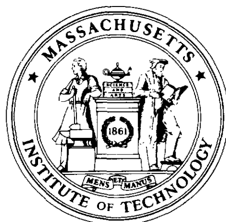
The RESEARCH LABORATORY
of
ELECTRONICS
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS 02139

Fault-Tolerant Computation in Semigroups and Semirings

Christoforos N. Hadjicostis

RLE Technical Report No. 594

May 1995



Fault-Tolerant Computation in Semigroups and Semirings

Christoforos N. Hadjicostis

RLE Technical Report No. 594

May 1995

The Research Laboratory of Electronics
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS 02139-4307

This work was supported in part by the Department of the Navy, Office of the Chief of Naval Research under Grant N00014-93-1-0686 as part of the Advanced Research Projects Agency's RASSP program.

Fault-Tolerant Computation in Semigroups and Semirings

by

Christoforos N. Hadjicostis

Submitted to the
Department of Electrical Engineering and Computer Science

January 27, 1995

In partial fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The traditional approach to fault-tolerant computation has been via modular redundancy. Although universal and simple, modular redundancy is inherently expensive and inefficient. By exploiting particular structural features of a computation or algorithm, recently developed Algorithm-Based Fault Tolerance (ABFT) techniques manage to offer more efficient fault coverage at the cost of narrower applicability and harder design. In the special case of arithmetic codes, previous work has shown that a variety of useful results and constructive procedures can be obtained when the computations take place in an abelian group. In this thesis, we develop a systematic algebraic approach for computations occurring in an abelian *semigroup*, thereby extending to a much more general setting many of the results obtained earlier for the group case. Examples of the application of these results to representative semigroups and higher semigroup-based algebraic structures, such as a semiring, are also included.

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

Acknowledgments

I would like thank my thesis supervisor, Professor George Verghese, for the excellent advice and the unlimited help and support that he has provided to me during my time as a graduate student. The successful completion of this thesis would not have been possible without his enthusiasm for my work and his encouragement and patience whenever I reached a point of difficulty.

I am also equally grateful to Professor Alan Oppenheim for his direction and help over these last two years. Not only did his support and guidance make this thesis possible, but also, by including me in the Digital Signal Processing Group (DSPG), he gave me the opportunity to work in an excellent academic environment that helped me mature both as a graduate student and as a person.

I would like to thank all members of the DSPG for the help and advice they have given me. Their friendship was also invaluable. My special thanks to Haralambos Papadopoulos for his patience and guidance at the numerous instances I turned to him for help.

Contents

1 Introduction **1**
1.1 Definitions and Motivation . . . 1
1.2 Main Approaches to Fault Tolerance . . . 2
1.2.1 Modular Redundancy . . . 2
1.2.2 Arithmetic Codes . . . 3
1.2.3 Algorithm-Based Fault Tolerance . . . 5
1.3 Scope and Major Contributions of the Thesis . . . 8
1.4 Outline of the Thesis . . . 9

2 Group-Theoretic Framework **11**
2.1 Introduction . . . 11
2.2 Computational Model . . . 11
2.2.1 General Model of a Fault-Tolerant System . . . 11
2.2.2 Computation in a Group . . . 13
2.2.3 Computational Model for Group Operations . . . 14
2.3 Group Framework . . . 15
2.3.1 Use of Group Homomorphisms . . . 15
2.3.2 Error Detection and Correction . . . 18
2.3.3 Separate Codes . . . 19
2.4 Applications to Other Algebraic Systems . . . 21
2.5 Summary . . . 22

3 Semigroup-Theoretic Framework **23**
3.1 Introduction . . . 23
3.2 Computation in a Semigroup . . . 23
3.2.1 Introduction to Semigroups . . . 24
3.2.2 Computational Model for Semigroup Operations . . . 25
3.3 Use of Semigroup Homomorphisms . . . 27
3.4 Redundancy Requirements . . . 30
3.5 Separate Codes for Semigroups . . . 32
3.5.1 Description of the Model for the Codes . . . 33
3.5.2 Analysis of the Parity Encoding . . . 33
3.5.3 Determination of Possible Homomorphisms . . . 34
3.5.4 Comparison with the Group Case . . . 37
3.6 Summary . . . 39

4 **Protecting Semigroup Computations: Some Examples** **40**
4.1 Introduction 40
4.2 Examples of Separate Codes 40
4.2.1 Separate Codes for $(\mathbf{N}_0, +)$ 41
4.2.2 Separate Codes for $(\mathbf{N}, \times)$ 45
4.2.3 Separate Codes for $(\mathbb{Z} \cup \{-\infty\}, \text{MAX})$ 48
4.3 Examples of Non-Separate Codes 49
4.4 Summary 49

5 **Frameworks for Higher Algebraic Structures** **51**
5.1 Introduction 51
5.2 Ring-Theoretic Framework 52
5.2.1 Computation in a Ring 52
5.2.2 Use of Ring Homomorphisms 54
5.2.3 Separate Codes for Rings 55
5.3 Examples in the Ring-Theoretic Framework 56
5.3.1 Examples of Non-Separate Codes 57
5.3.2 Examples of Separate Codes 58
5.4 Semiring-Theoretic Framework 58
5.4.1 Computation in a Semiring 58
5.4.2 Use of Semiring Homomorphisms 60
5.4.3 Separate Codes for Semirings 61
5.5 Examples in the Semiring-Theoretic Framework 65
5.5.1 Separate Codes for $(\mathbf{N}_0, +, \times)$ 65
5.5.2 Separate Codes for $(\mathbb{Z} \cup \{\pm\infty\}, \text{MIN}, \text{MAX})$ 68
5.5.3 Separate Codes for $(\mathbb{Z} \cup \{-\infty\}, \text{MAX}, +)$ 68
5.6 Summary 69

6 **Summary of Contributions, and Suggestions for Future Research** **70**
6.1 Contributions and Conclusions 70
6.2 Future Research Directions 71
6.2.1 Hardware Implementation and the Error Model 71
6.2.2 Non-Abelian Group or Semigroup Computations 72
6.2.3 Realizability of Arithmetic Codes 72
6.2.4 Development of a Probabilistic Framework 72
6.2.5 Subsystem Decomposition and Machines 73
6.2.6 Links to the Theory of Error-Correcting Codes 74

A **Proofs of Theorems** **76**
A.1 Enumerating all Separate Codes for $(\mathbf{N}_0, +)$ 76
A.2 Equivalence of Semiring Congruence Classes and Semiring Complexes 79

List of Figures

1-1 Fault-tolerant system design using Triple Modular Redundancy (TMR). . . 2
1-2 Protection of operation $\circ$ through the use of arithmetic codes. 3
1-3 A simple example of an αN arithmetic code for protecting integer addition. 5
1-4 ABFT technique for matrix multiplication. 7

2-1 Model of a fault-tolerant system as a cascade of three subsystems. . . 12
2-2 Model of a fault-tolerant computation for a group product. 14
2-3 Model of a fault-tolerant computation for a group product under an additive error model. 16
2-4 Fault tolerance in a computation using an abelian group homomorphism. 17
2-5 Structure of the redundant group H for error detection and correction. 19
2-6 A simple example of a separate arithmetic code. 20
2-7 Fault-tolerant model for a group operation using a separate code. . . 21

3-1 Fault-tolerant model for a semigroup computation. 25
3-2 Structure of the redundant semigroup for error detection and correction. 31
3-3 Model of a fault-tolerant system that uses separate codes. 32
3-4 Structure of the parity group in separate codes. 37
3-5 Structure of the parity semigroup in separate codes. 38

4-1 Example of a parity check code for the group $(\mathbb{Z}, +)$. 41
4-2 Example of a parity check code for the semigroup $(\mathbf{N}_0, +)$. 42
4-3 Example of a parity check code for the semigroup $(\mathbf{N}_0, +)$. 44
4-4 Example of a parity check code for the semigroup $(\mathbf{N}, \times)$. 46
4-5 Example of a parity check code for the semigroup $(\mathbf{N}, \times)$. 47

5-1 Fault-tolerant model for a ring computation. 53
5-2 Fault-tolerant model for a semiring computation. 60
5-3 Example of a residue check mod 4 for the semiring $(\mathbf{N}_0, +, \times)$. 65
5-4 Example of a parity check code for the semiring $(\mathbf{N}_0, +, \times)$. 66

List of Tables

5.1 Defining tables of the operations $\oplus$ and $\otimes$ for the parity semiring T. . 67

Chapter 1

Introduction

1.1 Definitions and Motivation

A system that performs a complex computational task is subject to many different kinds of failures, depending on the reliability of its components and the complexity of their subcomputations. These failures might corrupt the overall computation and lead to undesirable, erroneous results. A system designed with the ability to detect and, if possible, correct internal failures is called *fault-tolerant.*

A fault-tolerant system tolerates internal *errors* (caused by permanent or transient physical *faults*) by preventing these errors from corrupting the final result. This process is known as *error masking.* Examples of permanent physical faults would be manufacturing defects, or irreversible physical damage, whereas examples of transient physical faults include noise, signal glitches, and environmental factors, such as overheating. *Concurrent error masking*, that is detection and correction of errors concurrently with computation, is the most desirable form of error masking because no degradation in the overall performance of the system takes place.

A necessary condition for a system to be fault-tolerant is that it exhibits *redundancy*, to allow it to distinguish between the valid and invalid states or, equivalently, between the correct and incorrect results. However, redundancy is expensive and counter-intuitive to the traditional notion of system design. The success of a fault-tolerant design relies on making efficient use of hardware by adding redundancy in those parts of the system that are more liable to failures than others.

