that achieve arbitrarily small probability of failure for a specified time interval.) By combining these schemes with low-complexity error-correcting codes, we obtained interconnections of identical LFSM's that operate in parallel on distinct input sequences, fail with arbitrarily low probability during a specified finite time interval, and require only a constant amount of redundancy per machine. Our approach is quite comprehensive, allows us to construct reliable dynamic systems largely out of unreliable components, and in principle may enable the use of less expensive manufacturing technologies that can perhaps operate at faster speeds or in more hostile environments (e.g., high temperatures, space missions). It would be worthwhile to explore these possibilities further in order to construct novel computer architectures and employ new manufacturing technologies.

There are a number of additional future directions, particularly in terms of generalizing our approach to arbitrary finite-state machines, evaluating the possibility of using other (easily decodable) coding schemes and finding out more about the system structure that allows us to perform parallel simulations in an efficient manner. There are also interesting theoretical questions regarding how the bounds can be improved and whether we can define the computational capacity of unreliable LFSM's.

It may also be helpful to study generalizations of the approach suggested in this thesis. We have operated under the premise that the code (constraints) enforced on the state of the redundant implementation are time-independent; this means that the error-correcting mechanism has no memory (it does not remember any of the previous states or inputs). The thesis has shown that this approach can be used effectively and efficiently to protect the evolution of dynamic systems; it would be interesting, however, to investigate the appli-

cability of other, more general approaches. For example, instead of using block codes, one could try convolutional codes¹ to protect LFSM's. This approach appears promising since convolutional codes can also be decoded with low cost. In addition, using error-correcting mechanisms with memory may lead to reduced hardware complexity in our fault-tolerant implementations.

The bounds that we have obtained on the probability of failure increase linearly with the number of time steps for which our fault-tolerant constructions operate. An alternative to our approach that could potentially alleviate this problem would be to let the systems evolve for several time steps and then use an error-correction mechanism that looks at their overall state evolution (sequence of states) to decide what corrections to make. Of course, this would require a more sophisticated error detecting/correcting mechanism, which could potentially be subject to more failures than a simple voter.

Other interesting future directions include the investigation of error-correction techniques capable of handling permanent failures, the use of redundancy to guarantee robust performance in DES's and the construction of simple and effective monitoring schemes that are based on statistical observations (e.g., periodic behavior). It would also be interesting to connect our approach with techniques used to test and verify sequential circuits [62], or to enable systematic reconfiguration.

¹Some related work has appeared in [89]; the implicit assumption in this case is that error detection/correction is fault-free.

Appendix A

Conditions for Single-Error Detection and Correction

A.1 Semigroup Computations

Here, we derive necessary and sufficient conditions for error detection and correction in fault-tolerant semigroup computations using the approach described in Chapter 2 (Section 2.2). Throughout our analysis we assume that the error detecting/correcting unit is fault-free¹.

We assume that the set of possible failures $F = \{f_1, f_2, f_3, \dots\}$ is finite and that the faulty output due to one or more failures depends only on the fault-free result, i.e., the faulty result is independent of the actual pair of operands involved, but only depends on the semigroup *product* of the operands. Note that a failure can always be modeled this way: when a physical failure produces different erroneous results for pairs of operands that ideally result in the same semigroup product, we can use multiple f_i 's, each of which captures the effect of this physical failure for a particular pair of operands. We assume that the erroneous result reached due to the occurrence of failure f_i when operating on $\phi(s_1)$ and $\phi(s_2)$ lies in H , and is given by $\rho_{f_i} = e(\rho, f_i)$, where $\rho = \phi(s_1) \diamond \phi(s_2)$. We define the error model for k simultaneous failures similarly: the effect of a multiple failure $f_i^{(k)} = (f^1, f^2, \dots, f^k)$

¹As mentioned earlier, this is a reasonable assumption when the error detecting/correcting mechanism is simpler than the operation we are trying to protect. Another justification for this assumption is that the output stage of a fault-tolerant system *has* to be fault-free (otherwise failures in it would definitely cause incorrect outputs regardless of the redundancy).

(where $f^j \in F$, $1 \leq j \leq k$, and failures are ordered in the k -tuple according to their times of occurrence) is captured by the mapping $e^{(k)}(\rho, f_i^{(k)})$. We denote the set of possible multiple failures by $F^{(k)} = \{(f^1, f^2, \dots, f^k) \mid f^i \in F, 1 \leq i \leq k\}$.

Since the detector/corrector α in Figure 1-1 bases its decisions solely on the possibly faulty output ρ_f , the computation in the redundant semigroup H needs to meet the following condition for full single-error detection:

$$e(\rho_1, f_i) \neq \rho_2 \text{ for all } f_i \in F \text{ and all } \rho_1, \rho_2 \in S' \text{ such that } \rho_1 \neq \rho_2 .$$

Here, $S' = \phi(S) \subset H$ is the subset of valid results; an error is detected whenever the result ρ_f lies outside S' . To be able to correct *single* errors, we need the *additional* condition

$$e(\rho_1, F) \cap e(\rho_2, F) = \emptyset \text{ for all } \rho_1, \rho_2 \in S' \text{ such that } \rho_1 \neq \rho_2 ,$$

where $e(\rho, F) = \{e(\rho, f_i) \mid f_i \in F\}$. The error can be corrected by identifying the unique set $e(\rho_k, F)$ in which the erroneous result ρ_f lies; ρ_k is then the correct result. The above condition essentially establishes that no two *different* results ρ_1 and ρ_2 can be mapped, perhaps by different failures, to the same faulty result.