The design of fault-tolerant systems is motivated by applications that require high reliability. Examples of such applications are:

- Life-critical applications (such as medical equipment, or aircraft controllers) where errors can cost human lives.
- Remote applications where repair and monitoring is prohibitively expensive.
- Applications in a hazardous environment where repair is hard to accomplish.

The more intensive a computational task is, the higher is the risk for errors. For example, computationally intensive signal processing applications and algorithms are

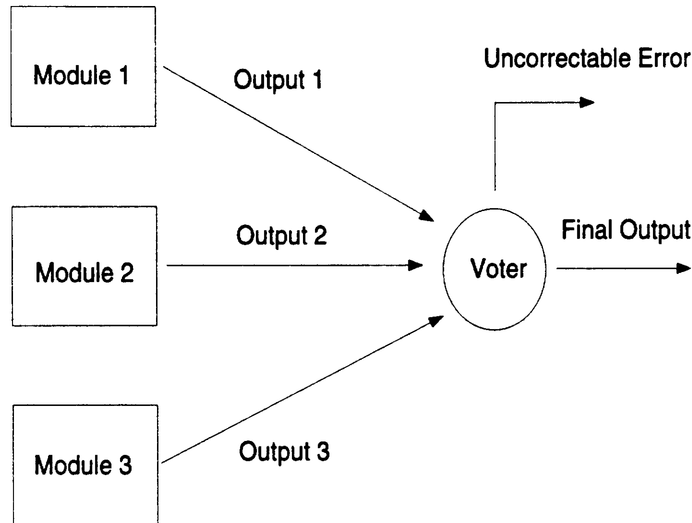


Figure 1-1: Fault-tolerant system design using Triple Modular Redundancy (TMR).

at high risk for erroneous results. As the complexity of Discrete Signal Processing (DSP) and other special-purpose integrated circuits increases, their vulnerability to faults (either permanent or transient) increases as well. By designing fault-tolerant integrated circuits, we can hope to achieve not only better reliability, but also higher yield during the manufacturing process since manufacturing defaults (a form of a permanent fault) can be accepted up to some degree.

All of the above examples show the importance of fault tolerance and underline the increasing need for fault-tolerant design techniques.

1.2 Main Approaches to Fault Tolerance

1.2.1 Modular Redundancy

The traditional approach for achieving fault tolerance has been modular redundancy. An example of *Triple Modular Redundancy* (TMR) is shown in Figure 1-1. Three identical modules perform the exact same computation separately and in parallel. Their results are compared by a voter, which chooses the final result based on what the majority of the modules decide. For example, if all the modules agree on a result, then the voter outputs that result. If only two of the them agree, then the voter outputs the result obtained by these two processors and declares the other one faulty. When all processors disagree, the voter signals an error in the system. It is worth mentioning that a variety of other approaches towards voting exist.

This methodology can easily be extended to *N-Modular Redundancy* by using N different identical modules that operate in parallel, and majority voting to distinguish and decide about the correct result. By using majority voting we can detect (but not correct) D errors and correct C errors ($D \geq C$), if $N \geq D + C + 1$. In fact, if the modules are *self-checking* (that is, they have the ability to detect internal errors), then we can detect up to N and correct up to $N - 1$ errors [1].

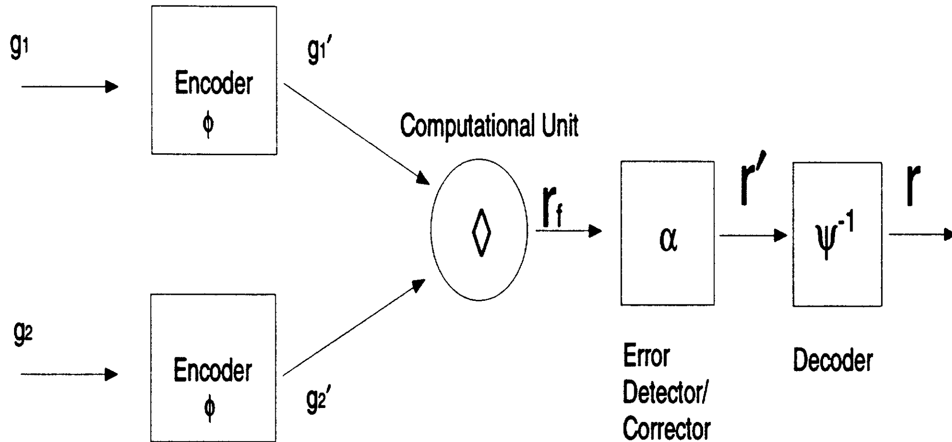


Figure 1-2: Protection of operation $\circ$ through the use of arithmetic codes.

N-Modular Redundancy has traditionally been the primary methodology for fault-tolerant system design, mainly because it can be applied in a very simple and straightforward way to any kind of computational (or other) system. A very desirable feature of modular redundancy is that it effectively decouples the system design from the fault tolerance design. However, it is usually prohibitively expensive because it involves replicating the system N times. For this reason, a variety of hybrid methods has evolved, involving hierarchical levels of modular redundancy: only the parts of the system that are more vulnerable to faults are replicated. When time delay is not an issue, we can afford to repeat a computation. Therefore, another approach is possible: rather than having N different modules perform the same computation at the same time, we can afford to have one system that repeats the same computation N times. The effect is exactly the same as N-Modular redundancy as long as no permanent faults have taken place.

Examples of commercial and other systems that use modular redundancy techniques are referenced in [1].

1.2.2 Arithmetic Codes

While universally applicable and simple to implement, modular redundancy is inherently expensive and inefficient. For example, in a TMR implementation we triplicate the whole system in order to detect and correct a single error. This is prohibitively expensive. A more efficient approach towards fault-tolerant computation is the use of *arithmetic codes*, although this is more limited in applicability and possibly harder to implement.

Arithmetic codes are used to protect simple operations on integer data, such as addition and multiplication. They can be thought of as a class of error-correcting codes whose properties remain invariant under the operation that needs to be made robust. Figure 1-2 provides a graphical illustration of the general operation of an arithmetic code. In this case, the desired, error-free result is: $r = g_1 \circ g_2$. In order to achieve this result, while protecting the operation $\circ$, the following major steps are

taken:

- *Encoding:* First, we add redundancy to the representation of the data by using a suitable and efficient encoding:

$$g_1' = \phi(g_1)$$
$$g_2' = \phi(g_2)$$

- *Operation:* The operation on the encoded data does not necessarily have to be the same as the desired operation on the original data. In terms of Figure 1-2, this modified operation is denoted by $\diamond$:

$$r' = g_1' \diamond g_2'$$

  where r' is the actual result that the modified operation give under fault-free conditions. In reality, one or more errors $\{e_i\}$ can take place, and the result of the computation of the encoded data is a possibly faulty result r_f, which is a function of the encoded data and the errors that took place:

$$r_f = f(g_1', g_2', e)$$

- *Error Detection and Correction:* If enough redundancy exists in the encoding of the data, we hope to be able to correct the error(s) by analyzing the way in which the properties of the encoding have been modified. In such a case, r_f can be masked back to r'. In Figure 1-2, this is done by the error correcting mapping α:

$$r' = \alpha(r_f)$$

- *Decoding:* The final, error-free result r can be obtained using ψ^{-1} as an inverse mapping. Note that the use of ϕ^{-1} is a possibility if ϕ is a one-to-one mapping; however, in general, there are a lot of other alternatives since the result r does not necessarily lie in the same space as the operands g_1 and g_2. Therefore, in order to have a more general model of arithmetic codes we denote the inverse mapping as ψ^{-1}:

$$r = \psi^{-1}(r')$$

An arithmetic code that can be formulated in the above four steps is not necessarily a useful one. Two further requirements need to be satisfied: first, it must provide sufficient protection for the faults that are likely to occur in the specific application, and second, it must be easy to encode and decode. If the above requirements are not met, then the code is not practical. It is either insufficient, or it is computationally intensive[1].

[1] An extreme example would be an encoding that is three times more complicated than the actual operation we would like to protect. In such a case, it would be more convenient to use TMR rather than this complicated arithmetic code.

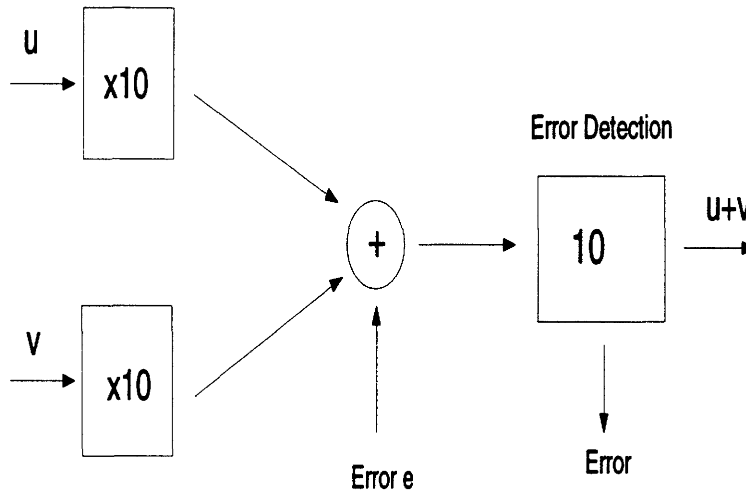


Figure 1-3: A simple example of an αN arithmetic code for protecting integer addition.