These conditions can be generalized for fully detecting up to d errors and correcting up to c errors ($c \leq d$); one can easily show that the following conditions are necessary and sufficient:

$$\begin{aligned} e^{(k)}(\rho_1, F^{(k)}) \cap S' &\subset \{\rho_1\} && \text{for all } \rho_1 \in S', \text{ and for } 1 \leq k \leq d , \\ \left\{ \bigcup_{k=1}^d e^{(k)}(\rho_1, F^{(k)}) \right\} \cap e^{(j)}(\rho_2, F^{(j)}) &= \emptyset && \text{for all } \rho_1, \rho_2 \in S', \rho_1 \neq \rho_2, \\ &&& \text{and for } 1 \leq j \leq c . \end{aligned}$$

Here, $e^{(k)}(\rho, F^{(k)})$ denotes the set $\{e^{(k)}(\rho, f_i^{(k)}) \mid f_i^{(k)} \in F^{(k)}\}$. The first condition guarantees detection of any combination of d or less failures (because no k failures, $k \leq d$, can cause the erroneous result $e^{(k)}(\rho_1, f^{(k)})$ to be a valid one — unless it is the correct result ρ_1). The second condition guarantees correction of up to c failures (no combination of up to c failures

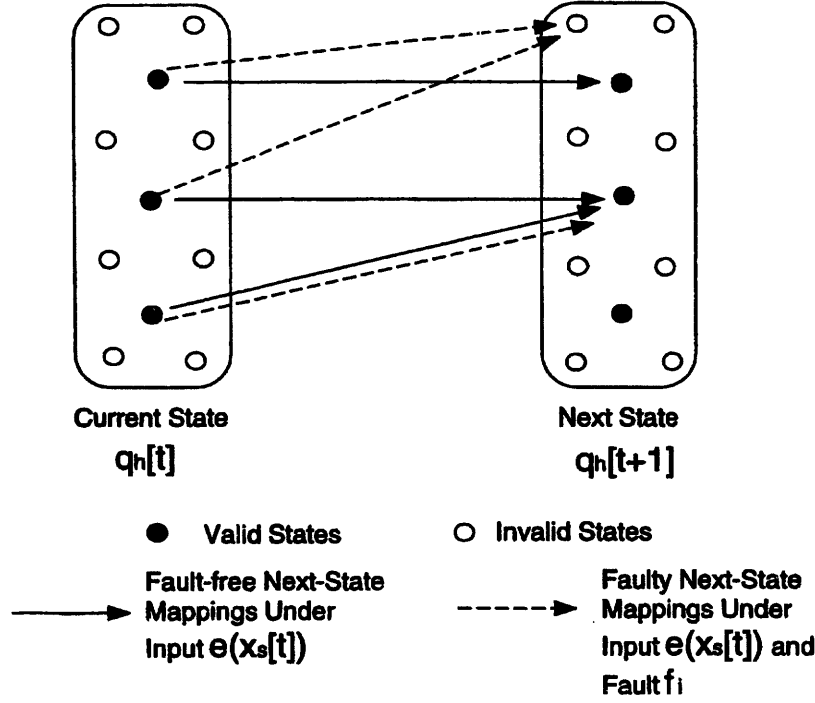


Figure A-1: Conditions for single-error detection in a finite semiautomaton.

on ρ_2 can result in an erroneous value also produced by up to d failures on a *different* result ρ_1).

A.2 Finite Semiautomata

In this section we derive the conditions that are necessary and sufficient for single-error detection and correction in a redundant implementation \mathcal{H} of a dynamic system \mathcal{S} (see Chapter 1 for definitions). We assume fault-free error-correction, which implies that dynamic system \mathcal{H} is in the correct state at the beginning of each time step.

Let $F = \{f_1, f_2, f_3, \dots\}$ be the set of possible state transition failures and assume that the erroneous next state due to failure f_i is described by $q_h^{f_i}[t+1] = \delta_{\mathcal{H}}^{f_i}(q_h[t], e(x_s[t]))$ defined for all states (valid or not) $q_h[t] \in Q_{\mathcal{H}}$ and all inputs $x_s[t] \in X_{\mathcal{S}}$ ($\delta_{\mathcal{H}}^{f_i}$ is assumed independent of t).

The following condition is necessary and sufficient for single-error detection at the end of time step t : for all $f_i \in F$, all $x_s[t] \in X_{\mathcal{S}}$, all $q_h'[t] \in Q'_{\mathcal{H}}$ and all $q_h''[t+1] \in Q'_{\mathcal{H}}$ such that

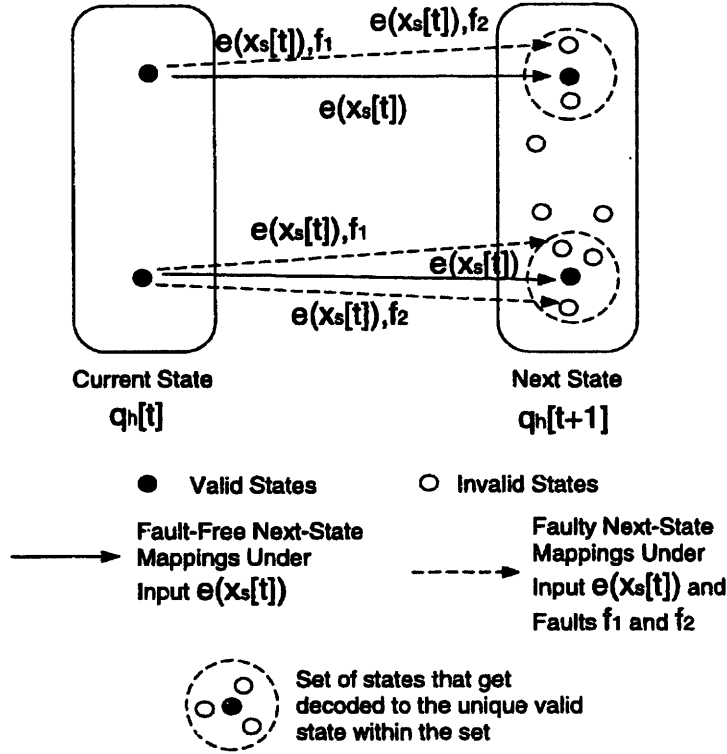


Figure A-2: Conditions for single-error correction in a finite semiautomaton.

$q_h''[t+1] \neq \delta_{\mathcal{H}}(q_h'[t], e(x_s[t]))$, we require that

$$\delta_{\mathcal{H}}^{f_i}(q_h'[t], e(x_s[t])) \neq q_h''[t+1] .$$

An illustration of this condition for a particular input $x_s[t] \in X_S$ and a particular failure $f_i \in F$ is given in Figure A-1. The fault-free next-state mappings are described by the solid lines; failure f_i is detectable by a fault-free detector because it causes transitions to invalid states (shown by dotted lines). When f_i and $x_s[t]$ do result in a valid state (within $Q_{\mathcal{H}}'$), the transition is the correct one (f_i is undetectable under input $x_s[t]$, but does not cause an erroneous transition — f_i may be less benign under different inputs).

Similarly, for concurrent correction of a single error, we need the *additional* condition: for all $q_h'[t] \in Q_{\mathcal{H}}'$, all $x_s[t] \in X_S$ and all $q_h''[t+1] \in Q_{\mathcal{H}}''$ such that $q_h''[t] \neq \delta_{\mathcal{H}}(q_h'[t], e(x_s[t]))$

$$\delta_{\mathcal{H}}^F(q_h'[t], e(x_s[t])) \cap \alpha^{-1}(q_h''[t+1]) = \emptyset ,$$

where

- $\delta_{\mathcal{H}}^F(q'_h[t], e(x_s[t]))$ denotes the set $\{\delta_{\mathcal{H}}^{f_i}(q'_h[t], e(x_s[t])) \mid \text{for all } f_i \in F\}$,
- $\alpha^{-1}(q''_h[t+1]) \subset Q_{\mathcal{H}}$ is the set of states in \mathcal{H} that get corrected to state $q''_h[t+1]$ (by the error-correcting mechanism).

An illustration of the necessary conditions for single-error correction is shown in Figure A-2 for a set of two failures $F = \{f_1, f_2\}$ and one input $x_s[t]$: the solid lines denote the correct transitions under the encoded input $e(x_s[t])$ for each of two states; the dotted lines denote incorrect transitions under failures f_1, f_2 .

These conditions can be generalized to detection and correction of multiple failures.

Appendix B

Proof of Theorem 6.2

In this appendix we discuss the proof of Theorem 6.2 in more detail. The proof follows the steps in [106, 105].

Assume that our overall redundant implementation starts operation at time step 0. As described in Section 6.4, during each time step, we first allow the Jn redundant systems to evolve to their corresponding (possibly corrupted) next state, and we then perform *one* iteration of Gallager's modified decoding scheme. We do this in parallel for all d (n, k) codewords, each of which has J copies — see Figure 6-5.

The construction of our low-density parity check (LDPC) coding scheme guarantees that for the first m time steps the parity checks that are involved in correcting a *particular* bit-copy are in error with *independent* probabilities (recall that each bit d_i has J copies $d_i^1, d_i^2, \dots, d_i^J$, and that m denotes the number of independent iterations of the LDPC code at hand). The reason for this independence is that the errors within these parity check sets are generated in different systems (failures in different components are, of course, statistically independent).

After the first m time steps, the independence condition in the parity checks will be true given that certain conditions are satisfied. More specifically, if we can ensure that no component failure influences decisions for m or more consecutive time steps (causing a bit-copy to be incorrect m or more time steps in the future), then we can guarantee that the $J-1$ parity checks for a particular bit-copy are in error with independent probabilities. We make this more precise with the following definition:

Definition B.1 *A propagation failure occurs whenever any of the Jn bit-copies in the overall redundant implementation is erroneous due to component failures that occurred more than m time steps in the past.*

Definition B.2 *The initial propagation failure denotes the first propagation failure that takes place, i.e., the occurrence of the first component failure that propagates for $m+1$ time steps in the future.*

We will show that a propagation failure is very unlikely and that in most cases the bit errors in the Jd codewords will depend only on component failures that occurred within the last few time steps. We calculate a bound on the probability of a propagation failure in Section B.3. To do that, we use a bound on the probability of error per bit-copy which we establish in the next section.

B.1 “Steady-State” Under No Initial Propagation Failure

For now, we concentrate on the first m time steps (during which the overall redundant implementation is guaranteed to operate under *no* propagation failure) and obtain an upper bound on the probability of error per bit-copy. More specifically, we show that the probability of error per bit-copy at the end of time step τ ($0 \leq \tau \leq m$) is bounded by a constant, i.e., we show that

$$\Pr[\text{error per bit-copy at end of time step } \tau] \leq p.$$

This assumption is certainly true at $\tau = 0$ (at time step 0 the probability of error per bit-copy is zero). It remains true as long as (i) $\tau \leq m$ (eventually we will show that it remains true as long as no propagation failure has taken place), and (ii) $\left(\frac{J-1}{J/2} \right) [(K-1)(2p + 3p_x)]^{J/2} + p_v + p_x$ is smaller than p . To see why this is the case, consider the following:

- In order to calculate a certain bit-copy in its next-state vector, each of the Jn redundant systems uses at most two bits (bit-copies) from its previous state vector and performs at most two 2-input XOR operations (one XOR-ing involves the two bit-copies in the previous state vector, the other one involves the input). Using the

union bound the probability of error per bit-copy once state evolution is performed is bounded above by

$$\Pr[\text{error per bit-copy after state evolution at time step } \tau] \leq 2p + 2p_x \equiv q.$$

This is simply the union bound of the events that any of the two previous bit-copies is erroneous and/or that any of the two XOR operations fails (for simplicity we assume that the input provided is correct). Note that we do not assume independence here.

- Once all Jn systems transition to their next states, we perform error-correction along the Jd codewords (see Figure 6-5, there are J copies for each of the $d(n, k)$ codewords). The correction consists of *one* iteration of Gallager's modified decoding scheme. Recall that each bit-copy is corrected using $J-1$ parity checks, each of which involves $K-1$ other bit-copies. We will say that a parity check associated with a particular bit-copy d_i^j ($1 \leq j \leq J$) is *in error* if bit-copy d_i^j is incorrect but the parity check is "0", or if d_i^j is correct but the parity check is "1" (this is because ideally we would like parity checks to be "0" if their corresponding bit-copy is correct and to be "1" if the bit-copy is incorrect). Note that this definition decouples the probability of a parity check being in error with whether or not the associated bit-copy is erroneous. The probability of an error in the calculation of a parity check (see the error-correcting mechanism in Figure 6-3) is bounded by

$$\Pr[\text{parity check in error}] \leq (K-1)(q + p_x) = (K-1)(2p + 3p_x)$$

(i.e., a parity check for a particular bit-copy is in error if there is an error in any of the $K-1$ *other* bit-copies or a failure in any of the $K-1$ XOR operations).