A very simple example of an arithmetic code is presented in Figure 1-3. In this case, we are trying to protect integer addition. The encoding simply involves multiplication of the operands $\{u, v\}$ by a factor of 10. The operation on the encoded data is again addition. Error detection is simply division by 10: if the result is corrupted, we hope that it will not be divisible by 10, in which case we will be able to detect that an error took place[2]. However, error correction is impossible under this kind of arithmetic coding, unless a more detailed error model is available. Decoding is performed at the same time as we perform error detection. This specific example is an instance of an αN code [2] where $\alpha = 10$. Under certain conditions and certain choices of α, αN codes can be used to correct a single error. Note that in the case of αN codes, redundancy is added into the computational system by increasing the *dynamic* range of the system (by a factor of α).

Arithmetic codes do not always have the simple structure of the example above. More advanced and more complicated schemes do exist. In fact, there exist arithmetic codes that are able to protect real or complex data (which is not true in the above example) and more elaborate computations than simply addition. Methods to protect entire arrays (sequences) of data have been developed as well. This more advanced form of arithmetic coding is usually referred to as *Algorithm-Based Fault Tolerance*, and is discussed briefly in the following section.

1.2.3 Algorithm-Based Fault Tolerance

Algorithm-Based Fault Tolerance (ABFT) schemes are highly involved arithmetic coding techniques that usually deal with real/complex arrays of data in multiprocessor concurrent systems. The term was introduced by J. Abraham and coworkers [3]-[9] in 1984. Since then, a variety of signal processing and other computationally intensive

[2]Note that an error under which the result remains a multiple of 10 is undetectable.

algorithms have been adapted to the requirements of ABFT.

As described in [5], there are three key steps involved in ABFT:

1. Encode the input data for the algorithm (just as in the general case of arithmetic coding).

2. Reformulate the algorithm so that it can operate on the encoded data and produce decodable results.

3. Distribute the computational tasks among different parts of the system so that any errors occurring within those subsystems can be detected and, hopefully, corrected.

A classic example of ABFT is the protection of $N \times N$ matrix multiplication on an $N \times N$ multiprocessor array [3]. The ABFT scheme detects and corrects any single (local) error using an extra checksum row and an extra checksum column. The resulting multiprocessor system is an $(N+1) \times (N+1)$ multiprocessor array. Therefore, the hardware overhead is minimal (it is of $O\left(\frac{1}{N}\right)$) compared to the naive use of TMR, which offers similar fault protection but triplicates the system ($O(1)$ hardware overhead). The execution time for the algorithm is slowed down negligibly: it now takes $3N$ steps, instead of $3N-1$. The time overhead is only $O(\frac{1}{N})$.

Figure 1-4 is an illustration of the above ABFT method for the case when $N = 3$. At the top of the figure, we see how unprotected computation of the product of two 3×3 square matrices A and B takes place in a 3×3 multiprocessor array. The data enters the multiprocessor system in the fashion illustrated by the arrows in the figure. Element a_{ij} corresponds to the element in the i-th row and j-th column of the matrix A, whereas b_{ij} is the corresponding element of the B matrix. At each time step n, each processor p_{ij} (the processor on the i-th row and j-th column of the $2D$ array) does the following:

1. It receives two pieces of data, one from the processor on the left and one from the processor on top. From the processor on the left ($p_{i(j-1)}$), it gets $b_{(n-(j+i-1))i}$ whereas from the processor on top it gets $a_{j(n-(j+i-1))}$. Note that if $(n-(j+i-1))$ is negative, no data has been received yet.

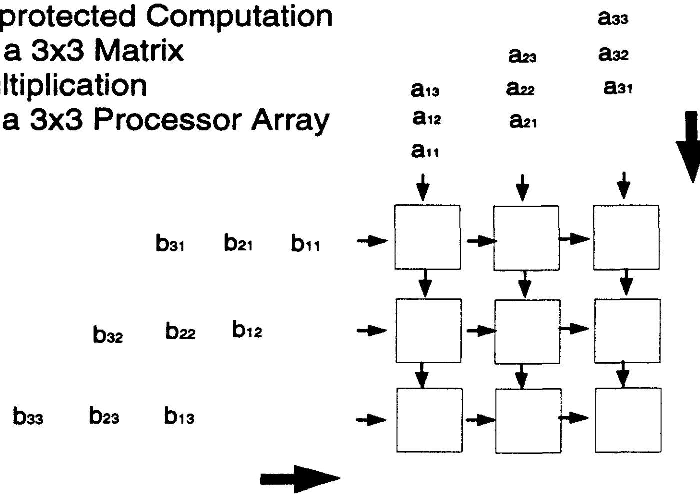
2. It multiplies the data received and adds the result to an accumulative sum s stored in its memory. Note that s is initialized to 0. If no data was received in the previous step, nothing is done at this step.

3. It passes the data received from the left to the processor on the right, and the data received from top to the processor below.

It is not hard to see that after $3N-1$ steps, the value of s_{ji} is:

$$s_{ji} = \sum_{n=0}^{3N-1} a_{i(n-(j+i-1))} \times b_{(n-(j+i-1))j} = C_{ij},$$

where a_{kl}, b_{kl} are 0 for $k,l < 0$ or $k,l > N$. Note that C_{ij} is the element in the i-th row and j-th column of the matrix $C = A \times B$. Therefore, after $3N-1$ steps, processor p_{ji} contains the value C_{ij}.



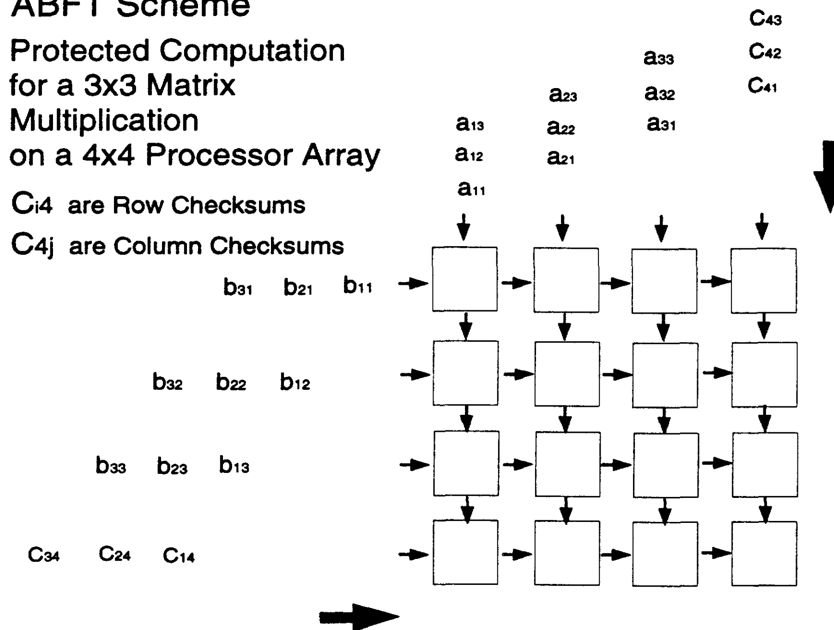


Figure 1-4: ABFT technique for matrix multiplication.

Protected computation is illustrated at the bottom of Figure 1-4. It uses a $(3+1) \times (3+1)$ multiprocessor array. Both matrices A and B are encoded into two new matrices, A' and B' respectively, in the following fashion:

- An extra row is added to matrix A, consisting of *column sums*, that is:

$$c_{4j} = \sum_{i=1}^{3} a_{ij}$$

  A' is now an $(N+1) \times N$ matrix.

- An extra column is added to matrix B, consisting of *row sums*, that is:

$$c_{i4} = \sum_{j=1}^{3} b_{ij}$$

  B' is now an $N \times (N+1)$ matrix.

The computation is executed in the usual way on a 4×4 multiprocessor array. The resulting matrix $C' = A' \times B'$ is a 4×4 matrix. If we exclude the last row and the last column from C', we get the original result $C = A \times B$. Moreover, the last row and column of C' consists of column and row checksums respectively. If one of the processors malfunctions, we can detect and correct the error by using the row and column checksums to first pinpoint the location of the error and then correct it. The basic assumption here is that no error propagates or, equivalently, the propagation of the data in the system is flawless. This is where TMR offers more coverage than this scheme: a single data propagation error will be caught by a TMR system, but not by this ABFT scheme.

The above example shows the superiority of ABFT over naive modular redundancy methods. By exploiting particular structural features of an algorithm or a computational system, ABFT achieves efficient fault protection at a much lower cost. Other examples of ABFT techniques involve other matrix arithmetic and signal processing applications [3] [4], fault-tolerant FFT computational systems [6], A/D conversion [10], and digital convolution [11].

1.3 Scope and Major Contributions of the Thesis

Arithmetic codes and ABFT techniques have been studied extensively by a lot of researchers for a variety of applications, such as those presented in [3]-[14]. All these cases involve efficient fault-tolerant schemes for protecting the specific application that was under consideration. However, until recently, no systematic and general way for developing arithmetic codes and ABFT techniques had been developed. The detection of exploitable structure in an algorithm, in a way that can provide efficient fault coverage, was more of a "black magic" technique than an engineering discipline.

In [1], an attempt to unify all of the above mentioned methods was made by developing a general framework that is extremely useful in analyzing, constructing and using arithmetic codes as a tool for protection against computational faults. Most

known arithmetic codes and ABFT techniques can be encompassed in this framework. What is required by [1] is that the operation (or, more generally, the computational task) can be modeled as an *abelian group* operation[3].