- A particular bit-copy will *not* be corrected if one or more of the following three events happen: (i) $J/2$ or more of the associated $J-1$ parity checks are in error, (ii) there is a failure in the voting mechanism, or (iii) there is a failure in the XOR gate that receives the voter output as input (see Figure 6-3). If the parity checks associated

with a particular bit-copy are in error with independent probabilities, then

$$\begin{aligned} \Pr[\text{error per bit-copy after correction}] &\leq \binom{J-1}{J/2} [(K-1)(2p+3p_x)]^{J/2} + \\ &\quad + p_v + p_x \tag{B.1} \\ &\leq p. \end{aligned}$$

We conclude that if the parity checks for each bit-copy are in error with independent probabilities, we will end up with a probability of error per bit-copy that satisfies

$$\Pr[\text{error per bit-copy at end of time step } \tau] \leq p.$$

Therefore, p can be viewed as a bound on the “steady-state” probability of error per bit-copy at the end/beginning of each time step (at least up to time step m).

What we show next is that the probability of error per bit-copy remains bounded by p for $\tau > m$ under the condition that no propagation failure has taken place. In Section B.3 we bound the probability of *initial* propagation failure (using p as the probability of error per bit-copy); this leads to a bound on the probability of an overall failure in our system.

B.2 Conditional Probabilities Given No Initial Propagation Failure

So far we have established that the probability of error per bit-copy will satisfy

$$\Pr[\text{error per bit-copy at end of time step } \tau] \leq p$$

for $0 \leq \tau \leq m$. This “steady-state” probability of error per bit-copy remains valid for $\tau > m$ as long as the initial propagation failure does not take place. The only problem is that once we condition on the event that no propagation failure has taken place, the probability of error per bit-copy may not necessarily be bounded by p . What we do in this section is show that this assumption remains true. The proof is a direct consequence of our definition of a propagation failure.

At the end of time step $\tau = m$, the probability of error per bit-copy is bounded by p . However, in order to derive the same bound for the probability of error per bit-copy at the end of time step $\tau = m + 1$, we have to assume that different parity checks for a particular bit-copy are in error independently. To ensure this, it is enough to require that no component failure which took place at time step $\tau = 0$ has propagated all the way up to time step m (so that it causes a propagation failure at time step $\tau = m + 1$).

We will show that the probability that a particular bit-copy d_i^j ($1 \leq j \leq J$) is in error at the end of time step $\tau = m$ *given* that no propagation failure (PF) takes place at time step $\tau = m$, denoted by

$$\Pr[\text{error per bit-copy at the end of time step } \tau = m \mid \text{no initial PF at } \tau = m],$$

is smaller or equal to the “steady-state” probability of error per bit-copy (i.e., smaller than p).

Consider the set of primitive events (i.e., patterns of component failures at time steps $\tau = 0, 1, \dots, m$) that cause bit-copy d_i^j to be erroneous at the end of time step m . Call this set of events A (i.e., $\Pr(A) = \Pr[d_i^j \text{ is erroneous at end of time step } \tau = m]$). Clearly, from our discussion in the previous section we know that $\Pr(A) \leq p$. Let B denote the set of primitive events (patterns of component failures at time steps $\tau = 0, 1, \dots, m$) that lead to a propagation failure at bit-copy d_i^j . Note that *by definition* set B is a subset of A ($B \subset A$) because a propagation failure at time step $\tau = m$ has to corrupt bit d_i^j . Therefore,

$$\begin{aligned} \Pr[d_i^j \text{ is erroneous at end of } \tau = m \mid \text{no PF at } d_i^j \text{ at time } m] &= \frac{\Pr(A) - \Pr(B)}{1 - \Pr(B)} \\ &\leq \Pr(A) \leq p. \end{aligned}$$

We conclude that the “steady-state” probability of error per bit-copy remains bounded by p given that no propagation failure takes place. Note that we actually condition on the event that “no propagation failure takes place in *any* of the bit-copies at time step m ”, which is different than event B . The proof goes through in exactly the same way because a pattern of component failures that causes a propagation failure to a different bit-copy can either cause an error in the computation of bit-copy d_i^j or not interfere with it at all.

B.3 Bounds on the Probabilities of Failures

Recall that our analyses in the previous sections hold as long as there has not been a *propagation failure* (otherwise, eq. (B.1) is not valid because errors in parity checks are dependent). What we do next is bound the probability of the initial propagation failure. This gives us an analytically tractable way of guaranteeing the validity of the probabilities we derived in the previous sections and eventually obtaining a bound on the probability of an overall failure in our system.

B.3.1 Bounding the Probability of Initial Propagation Failure

Given that no propagation failure has taken place up to time step τ , we have a bound on the probability of error per bit-copy and we know that different parity checks for any given bit-copy are in error with independent probabilities. Using this we can focus on a particular bit-copy d_i^j ($1 \leq j \leq J$) and calculate the probability that a component failure which took place at time step $\tau - m$ propagates up to time step τ (so that it causes the initial propagation failure). We call this the “probability of initial propagation failure at bit-copy d_i^j ”.

Note that in order for a component failure to propagate for m time steps it is necessary that it was *critical* in causing a wrong decision during the correcting stage of m consecutive time steps. In other words, without this particular component failure the decision/correction for all of these time steps would have had the desired effect.