The framework extends naturally to other algebraic structures that have the underlying characteristics of an abelian group, such as rings, fields, modules and vector spaces. Therefore, even though the analysis started with a seemingly limited set of computational tasks that could be modeled using this framework, it has now been extended enough to include many of the examples of arithmetic codes and ABFT techniques that have been mentioned earlier and have been developed on an individual basis, for instance αN codes, matrix multiplication in the ring of matrices, and so on. A more detailed discussion of the results obtained in [1] is presented in Chapter 2 as an introduction to the topic of this thesis.

In this thesis, we extend the results obtained in [1]. We relax the requirement that the computation occurs in an abelian group, to the less strict requirement of an underlying *abelian semigroup* structure. The result is a much more general framework for investigating fault-tolerant systems in a mathematically rigorous and complete way. Important results from group and semigroup theory can directly be applied in systems of interest that comply to our requirements. Moreover, we manage to rigorously extend this framework to higher algebraic systems with an underlying group or semigroup structure, namely the ring and semiring structures.

Detailed connection to actual hardware realizations and their failure modes is *not* addressed in this thesis. There are many research issues that arise in making this hardware connection in a systematic and general way, and we intend to pursue these questions in follow-on work.

1.4 Outline of the Thesis

This thesis is organized as follows:

Chapter 2 provides an overview of the basic assumptions, techniques and results that were used in [1]. This chapter not only serves as a valuable introduction, but also provides the basic definitions that we need, the descriptions of the models that we use, and the assumptions that we make.

Chapter 3 rigorously extends the results of Chapter 2 to the semigroup setting. The analysis starts by defining the error and computational models that we use, and then proceeds to analyze arithmetic coding as a semigroup mapping. This analysis arrives at the important conclusion that the arithmetic code corresponds to a semigroup homomorphism. Thus, it provides us with a variety of algebraic tools that we can use in our subsequent analysis. The redundancy conditions for the mapping to provide sufficient fault tolerance under an *additive* error model[4] are derived next. A brief comparison with the corresponding conditions for the group case follows. Finally, a constructive procedure for the special case of separate codes[5] is presented.

[3]The definition of a group is given later in Chapter 2.

[4]The definition of the additive error model is given in Chapters 2 and 3.

[5]Separate codes are also known as "systematic separate codes". However, since a separate code

Chapter 4 demonstrates the use of the semigroup-theoretic framework that is developed in Chapter 3 by presenting a variety of examples of arithmetic codes for simple semigroup-based computations. In some cases, we achieve a complete characterization of all possible separate codes for a semigroup.

The results are extended in Chapter 5 to higher algebraic structures that admit two operations, namely the ring and semiring structures. In order to demonstrate the use of the framework, we also present a variety of examples of arithmetic codes for these structures.

Chapter 6 concludes the thesis with a summary of its major contributions and results. Moreover, we make suggestions for possible future directions and potentially interesting areas where research could be made.

is necessarily systematic [2], we will simply refer to them as "separate codes".

Chapter 2

Group-Theoretic Framework

2.1 Introduction

The development of a suitable arithmetic code for a given computational task can be very difficult, or even impossible. Arithmetic codes and ABFT techniques have been constructed for particular algorithms, but a systematic and mathematically rigorous way of addressing the design of arithmetic codes for a general computational task still needs to be developed.

A considerable step in this direction has been taken in [1]. By concentrating on computational tasks that can be modeled as abelian group operations, one can impose sufficient structure upon the computations to allow accurate characterization of the possible arithmetic codes and the form of redundancy that is needed. For example, it turns out that the encoding has to be an algebraic homomorphism that maps the computation from the original group to a homomorphic computation in a *larger* group, thereby adding redundancy into the system.

Computational tasks with an underlying abelian group structure are not a limited set of computational tasks, since many arithmetic operations are, in fact, abelian group operations. Moreover, once the framework for group-like operations is available, we can extend the domain of applications to algebraic systems with an underlying group structure, such as rings, fields, modules, and vector spaces.

In what follows, we give a brief introduction to the above results. We also describe the computational model and the basic assumptions that were made.

2.2 Computational Model

2.2.1 General Model of a Fault-Tolerant System

In [1], a general fault-tolerant system is modeled as shown in Figure 2-1. It consists of three major subsystems:

- *Redundant Computation Unit:* The actual computation takes place here in a way that incorporates redundancy. By redundancy we mean that the computation

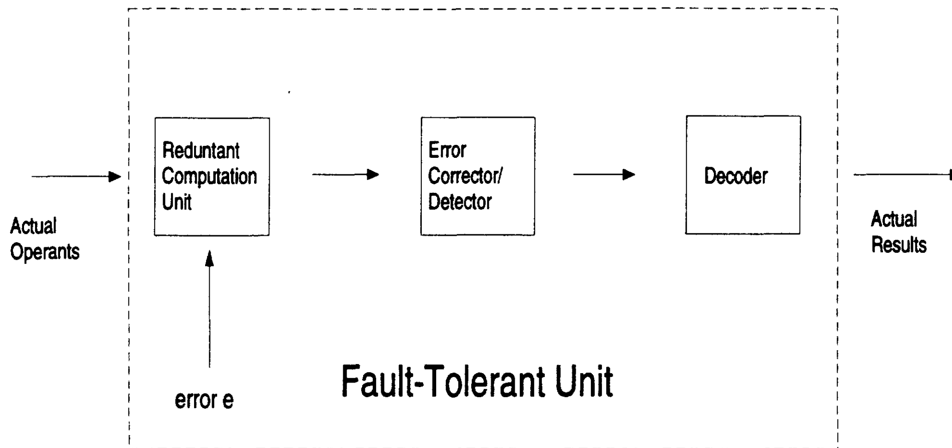


Figure 2-1: Model of a fault-tolerant system as a cascade of three subsystems.

unit involves extra states that only arise when an error occurs. Under fault-free operation these states are never involved; thus, we call them *redundant.*

- *Error Detector/Corrector:* By examining the state of the system after the output is produced, we can decide whether the output is valid or not. If the output is invalid, we might be able to use information about the particular state of the system to correct the error. However, this may not be always the case.

- *Decoder:* Once the result is corrected, all that remains is to map it to its non-redundant form. This is done at the decoder stage.

For example, in terms of this model, the αN arithmetic coding example of Figure 1-3 consists of the following subsystems:

- The units that perform the multiplication by 10 and the unit that performs the addition comprise the redundant computation unit.

- The unit that divides by 10 performs the error detection and/or correction; at the same time, it performs decoding.

Another example that can be viewed in terms of this model is the TMR system shown in Figure 1-1. The 3 copies of the system form the redundant computation unit, whereas the voter forms the error detector/corrector and the result decoder.

As can be seen in the above examples, some of the subsystems shown in Figure 2-1 might end up being indistinguishably combined. However, the model clearly presents the basic idea of a fault-tolerant system: at the point where the error takes place, the representation of the result involves redundancy. This redundancy gives us the ability to detect and/or correct the error in later stages.

An implicit assumption in the model is that no error takes place during error correcting and decoding. In a real system, we would need to protect the error corrector/detector and the result decoder using modular redundancy, or by making these subsystems extremely reliable. This is a reasonable assumption as long as the error

detector/corrector and the result decoder are simple enough compared to the computational unit, so that replicating them will not add a significant amount to the overall cost of the system. In fact, in the opposite case, when these units are expensive, it is probably preferable not to follow the approach outlined here and to simply use modular redundancy for the overall system[1].

2.2.2 Computation in a Group

For the rest of this chapter, we will focus on computational tasks that have an underlying group structure. The computation takes place in a set of elements that forms a group under the operation of interest. We start by the definition of a group (as given in [15]):

Definition: A non-empty subset of elements G is said to form a **group** $\mathcal{G} = (G, \circ)$ if on G there is a defined binary operation, called the *product* and denoted by $\circ$, such that

1. $a, b \in G$ implies $a \circ b \in G$ (closure).
2. $a, b, c \in G$ implies that $a \circ (b \circ c) = (a \circ b) \circ c$ (associativity).
3. There exists an element $e \in G$ such that $a \circ e = e \circ a = a$ for all $a \in G$ (e is called the *identity* element).
4. For every $a \in G$ there exists an element $a^{-1} \in G$ such that $a \circ a^{-1} = a^{-1} \circ a = e$ (the element a^{-1} is called the *inverse* of a).

Furthermore, if the group operation $\circ$ of G is commutative (for all $a, b \in G$, $a \circ b = b \circ a$), then G is called an *abelian group* [15]. In an abelian group, because of the associativity and commutativity, the order in which a series of group products is taken does not matter:

$$g_1 \circ g_2 \circ \ldots \circ g_p = g_{i_1} \circ g_{i_2} \circ \ldots \circ g_{i_p}$$

where $\{i_k\}$ for $k \in \{1, 2, ..., p\}$ is any permutation of $\{1, 2, ..., p\}$.

A simple example of a group is the set integers under addition, usually denoted by $(\mathbb{Z}, +)$. The four properties denoted above can be verified very easily. Specifically, in this case, the identity element is 0, and the inverse of an integer a is the integer $-a$. Another example, of a group is the set of non-zero rational numbers under multiplication, usually denoted by $(\mathbb{Q} - \{0\}, \times)$. The identity element in this case is 1 and the inverse of a rational number $q = \frac{n}{d}$ (where n, d are integers) is the rational number $q^{-1} = \frac{d}{n}$.

[1]The voter circuitry, which performs error detection and correction in the modular redundancy case, is usually simple enough to allow us to make it reliable easily.

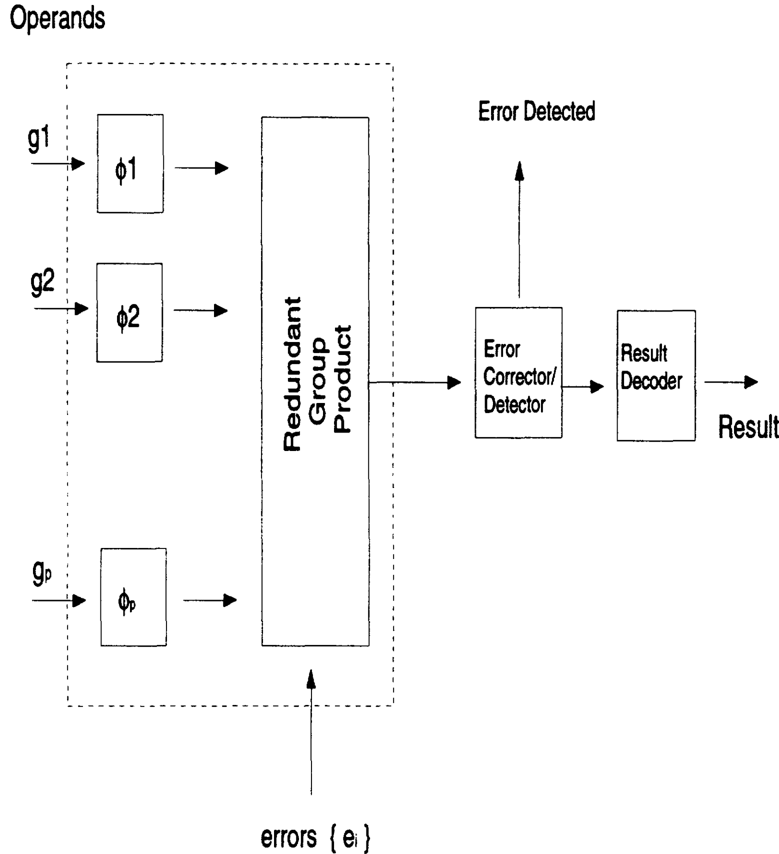


Figure 2-2: Model of a fault-tolerant computation for a group product.

2.2.3 Computational Model for Group Operations

Assume that the computation we want to protect can be modeled as an abelian group operation $\circ$ with operands $\{g_1, g_2, ..., g_p\}$. Then, the desired result r is:

$$r = g_1 \circ g_2 \circ ... \circ g_p$$

(any order of the $\{g_i\}$ will do). The fault-tolerant system for protecting this group product can be described as shown in Figure 2-2. The error corrector and the result decoder are exactly the same as in Figure 2-1. We assume that these units and the encoders $\{\phi_i\}$ are error-free. The redundant computation unit decomposes into a set of encoders (encoder ϕ_i corresponds to operand g_i), and a unit that performs a new *redundant* group product.

Essentially, as shown in [1] and discussed here later, this amounts to mapping the computation in the abelian group $(G, \circ)$ to another abelian group $(H, \diamond)$ of higher order, so that we are able to incorporate redundancy in our system. Note that the operation of the redundant group is not necessarily the same as the operation of the original one.

The desired result r is given by decoding (using a decoding mapping denoted by σ)

the result r_H of a redundant computation that took place in H:

$$r = g_1 \circ g_2 \circ ... \circ g_p = \sigma(r_H)$$

where $r_H = \phi_1(g_1) \diamond \phi_2(g_2) \diamond ... \diamond \phi_p(g_p)$

In [1] an *additive* error model is assumed. Errors $\{e_i\}$ are modeled as elements of H and are assumed to corrupt r_H in an additive[2] fashion. The possibly corrupted redundant result r'_H is given by:

$$\begin{aligned} r'_H &= \phi_1(g_1) \diamond \phi_2(g_2) \diamond ... \diamond \phi_p(g_p) \diamond e_1 \diamond e_2 \diamond ... \diamond e_\lambda \\ &= r_H \diamond e_1 \diamond e_2 \diamond ... \diamond e_\lambda \end{aligned}$$

The underlying assumptions of the additive error model are that errors are independent of the operands (which is a very realistic assumption for reasons explained in [1]), and that the effect of any error on the overall result is independent of which stage in the computation it occurs in. This last assumption is realistic because we have limited ourselves to *associative and abelian* operations. In such a case, the expression above is well-defined and its result is the same, irrespective of the order and the position in which the operands are evaluated.

We can simplify notation if we define the error $e = e_1 \diamond e_2 \diamond ... \diamond e_\lambda$:

$$r'_H = r_H \diamond e$$

where $e \in \mathcal{E}^{(\lambda)} = \mathcal{E} \diamond \mathcal{E} \diamond ... \diamond \mathcal{E}$ (λ times). If no error took place, e is the identity element.

Note that we can view the errors $\{e_1, e_2, ..., e_\lambda\}$ as regular operands that corrupt the product when a fault takes place during the computation. The computational model then becomes as shown in Figure 2-3. Under fault-free computation:

$$e_1 = e_2 = ... = e_\lambda = 0_\diamond$$

where $0_\diamond$ is the identity element of the redundant group.

2.3 Group Framework

2.3.1 Use of Group Homomorphisms

Once we have defined the computational and error models, we are ready to proceed with the analysis of the system. The subset of *valid results* H_V, which is obtained under error-free computation, can be defined by

$$H_V = \{\phi_1(g_1) \diamond \phi_2(g_2) \diamond ... \diamond \phi_p(g_p) \mid g_1, g_2, ..., g_p \in G\}$$

[2]The term "additive" makes more sense if the group operation $\circ$ is addition.

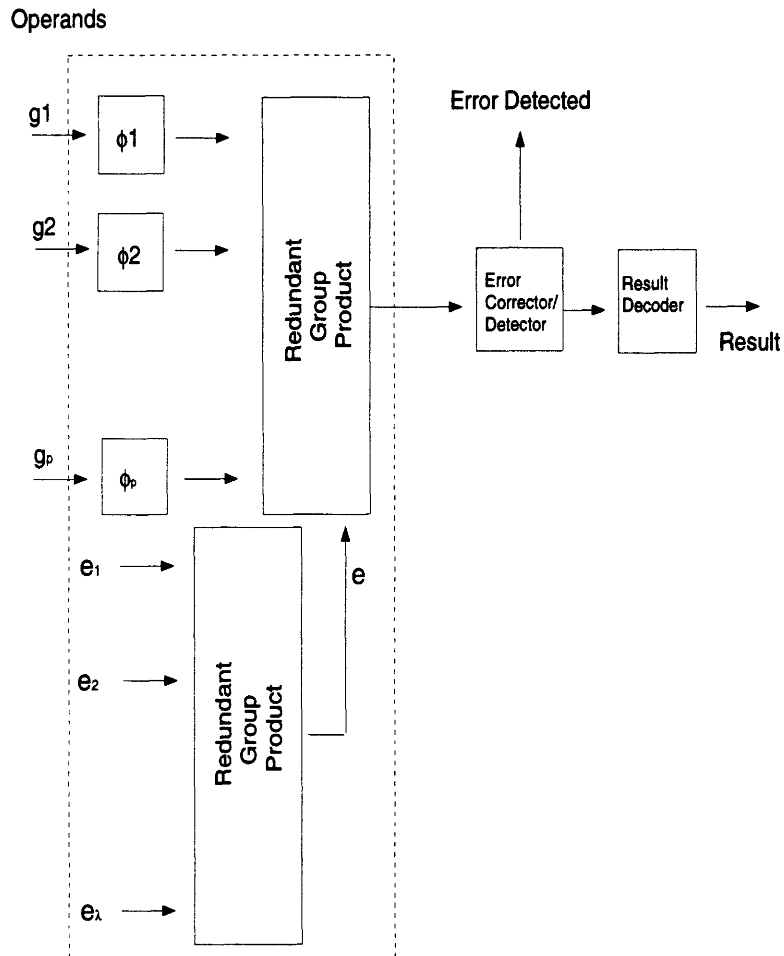


Figure 2-3: Model of a fault-tolerant computation for a group product under an additive error model.

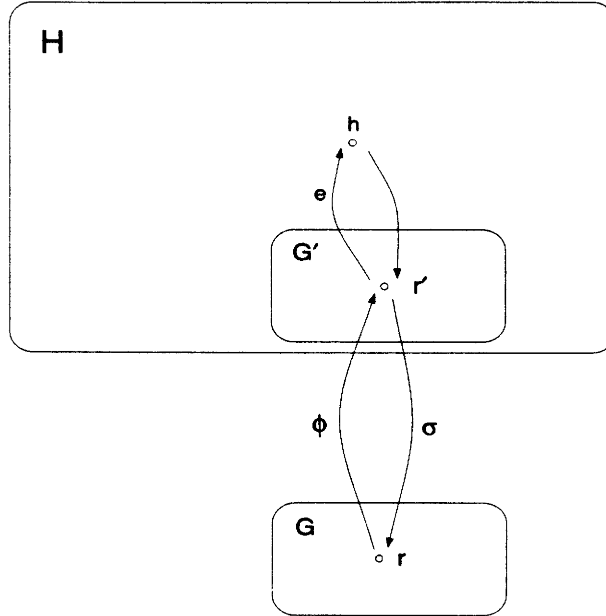


Figure 2-4: Fault tolerance in a computation using an abelian group homomorphism.