Assume that no propagation failure has taken place up to time step τ (where $\tau \geq m$). Let us denote by P_m the probability that a component failure has propagated for m consecutive time steps in a way that causes the value of bit-copy d_i^j at time step τ to be incorrect. In order for this situation to happen, we require *both* of the following two *independent* conditions:

1. The value of one or more of the $(J-1)(K-1)$ bit-copies involved in the parity checks of bit-copy d_i^j is incorrect because of a component failure that has propagated for $m-1$ time steps. Since each such bit-copy was generated during the state evolution stage of time step τ based on *at most two* bit-copies from the previous state vector (the input

bit is irrelevant in error propagation), the probability of this event is bounded by

$$(J-1)(K-1)2P_{m-1} ,$$

where P_{m-1} is the probability that a component failure has propagated for $m-1$ consecutive time steps (causing an incorrect value to one of the bit-copies used in the parity checks for d_i^j). The factor of two comes in because the failure that propagates for $m-1$ time steps could be in any of the at most two bit-copies used to generate d_i^j during the state evolution stage. This is due to the fact that the system matrix A_c in eq. (6.1) is in standard canonical form.

2. There were at least $J/2-1$ erroneous parity checks (among the $J-2$ *remaining*¹ parity checks). The probability of this event is bounded by

$$\binom{J-2}{J/2-1} [(K-1)(q+p_x)]^{J/2-1} .$$

Since no propagation failure has taken place yet, errors in parity checks are independent. Therefore, the probability of a failure propagating for m consecutive time steps is bounded by

$$P_m \leq (J-1)(K-1)2P_{m-1} \binom{J-2}{J/2-1} [(K-1)(q+p_x)]^{J/2-1} .$$

Similarly,

$$P_{m-1} \leq (J-1)(K-1)2P_{m-2} \binom{J-2}{J/2-1} [(K-1)(q+p_x)]^{J/2-1} ,$$

¹One parity check is associated with the failure that has propagated for $m-1$ time steps.

and so forth. We conclude that

$$\begin{aligned} P_m &\leq (q + p_x) \left[(J-1)(K-1) 2^{\binom{J-2}{J/2-1}} [(K-1)(q + p_x)]^{J/2-1} \right]^m \\ &\leq (q + p_x) 2^m \left[(J-1)(K-1) \binom{J-2}{J/2-1} [(K-1)(q + p_x)]^{J/2-1} \right]^m. \end{aligned}$$

The union bound can be used to obtain an upper bound on the probability that the initial propagation failure takes place at time step τ . We need to have a pattern of component failures that propagates for m time steps in at least one of the Jnd bit-copies of our redundant construction, i.e.,

$$\Pr[\text{initial prop. failure at time step } \tau] \leq JndP_m.$$

Our LDPC codes were constructed so that the following conditions are satisfied:

$$\begin{aligned} m &\geq \frac{\log n + \log \frac{KJ-K-J}{2KJ(K-1)}}{2 \log[(J-1)(K-1)]} \equiv A(n), \\ m &\leq \frac{\log n}{\log[(J-1)(K-1)]}, \\ n &\leq \frac{k}{1 - J/K}. \end{aligned}$$

(See [37, 104] for more details.)

Using the first inequality we obtain the bound

$$\begin{aligned} \Pr[\text{initial prop. failure at } \tau] &\leq Jnd(q + p_x)2^m \\ &\quad \left[(J-1)(K-1) \binom{J-2}{J/2-1} [(K-1)(q + p_x)]^{J/2-1} \right]^{A(n)} \\ &\leq Jnd(q + p_x)2^m \left\{ \left[\frac{1}{2K} - \frac{1}{2J(K-1)} \right] n \right\}^{-\beta'}, \end{aligned}$$

where β' is given by

$$\beta' \equiv -\frac{\log \left\{ (J-1)(K-1) \binom{J-2}{J/2-1} [(K-1)(q+p_x)]^{J/2-1} \right\}}{2 \log[(J-1)(K-1)]}.$$

Since $k \leq n$ and $n \leq \frac{k}{1-J/K}$, we get

$$\begin{aligned} \Pr[\text{initial prop. failure at } \tau] &\leq J \left\{ \frac{k}{1-J/K} \right\} d (q+p_x) 2^m \\ &\quad \left\{ \left[\frac{1}{2K} - \frac{1}{2J(K-1)} \right] k \right\}^{-\beta'}. \end{aligned}$$

Since $m \leq \log n$ (because $K > J \geq 4$), we obtain the bound

$$2^m \leq 2^{\log n} \leq n \leq \frac{k}{1-J/K},$$

which leads to

$$\Pr[\text{initial prop. failure at } \tau] \leq dC'k^{-\beta'+2},$$

where

$$C' \equiv (q+p_x) \frac{J}{(1-J/K)^2} \left[\frac{1}{2K} - \frac{1}{2J(K-1)} \right]^{-\beta'}.$$

B.3.2 Bounding the Probability of Overall Failure

Let us assume that no propagation failure has taken place in the time interval $0 \leq t < \tau$. We want to find out an upper bound on the probability that the initial propagation failure takes place at time step τ . This means that if we were to apply a *fault-free* iterative decoding scheme at the state of our systems (all Jd codewords) at time step τ , then no failure would be critical in causing consecutive erroneous decisions in all m decoding iterations that may follow.

An upper bound on the probability that the initial propagation or decoding failure takes

place is given by

$$\Pr[\text{overall failure at time step } \tau] < mdC'k^{-\beta'+2},$$

or, since $m \leq \log n \leq n \leq \frac{k}{1-J/K}$,

$$\Pr[\text{overall failure at time step } \tau] < dCk^{-\beta},$$

where

$$\beta \equiv \beta' - 3 \equiv -\frac{\log \left\{ (J-1)(K-1) \binom{J-2}{J/2-1} [(K-1)(q+p_x)]^{J/2-1} \right\}}{2 \log[(J-1)(K-1)]} - 3,$$

$$C \equiv \frac{C'}{1-J/K} \equiv (q+p_x) \frac{J}{(1-J/K)^3} \left[\frac{1}{2K} - \frac{1}{2J(K-1)} \right]^{-\beta'}.$$

Using the union bound we see that the probability of an overall failure at or before time step L is bounded by

$$\Pr[\text{overall failure at or before time step } L] < LdCk^{-\beta}.$$

Bibliography

- [1] J. A. Abraham, P. Banerjee, C.-Y. Chen, W. K. Fuchs, S.-Y. Kuo, and A. L. N. Reddy. Fault tolerance techniques for systolic arrays. *IEEE Computer*, pages 65–75, July 1987.
- [2] M. A. Arbib, editor. *Algebraic Theory of Machines, Languages, and Semigroups*. Academic Press, New York, 1968.
- [3] M. A. Arbib. *Theories of Abstract Automata*. Prentice-Hall, Englewood Cliffs, New Jersey, 1969.
- [4] P. Ashar, S. Devadas, and A. R. Newton. Optimum and heuristic algorithms for an approach to finite state machine decomposition. *IEEE Transactions on Computer-Aided Design*, 10:296–310, March 1991.
- [5] A. Avizienis. Toward systematic design of fault-tolerant systems. *IEEE Computer*, pages 51–58, April 1997.
- [6] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin. The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. In *Proceedings of the 1st Int. Conf. on Fault-Tolerant Computing*, pages 1312–1321, 1971.
- [7] F. Baccelli, G. Cohen, G. J. Olsder, and J. P. Quadrat. *Synchronization and Linearity*. Wiley, New York, 1992.
- [8] P. E. Beckmann. Fault-Tolerant Computation Using Algebraic Homomorphisms. PhD thesis, EECS Department, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1992.
- [9] P. E. Beckmann and B. R. Musicus. A group-theoretic framework for fault-tolerant computation. In *Proceedings of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, volume V, pages 557–560, 1992.
- [10] P. E. Beckmann and B. R. Musicus. Fast fault-tolerant digital convolution using a polynomial residue number system. *IEEE Transactions on Signal Processing*, 41:2300–2313, July 1993.
- [11] R. E. Blahut. *Theory and Practice of Data Transmission Codes*. Addison-Wesley, Reading, Massachusetts, 1983.

- [12] W. G. Bliss and M. R. Lightner. The reliability of large arrays for matrix multiplication with algorithm-based fault tolerance. In M. Sami and F. Distanté, editors, *Proceedings of Wafer Scale Integration III*, pages 305–316, 1990.
- [13] T. L. Booth. *Sequential Machines and Automata Theory*. Wiley, New York, 1968.
- [14] J. W. Brewer, J. W. Bunce, and F. S. Van Vleck. *Linear Systems Over Commutative Rings*, volume 104 of *Lecture Notes in Pure and Applied Mathematics*. Marcel Dekker Inc., New York, 1986.
- [15] R. W. Brockett and A. S. Willsky. Finite group homomorphic sequential systems. *IEEE Transactions on Automatic Control*, AC-17:483–490, August 1972.
- [16] G. Caire and E. Biglieri. Linear block codes over cyclic groups. *IEEE Transactions on Information Theory*, 41:1246–1256, September 1995.
- [17] C. G. Cassandras. *Discrete Event Systems*. Aksen Associates, Boston, 1993.
- [18] C. G. Cassandras, S. Lafortune, and G. J. Olsder. *Trends in Control: A European Perspective*. Springer-Verlag, London, 1995.
- [19] K. Cattell and J. C. Muzio. Analysis of one-dimensional linear hybrid cellular automata over $GF(q)$. *IEEE Transactions on Computers*, 45:782–792, July 1996.
- [20] S. Chakraborty, D. R. Chowdhury, and P. P. Chaudhuri. Theory and application of nongroup cellular automata for synthesis of easily testable finite state machines. *IEEE Transactions on Computers*, 45:769–781, July 1996.
- [21] A. Chatterjee. Concurrent error detection in linear analog and switched-capacitor state variable systems using continuous checksums. In *Proceedings of Int. Test Conference*, pages 582–591, 1991.
- [22] A. Chatterjee and M. d’Abreu. The design of fault-tolerant linear digital state variable systems: Theory and techniques. *IEEE Transactions on Computers*, 42:794–808, July 1993.
- [23] C.-Y. Chen and J. A. Abraham. Fault tolerance systems for the computation of eigenvalues and singular values. In *Proceedings of SPIE*, pages 228–237, 1986.
- [24] K.-T. Cheng and J.-Y. Jou. A functional fault model for sequential machines. *IEEE Transactions on Computer-Aided Design*, 11:1065–1073, September 1992.
- [25] Y.-H. Choi and M. Malek. A fault-tolerant systolic sorter. *IEEE Transactions on Computers*, 37:621–624, May 1988.
- [26] G. Cohen, P. Moller, J.-P. Quadrat, and M. Viot. Algebraic tools for the performance evaluation of discrete event systems. *Proceedings of the IEEE*, 77:39–85, January 1989.
- [27] R. Cuninghame-Green. *Minimax Algebra*. Springer-Verlag, Berlin/Heidelberg/New York, 1979.

- [28] W. Daehn, T. W. Williams, and K. D. Wagner. Aliasing errors in linear automata used as multiple-input signature analyzers. *IBM Journal of Research and Development*, 34:363–380, March-May 1990.
- [29] M. Damiani, P. Olivo, and B. Ricco. Analysis and design of linear finite state machines for signature analysis testing. *IEEE Transactions on Computers*, 40:1034–1045, September 1991.
- [30] A. A. Desrochers and R. Y. Al-Jaar. *Applications of Petri Nets in Manufacturing Systems*. IEEE Press, 1994.
- [31] S. Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis*. McGraw Hill, New York, 1994.
- [32] R. L. Dobrushin and S. S. Ortyukov. Upper bound on the redundancy of self-correcting arrangements of unreliable functional elements. *Problems of Information Transmission*, 13:203–218, July-September 1977.
- [33] P. Elias. Computation in the presence of noise. *IBM Journal of Research and Development*, 2:346–353, October 1958.
- [34] F. Fagnani and S. Zampieri. Dynamical systems and convolutional codes over finite abelian groups. *IEEE Transactions on Information Theory*, 42:1892–1912, November 1996.
- [35] V. Y. Fedorov and V. O. Chukanov. Analysis of the fault tolerance of complex systems by extensions of Petri nets. *Automation and Remote Control*, 53:271–280, February 1992.
- [36] P. Gács. Reliable computation with Cellular Automata. *Journal of Computer and System Sciences*, 32:15–78, February 1986.
- [37] R. G. Gallager. *Low-Density Parity Check Codes*. MIT Press, Cambridge, Massachusetts, 1963.
- [38] R. G. Gallager. *Information Theory and Reliable Communication*. John Wiley & Sons, New York/London/Sydney/Toronto, 1968.
- [39] M. Geiger and T. Müller-Wipperfurth. FSM decomposition revisited: Algebraic structure theory applied to MCNC benchmark FSM's. In *Proceedings of the 28th Design Automation Conference*, volume 6, pages 182–185, 1991.
- [40] A. Ginzburg. *Algebraic Theory of Automata*. Academic Press, New York, 1968.
- [41] S. W. Golomb. *Shift Register Sequences*. Holden-Day, San Francisco, 1967.
- [42] P. A. Grillet. *Semigroups*. Marcel Dekker Inc., New York, 1995.
- [43] C. N. Hadjicostis. Fault-Tolerant Computation in Semigroups and Semirings. M. Eng. thesis, EECS Department, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1995.