Recalling that $\sigma : H_V \longmapsto G$ denotes the mapping used by the decoder unit to map the set of valid results in H_V to elements of G, we have:

$$r = g_1 \circ g_2 \circ \ldots \circ g_p = \sigma(\phi_1(g_1) \diamond \phi_2(g_2) \diamond \ldots \diamond \phi_p(g_p)) \qquad (2.1)$$

If we require this mapping to be one-to-one[3], then σ^{-1} is a well-defined mapping and, as shown in [1], all encoders $\{\phi_i\}$ have to be the same and satisfy

$$\phi_i = \sigma^{-1} \equiv \phi$$

where the symbol ϕ will be used from now on to denote the encoding function. Equation (2.1) then reduces easily to:

$$\phi(g_1 \circ g_2) = \phi(g_1) \diamond \phi(g_2)$$

which is the defining property of a group homomorphism. This establishes a one-to-one correspondence between arithmetic codes for groups and group homomorphisms. Therefore, the study of group homomorphisms can greatly facilitate the development of fault-tolerant systems when the computations have an underlying abelian group structure.

In Figure 2-4, we visualize schematically how fault tolerance is achieved: the group homomorphism ϕ adds redundancy to the computation by mapping the abelian group G, in which the original operation took place, to a subgroup G' of a larger (redundant) group H. (It is interesting to note here that G' is exactly the subset of

[3]This is a very reasonable assumption, because it corresponds to efficient use of the elements in the redundant group H.

valid results defined earlier as H_V.) Any error e will be detected, as long as it takes us out of the subgroup G' to an element h not in G'. If enough redundancy exists in H, the error might be correctable. Details about the error detection and error correction procedure are presented in the next section.

2.3.2 Error Detection and Correction

In order to be able to detect an error $e_d \in \mathcal{E}^{(\lambda)} \subset H$, we need every possible valid result $g' \in G' \subset H$ to be an invalid result when corrupted by $e_d \neq 0_\diamond$. Mathematically, this can be expressed by:

$$\{g' \diamond e_d \mid g' \in G'\} \cap G' = \emptyset$$

If we use the set notation $G' \diamond e_d \equiv \{g' \diamond e_d \mid g' \in G'\}$, then the above equation becomes:

$$(G' \diamond e_d) \cap G' = \emptyset \tag{2.2}$$

Similarly, for an error $e_c \in \mathcal{E}^{(\lambda)}$ ($e_c \neq 0_\diamond$) to be correctable (and a fortiori detectable), we require that it satisfies the following:

$$(G' \diamond e_c) \cap (G' \diamond e) = \emptyset \quad \forall \quad e \neq e_c \in \mathcal{E}^{(\lambda)} \tag{2.3}$$

In group theory ([15] or any other standard textbook can be used as a reference), the sets $(G' \diamond e)$ for any $e \in H$ are known as *cosets* of the subgroup G' in H. Two cosets are either identical or have no elements in common. Therefore, they form an *equivalence class decomposition* (partitioning) of H into subsets[4]. This collection of cosets is denoted by H/G' and it forms a group, called the "quotient group of H under G' ", under the operation:

$$A \oplus B = \{a \diamond b \mid a \in A, b \in B\}$$

where A and B now denote sets of elements of H rather than single elements of H.

Since two cosets are either identical or have no elements in common, Equations (2.2) and (2.3) become

$$\begin{aligned} (G' \diamond e_d) &\neq G', \text{ for } e_d \neq 0_\diamond \\ (G' \diamond e_c) &\neq (G' \diamond e) \quad \forall \quad e \neq e_c \in \mathcal{E}^{(\lambda)} \end{aligned}$$

[4]In [15], an equivalence relation $\sim$ on a subset A of a set H is defined as a binary relation that satisfies the *identity*, *reflexivity*, and *transitivity* properties, that is, for all elements $a, b, c \in A$ the following hold:

- $a \sim a$ (identity)
- $a \sim b$ implies $b \sim a$ (reflexivity)
- $a \sim b$ and $b \sim c$ implies $a \sim c$ (transitivity)

The subset A is called an *equivalence class.*

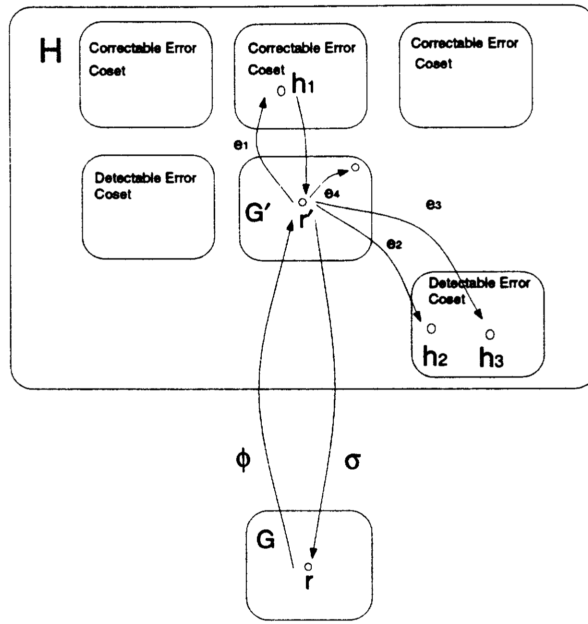


Figure 2-5: Structure of the redundant group H for error detection and correction.

Therefore, error detection and correction can proceed as shown in Figure 2-5. Any error, such as e_1, e_2, and e_3 in the figure, that takes us out of the subgroup G' to an element h_i ($i = 1, 2, 3$) of the redundant group, will be detected. Furthermore, if enough redundancy exists in H, some errors can be corrected. For example, the error e_1 that takes us to h_1 is correctable because the coset $G' \diamond e_1$ is not shared with any of the other errors e_i. Once we realize that h_1 lies in the coset of e_1, we can get the uncorrupted result $r' \in G'$ by performing the operation $h_1 \diamond e_1^{-1}$. If h_i lies in a coset shared by more than one error (which is the case for h_2 and h_3), the corresponding errors are detectable but not correctable. Errors that let the result stay within G', such as e_4, are *not* detectable.

To summarize, the correctable errors are those that force the result into distinct non-zero cosets[5]. In order for an error to be detectable, it only has to force the result into a non-zero coset.

2.3.3 Separate Codes

If we focus on the so-called *separate codes*, we can obtain some extremely useful results. Separate codes [2] are arithmetic codes in which redundancy is added in a separate "parity" channel. Error detection and correction are performed by appropriately comparing the results of the main computational channel, which performs the original operation, with the results of the parity channel, which performs a parity computation. No interaction between the operands and the parity occurs.

A simple example of a separate code is presented in Figure 2-6. The operation that we would like to protect here is integer addition. The main computational channel

[5]By a non-zero coset we mean a coset other than G'.

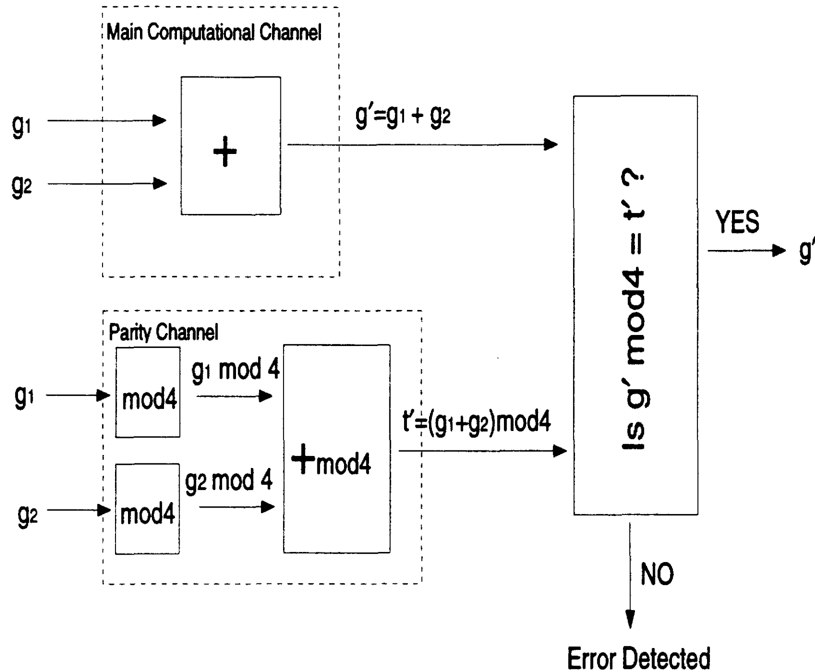


Figure 2-6: A simple example of a separate arithmetic code.

performs integer addition, whereas the parity channel performs addition modulo 4. The results of these two channels, g' and t' respectively, are compared. If they agree modulo 4, then the result of the computational channel is accepted as fault-free. If they do not agree, then an error has been detected. This figure also shows one of the important advantages of separate codes over other codes, such as the αN code shown in Figure 1-3, namely that if we know the result to be error-free then we can output it without any further processing or decoding.

When we restrict ourselves to separate codes, the computational model of Figure 2-2 reduces to the model shown in Figure 2-7. For simplicity, only two operands are shown in this figure, but the discussion that follows analyses the general case of p operands.

In the case of separate codes, the group homomorphism ϕ maps the computation in the group G to a redundant group H which is the *cartesian product* of G and a parity set that we call T, that is

$$H = G \times T$$

The homomorphic mapping satisfies $\phi(g) = [g, \theta(g)]$, where θ is the mapping that creates the parity information from the operands (refer to Figure 2-7). The set of valid results H_V is now the set of elements of the form $[g, \theta(g)]$. In [1], it is shown that T is a group and that θ is a group homomorphism from G to T.