- [44] C. N. Hadjicostis and G. C. Verghese. Fault-tolerant computation in semigroups and semirings. In *Proceedings of Int. Conf. on Digital Signal Processing*, volume 2, pages 779–784, 1995.
- [45] R. E. Harper, J. H. Lala, and J. I. Deyst. Fault-tolerant parallel processor architecture review. In *Eighteenth Int. Symposium on Fault-Tolerant Computing, Digest of Papers*, pages 252–257, 1988.
- [46] M. A. Harrison. *Lectures on Linear Sequential Machines*. Academic Press, New York/London, 1969.
- [47] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, Englewood Cliffs, New Jersey, 1966.
- [48] J. T. Håstad. *Computational Limitations for Small-Depth Circuits*. MIT Press, Cambridge, Massachusetts, 1987.
- [49] I. N. Herstein. *Topics in Algebra*. Xerox College Publishing, Lexington, Massachusetts, 1975.
- [50] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33:518–528, June 1984.
- [51] M. Ikeda and D. D. Šiljak. Generalized decompositions of dynamic systems and vector Lyapunov functions. *IEEE Transactions on Automatic Control*, AC-26:1118–1125, October 1981.
- [52] M. Ikeda and D. D. Šiljak. An inclusion principle for dynamic systems. *IEEE Transactions on Automatic Control*, AC-29:244–249, March 1984.
- [53] M. Ikeda, D. D. Šiljak, and D. E. White. Expansion and contraction of linear time-varying systems. In *Proceedings of 21st IEEE Conf. on Decision and Control*, pages 1202–1209, 1982.
- [54] V. S. Iyengar and L. L. Kinney. Concurrent fault detection in microprogrammed control units. *IEEE Transactions on Computers*, 34:810–821, September 1985.
- [55] N. Jacobson. *Basic Algebra I*. W. H. Freeman and Company, San Francisco, 1974.
- [56] J.-Y. Jou and J. A. Abraham. Fault-tolerant matrix arithmetic and signal processing on highly concurrent parallel structures. *Proceedings of the IEEE*, 74:732–741, May 1986.
- [57] J.-Y. Jou and J. A. Abraham. Fault-tolerant FFT networks. *IEEE Transactions on Computers*, 37:548–561, May 1988.
- [58] T. Kailath. *Linear Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1980.
- [59] I. Koren and A. D. Singh. Fault-tolerance in VLSI circuits. *IEEE Computer*, pages 73–83, July 1990.

- [60] A. V. Kuznetsov. Information storage in a memory assembled from unreliable components. *Problems of Information Transmission*, 9:254–264, July–September 1973.
- [61] R. W. Larsen and I. S. Reed. Redundancy by coding versus redundancy by replication for failure-tolerant sequential circuits. *IEEE Transactions on Computers*, 21:130–137, February 1972.
- [62] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines. *Proceedings of the IEEE*, 84:1090–1123, August 1996.
- [63] R. Leveugle, Z. Koren, I. Koren, G. Saucier, and N. Wehn. The Hyeti defect tolerant microprocessor: A practical experiment and its cost-effectiveness analysis. *IEEE Transactions on Computers*, 43:1398–1406, December 1994.
- [64] S. C. Liang and S. Y. Kuo. Concurrent error detection and correction in real-time systolic sorting arrays. In *Proceedings of 20th IEEE Int. Symposium on Fault-Tolerant Computing*, 1990.
- [65] R. Lidl and G. Pilz. *Applied Abstract Algebra*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1985.
- [66] E. S. Ljapin. *Semigroups*, volume Three of *Translations of Mathematical Monographs*. American Mathematical Society, Providence, Rhode Island, 1974.
- [67] D. G. Luenberger. *Introduction to Dynamic Systems: Theory, Models, & Applications*. John Wiley & Sons, New York, 1979.
- [68] D. J. C. MacKay and R. M. Neal. Good codes based on very sparse matrices. In Colin Boyd, editor, *Cryptography and Coding, 5th IMA Conference*, volume 1025 of *Lecture Notes in Computer Science*, pages 100–111, 1995.
- [69] G. A. Margulis. Explicit construction of concentrators. *Problems of Information Transmission*, 9:71–80, October–December 1974.
- [70] R. L. Martin. *Studies in Feedback-Shift-Register Synthesis of Sequential Machines*. MIT Press, Cambridge, Massachusetts, 1969.
- [71] J. O. Moody and P. J. Antsaklis. Supervisory control using computationally efficient linear techniques: A tutorial introduction. In *Proceedings of 5th IEEE Mediterranean Conf. on Control and Systems*, 1997.
- [72] J. O. Moody and P. J. Antsaklis. *Supervisory Control of Discrete Event Systems Using Petri Nets*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1998.
- [73] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541–580, April 1989.
- [74] V. S. S. Nair and J. A. Abraham. Real-number codes for fault-tolerant matrix operations on processor arrays. *IEEE Transactions on Computers*, 39:426–435, April 1990.