If we require that θ is onto T[6], then the problem of finding suitable separate

[6]This is a very reasonable assumption because it corresponds to efficient use of the parity symbols

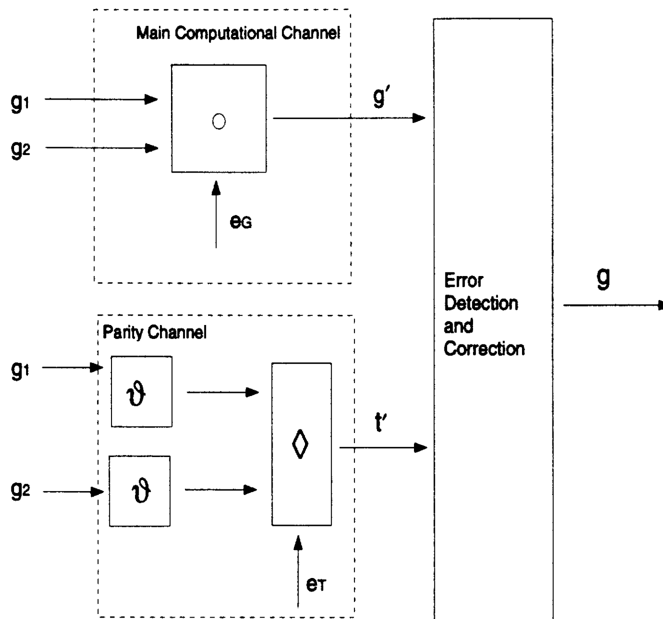


Figure 2-7: Fault-tolerant model for a group operation using a separate code.

codes reduces to the problem of finding suitable *epimorphisms*[7] θ from G onto T. A theorem from group theory [15] states that there is a one-to-one correspondence between epimorphisms θ of the abelian group G onto T and subgroups N of G. In fact, the quotient group G/N, constructed from G using N as a subgroup, provides an isomorphic image of T. Therefore, by finding all possible subgroups of G, we can find all possible epimorphisms θ from G onto T, and hence all possible parity checks.

Finding the subgroups of a group is not a trivial task but it is relatively easy for most of the groups we are interested in protecting. By finding all subgroups of a given group, we are guaranteed to find all possible separate arithmetic codes that can protect a given group computation. This is the first systematic procedure that can construct arithmetic codes in the group setting. It results in a complete characterization of the possible separate codes for a given abelian group. The result is a generalization of one proved by Peterson for the case of integer addition and multiplication [16], [17].

2.4 Applications to Other Algebraic Systems

The analysis presented so far for the group setting extends naturally to other algebraic systems with the underlying structure of a group, such as rings, fields and vector spaces [1]. By exploiting the abelian group structure in each of these other structures, and by assuming an error model that is *"additive" with respect to the group operation*, we can place the construction of arithmetic codes for computations in them into the

in the group T.

[7]In the group case, an epimorphism is equivalent to a surjective homomorphism.

group framework that we have developed.

Therefore, even though the analysis in [1] starts with a seemingly limited set of computational tasks, it can be extended sufficiently to include a variety of previously studied examples, such as Integer Residue Codes, Real Residue Codes, Multiplication of Non-Zero Real Numbers, Linear Transformation, and Gaussian Elimination [1].

A complete development of the ring-theoretic framework for computations that have an underlying ring structure as well as examples can be found in Chapter 5.

2.5 Summary

This chapter presented the group-theoretic framework that was developed in [1] for the analysis of arithmetic codes. We discussed the assumptions that were made, the error and computational models that were used, and the results that were obtained. An one-to-one correspondence between arithmetic codes and group homomorphisms was established first. Since algebraic homomorphisms are a well-studied topic in algebra, this facilitated the study of arithmetic codes and helped define a natural error detection and correction procedure, based on the construction of cosets in the redundant group. Then, a procedure for constructing separate codes for a given computation was developed, and finally, the results were extended to higher algebraic systems with an underlying abelian group structure, such as rings, fields and vector spaces.

In the next chapter, we show how the results obtained for the group case extend naturally, under the same model and assumptions, to the less constrained setting of semigroup operations.

Chapter 3

Semigroup-Theoretic Framework

3.1 Introduction

The framework developed in Chapter 2 deals with computations that have an underlying abelian group structure. In this chapter, we show how the results obtained there can be extended in a very natural way to computations with an abelian *semigroup* structure. We thereby relax the requirements that the computations need to satisfy, and have available a framework that provides a systematic algebraic approach to arithmetic coding and ABFT for a much broader class of computations.

Our analysis follows closely the analysis for the group case in [1] that was explained briefly in Chapter 2. We develop our framework in the following way. In Section 3.2, we describe the model of the semigroup computation. Then, in Section 3.3, we derive the conditions under which the mapping from the original computation to the fault-tolerant computation will correspond to a semigroup homomorphism. This places the analysis in a well defined algebraic framework. In Section 3.4 we adopt the *additive* error model, and proceed to analyze the requirements for redundancy in the semigroup case and compare the results with the corresponding results in the group case. A framework for separate codes is developed in Section 3.5 and then a constructive procedure that generates all separate codes for a semigroup computation is presented. A comparison of the constructive procedure for such codes between the semigroup and the group case is also made. Finally, Section 3.6 provides a summary of the results that were obtained and discusses the tradeoffs that emerged in transitioning from computations with an abelian group structure to computations with an abelian semigroup structure. The treatment of specific classes of semigroups is deferred to Chapter 4.

3.2 Computation in a Semigroup

This section defines the model that we use for performing a computation with an underlying semigroup structure. We first provide an introduction to the notion of a semigroup and then proceed to the computational and error models.

3.2.1 Introduction to Semigroups

The following is the definition of a semigroup (taken from [18]):

Definition: A **semigroup** $\mathcal{S} = (S, \circ)$ is an algebraic system that consists of a set of elements S, and a binary operation $\circ$, called the semigroup product, such that the following properties are satisfied:

1. For all $s_1, s_2 \in S$, $s_1 \circ s_2 \in S$ (closure).
2. For all $s_1, s_2, s_3 \in S$, $(s_1 \circ s_2) \circ s_3 = s_1 \circ (s_2 \circ s_3)$ (associativity).

If the binary operation $\circ$ is obvious from the context, then we usually denote the semigroup simply by S. If the operation $\circ$ is commutative, that is:

$$s_1 \circ s_2 = s_2 \circ s_1 \text{ for all } s_1, s_2 \in S$$

then the semigroup is called an *abelian* (or commutative) semigroup.

A familiar example of a semigroup that is not a group is the set of positive integers under the operation of addition, usually denoted by $(\mathbf{N}, +)$. The two properties above can be verified easily since addition is an associative operation. In fact $(\mathbf{N}, +)$ is an abelian semigroup (since addition is a commutative operation). Other examples of abelian semigroups that are not groups are the set of integers under multiplication (usually denoted by $(\mathbb{Z}, \times)$), and the set of polynomials with real coefficients under the operation of polynomial multiplication. Examples of non-abelian semigroups are the set of $N \times N$ matrices under matrix multiplication, and the set of polynomials with real coefficients under the operation of polynomial substitution. More examples of semigroups can be found in [18].

From the above definition and examples, one sees that a semigroup, unlike a group, is an algebraic structure with minimal requirements: other than closure and associativity, there is not any other requirement. However, under certain assumptions, this minimal structure allows us to achieve extremely useful results. In order to proceed to the examination of these results we will need some more terminology.

If an element $0_\circ \in S$ exists such that:

$$s \circ 0_\circ = 0_\circ \circ s = s \text{ for all } s \in S$$

then the element $0_\circ$ is called the *identity* element and the semigroup is called a *monoid.* For example, the semigroup $(\mathbf{N} \cup \{0\}, +)$ (usually denoted by $(\mathbf{N}_0, +)$) is a monoid with 0 as the identity element. In fact, any semigroup that does not possess an identity element can be made artificially into a monoid simply by *adding* an element $0_\circ$ to it, and *defining* that element to behave just like an identity element. The union of the original semigroup S with this identity element is a monoid. Note that the identity can easily be shown to be unique (see for example [18]) and therefore the addition of an identity element is well-defined and it is only possible when the semigroup is not a monoid.

Since any semigroup can be easily modified to be a monoid, we will assume without loss of generality that we are dealing with a monoid. For the rest of this chapter we restrict our attention specifically to *abelian monoids.*

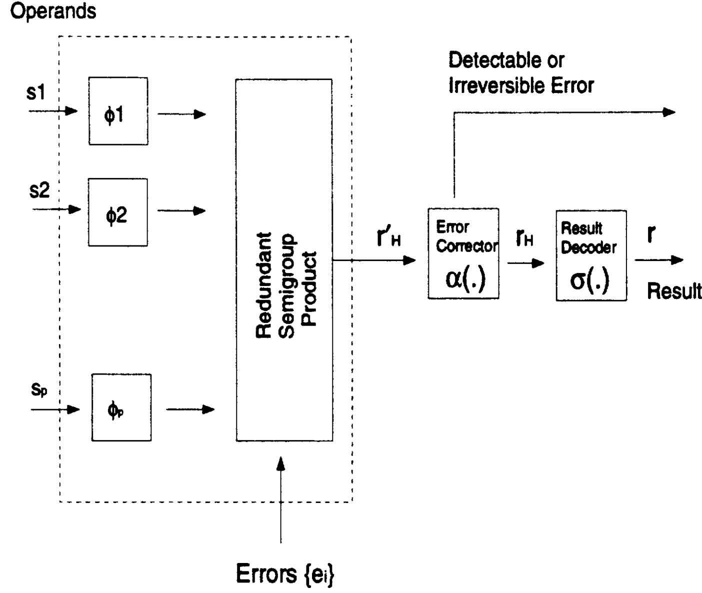


Figure 3-1: Fault-tolerant model for a semigroup computation.