- [75] S. Niranjana and J. F. Frenzel. A comparison of fault-tolerant state machine architectures for space-born electronics. *IEEE Transactions on Reliability*, 45:109–113, March 1996.
- [76] J. P. Norton. Structural zeros in the modal matrix and its inverse. *IEEE Transactions on Automatic Control*, AC-25:980–981, October 1980.
- [77] R. A. Parekhji, G. Venkatesh, and S. D. Sherlekar. A methodology for designing optimal self-checking sequential circuits. In *Proceedings of Int. Conf. VLSI Design*, pages 283–291. IEEE CS Press, 1991.
- [78] R. A. Parekhji, G. Venkatesh, and S. D. Sherlekar. Concurrent error detection using monitoring machines. *IEEE Design and Test of Computers*, 12:24–32, March 1995.
- [79] B. Parhami. Voting algorithms. *IEEE Transactions on Reliability*, 43:617–629, December 1994.
- [80] M. Peercy and P. Banerjee. Fault-tolerant VLSI systems. *Proceedings of the IEEE*, 81:745–758, May 1993.
- [81] W. W. Peterson and E. J. Weldon Jr. *Error-Correcting Codes*. MIT Press, Cambridge, Massachusetts, 1972.
- [82] N. Pippenger. On networks of noisy gates. In *Proceedings of the 26th IEEE FOCS Symposium*, pages 30–38, 1985.
- [83] N. Pippenger. Developments in the synthesis of reliable organisms from unreliable components. In *Proceedings of Symposia in Pure Mathematics*, volume 50, pages 311–324, 1990.
- [84] D. K. Pradhan, editor. *Fault-Tolerant Computing, Theory and Techniques*, chapter 5. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [85] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1996.
- [86] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–97, January 1989.
- [87] T. R. N. Rao. *Error Coding for Arithmetic Processors*. Academic Press, New York, 1974.
- [88] T. R. N. Rao and E. Fujiwara. *Error-Control Coding for Computer Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
- [89] G. R. Redinbo. Finite field fault-tolerant digital filtering architecture. *IEEE Transaction on Computers*, 36:1236–1242, October 1987.
- [90] G. R. Redinbo. Signal processing architectures containing distributed fault-tolerance. In *Conference Record — Twentieth Asilomar Conf. on Signals, Systems & Computers*, pages 711–716, 1987.

- [91] C. Reutenauer. *The Mathematics of Petri Nets*. Prentice Hall, New York, 1990.
- [92] R. A. Roberts and C. T. Mullis. *Digital Signal Processing*. Addison-Wesley, Reading, Massachusetts, 1987.
- [93] A. Roy-Chowdhury and P. Banerjee. Algorithm-based fault location and recovery for matrix computations on multiprocessor systems. *IEEE Transactions on Computers*, 45:1239–1247, November 1996.
- [94] A. Sengupta, D. K. Chattopadhyay, A. Palit, A. K. Bandyopadhyay, and A. K. Choudhury. Realization of fault-tolerant machines — linear code application. *IEEE Transactions in Computers*, 30:237–240, March 1981.
- [95] C. E. Shannon. A mathematical theory of communication (Part I). *Bell System Technical Journal*, 27:379–423, July 1948.
- [96] C. E. Shannon. A mathematical theory of communication (Part II). *Bell System Technical Journal*, 27:623–656, October 1948.
- [97] J. Sifakis. Realization of fault-tolerant systems by coding Petri nets. *Journal of Design Automation and Fault-Tolerant Computing*, 3:93–107, April 1979.
- [98] M. Silva and S. Velilla. Error detection and correction on Petri net models of discrete events control systems. In *Proceedings of the ISCAS*, pages 921–924, 1985.
- [99] M. Sipser and D. A. Spielman. Expander codes. *IEEE Transactions on Information Theory*, 42:1710–1722, November 1996.
- [100] D. A. Spielman. Highly fault-tolerant parallel computation. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, volume 37, pages 154–160, 1996. Also available at www-math.mit.edu/~spielman/.
- [101] D. A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42:1723–1731, November 1996.
- [102] J. Sun, E. Cerny, and J. Gecsei. Fault tolerance in a class of sorting networks. *IEEE Transactions on Computers*, 43:827–837, July 1994.
- [103] D. L. Tao and K. Kantawala. Evaluating the reliability improvements of fault-tolerant array processors using algorithm-based fault tolerance. *IEEE Transactions on Computers*, 46:725–730, June 1997.
- [104] M. G. Taylor. Randomly Perturbed Computational Systems. PhD thesis, EECS Department, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1966.
- [105] M. G. Taylor. Reliable computation in computing systems designed from unreliable components. *The Bell System Journal*, 47:2239–2366, December 1968.
- [106] M. G. Taylor. Reliable information storage in memories designed from unreliable components. *The Bell System Journal*, 47:2299–2337, December 1968.

- [107] J. von Neumann. *Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components*. Princeton University Press, Princeton, New Jersey, 1956.
- [108] J. Wakerly. *Error Detecting Codes, Self-Checking Circuits and Applications*. Elsevier Science, Amsterdam/New York, 1978.
- [109] G. X. Wang and G. R. Redinbo. Probability of state transition errors in a finite state machine containing soft failures. *IEEE Transactions on Computers*, 33:269–277, March 1984.
- [110] S. A. Ward and R. H. Halstead. *Computation Structures*. MIT Press, McGraw-Hill Company, Cambridge, Massachusetts, 1990.
- [111] S. B. Wicker. *Error Control Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.
- [112] S. Winograd and J. D. Cowan. *Reliable Computation in the Presence of Noise*. MIT Press, Cambridge, Massachusetts, 1963.
- [113] W. M. Wonham. *Linear Multivariable Control: A Geometric Approach*. Springer-Verlag, New York, 1985.
- [114] K. Yamalidou, J. Moody, M. Lemmon, and P. Antsaklis. Feedback control of Petri nets based on place invariants. *Automatica*, 32:15–28, January 1996.
- [115] B. P. Zeigler. Every discrete input machine is linearly simulatable. *Journal of Computer and System Sciences*, 7:161–167, April 1973.