3.2.2 Computational Model for Semigroup Operations

We assume that the computation we want to protect can be modeled as a semigroup operation with p operands:

$$r = s_1 \circ s_2 \circ \ldots \circ s_p$$

where r is the desired fault-free result. The model of the fault-tolerant computational unit can be seen in Figure 3-1. The operands $s_1, s_2, \ldots, s_p$ are encoded via the encoders $\{\phi_i\}$, which map the operands to a higher order monoid $(H, \diamond)$. The computation takes place in the redundant computational unit where additive (or other) errors $\{e_i\}$ might be introduced.

The output of the redundant computation unit is a possibly corrupted result which we denote by r'_H. The error corrector, through the use of the error correcting mapping α, tries to map r'_H to the fault-free redundant result r_H. However, this might not be always possible, as for example when the error that took place is irreversible. In such a case, the error corrector signals that a detectable or an irreversible error took place. Finally, the decoder maps the fault-free redundant result r_H to its desirable, unencoded form r. The decoding mapping is denoted by $\sigma : H_V \longmapsto S$ and maps the elements in the set of valid results (which we call H_V) back to the original monoid S, that is

$$r = s_1 \circ s_2 \circ \ldots \circ s_p = \sigma(r_H)$$

where $r_H = \phi_1(s_1) \diamond \phi_2(s_2) \diamond \ldots \diamond \phi_p(s_p)$.

Just like the model for the group case, an implicit assumption in this diagram is that the encoders, the error corrector and the result decoder are fault-free. If necessary, they have to be protected using modular redundancy.

If we adopt the additive error model that was introduced in Chapter 2, the errors

$\{e_i\}$ will be elements of H. We assume that they corrupt r_H in an additive fashion. The possibly corrupted redundant result r'_H is given by:

$$\begin{aligned} r'_H &= \phi_1(s_1)\diamond\phi_2(s_2)\diamond\ldots\diamond\phi_p(s_p)\diamond e_1\diamond e_2\diamond\ldots\diamond e_\lambda \\ &= r_H\diamond e_1\diamond e_2\diamond\ldots\diamond e_\lambda \end{aligned}$$

The underlying assumptions of the additive error model are that errors are independent of the operands, and that the effect of any error on the overall result is independent of the point in time where it took place during the computation. This last assumption is realistic because we have limited ourselves to *associative and abelian* operations. As mentioned in Chapter 2, in such a case, the above expression is well-defined and its result is the same, irrespective of the order and the position in which the operands are evaluated.

At this point, we would like to make an important distinction between the *actual* faults that influence the result of the computation and the *modeled* errors $\{e_i\}$ that we use in our additive error model. An actual fault is a hardware failure and can have any effect on the result. More specifically, it need not affect the result in an inherently "additive" way. However, as long as we can *model* the effect of a fault on the result as an additive error, then we expect the additive error model to be valid. As an example, consider a computation unit that performs addition modulo 64. It represents each valid result as a 6-bit binary number. Furthermore, assume that a single fault forces a 1 ("high") at one of the six binary digits. Then, a single fault produces an additive error of the form 2^i for $i \in \{0,1,2,3,4,5\}$. If our additive error model protects against errors of the above form, then it effectively protects against any single fault that takes place during the computation. Of course, the more closely we can exploit the actual error structure, the more efficient our fault-tolerant scheme will be.

The error correcting mapping α in Figure 3-1 should map r'_H to r_H, so that $r_H = \alpha(r'_H)$ (except when this is not possible). We can simplify notation if we define the overall error $e = e_1\diamond e_2\diamond\ldots\diamond e_\lambda$:

$$r'_H = r_H \diamond e$$

where $e \in \mathcal{E}^{(\lambda)} = \mathcal{E} \diamond \mathcal{E} \diamond \ldots \diamond \mathcal{E}$ (λ times). If no error took place, e is the identity element.

We draw attention now to the important distinction between the group and the semigroup cases. In a semigroup setting, inverses are not guaranteed[1]. If an error e occurs, then the result is not necessarily correctable. Even if we identify the error e that occurred, we might not be able to correct the corrupted result r'_H. In the group case this was not a problem; since e was invertible, all we needed to do after identifying the exact error that occurred was to compose the corrupted result r'_H with the inverse

[1]An element $s^{-1} \in S$ is the inverse of $s \in S$ if and only if $s \circ s^{-1} = s^{-1} \circ s = 0_\circ$. Of course, this can only be defined in the monoid case.

e^{-1} of the error:

$$r_H = r'_H \circ e^{-1}$$

However, when the error is not invertible, we cannot in general expect anything more than detecting that error. If the error is invertible, then we can correct it provided that enough redundancy exists in our coding.

Other Error Models

Throughout our analysis in this chapter, we adopt the additive error model which was used in [1] and was presented in Chapter 2. However, it is important to clarify at this point that the error model does not really impact most of the results that we obtain in the group- and semigroup-theoretic frameworks. For example, the use of algebraic homomorphisms in these frameworks does not depend in any way on whether the error model is additive or not. In fact, the only point where the error model enters the analysis is when we consider the redundancy requirements and the error-correcting capabilities of our codes (refer to Section 3.4).

The additive error model has some advantages when considering the redundancy requirements in the group case (see [1], as well as the brief discussion in Section 3.4), but it is not clear that this is the best choice, especially when we move to higher algebraic structures with additive as well as multiplicative operations. Moreover, hardware considerations might make the choice of a different error model more sensible.

3.3 Use of Semigroup Homomorphisms

The computational model described in the previous section achieves fault tolerance by mapping (through the $\{\phi_i\}$ mappings) the computation in the original abelian monoid $\mathcal{S} = (S, \circ)$ to a *redundant* computation in a larger monoid $\mathcal{H} = (H, \diamond)$. The additional elements of H will be used to detect and correct erroneous results. Under the additive error model, faults introduce errors that have an additive effect on the result.

The analysis in this section investigates the properties that the set of mappings $\{\phi_i\}$ needs to have in order to satisfy the computational model that we have adopted. We show that under a few reasonable assumptions, all mappings have to be the same. Furthermore, this mapping is a semigroup homomorphism.

Let $\{\phi_1, \phi_2, ..., \phi_p\}$ be the set of mappings used to map elements of S to elements of H (refer to Figure 3-1):

$$\begin{aligned} \phi_1 &: S \longmapsto H \\ \phi_2 &: S \longmapsto H \\ &\quad \cdot \\ &\quad \cdot \\ \phi_p &: S \longmapsto H \end{aligned}$$

Then the computation of the product $s_1 \circ s_2 \circ ... \circ s_p$ in S takes place in H in the following form:

$$r'_H = \phi_1(s_1) \diamond \phi_2(s_2) \diamond ... \diamond \phi_p(s_p) \diamond e$$

where r'_H is a possibly corrupted result in H and $e \in \mathcal{E}^{(\lambda)}$.

When r'_H is not corrupted (let us denote the uncorrupted result in H by r_H), we require that it maps back to the correct result in the original set S via the mapping σ (refer again to Figure 3-1):

$$r = s_1 \circ s_2 \circ ... \circ s_p = \sigma(r_H) = \sigma(\phi_1(s_1) \diamond \phi_2(s_2) \diamond ... \diamond \phi_p(s_p)) \tag{3.1}$$

Now we have the tools to prove the following claim:

Claim: When S and H are monoids, under the assumptions that:

1. The mapping σ is one-to-one, and
2. $\phi_1(0_\circ) = \phi_2(0_\circ) = ... = \phi_p(0_\circ) = 0_\diamond$ (that is, all encodings map the identity of S to the identity of H),

all ϕ_i's have to be equal to the same semigroup homomorphism ϕ. Moreover for all $s \in S$,

$$\sigma^{-1}(s) = \phi_1(s) = \phi_2(s) = ... = \phi_p(s) \equiv \phi(s)$$

Proof: The mapping σ maps elements of H_V to S:

$$\sigma : H_V \longmapsto S$$

where H_V is the set of valid results in H defined as:

$$H_V = \{\phi_1(s_1) \diamond \phi_2(s_2) \diamond ... \diamond \phi_p(s_p) \mid s_1, s_2, ..., s_p \in S\} \tag{3.2}$$

By our first assumption, σ is one-to-one; therefore, its inverse function:

$$\sigma^{-1} : S \longmapsto H_V$$

is well-defined. Moreover, from (3.1):

$$\sigma^{-1}(s_1 \circ s_2 \circ ... \circ s_p) = \phi_1(s_1) \diamond \phi_2(s_2) \diamond ... \diamond \phi_p(s_p) \tag{3.3}$$

When $s_i = 0_\circ$ for all i except $i = 1$ we have:

$$\begin{aligned} \sigma^{-1}(s_1) &= \phi_1(s_1) \diamond \phi_2(0_\circ) ... \diamond \phi_p(0_\circ) \\ &\Rightarrow \\ \sigma^{-1}(s) &= \phi_1(s), \ s \in S \end{aligned}$$

since $\phi_i(0_\circ) = 0_\diamond$ was part of our second assumption.

Similarly, we get:

$$\sigma^{-1}(s) = \phi_i(s), \text{ for all } s \in S, \ i = 1, 2, ..., p$$

Therefore, $\sigma^{-1}(s) = \phi_1(s) = \phi_2(s) = ... = \phi_p(s)$, for all $s \in S$. Let $\phi \equiv \sigma^{-1} = \phi_1 = \phi_2 = ... = \phi_p$. Then, from (3.3) we get:

$$\phi(s_1 \circ s_2 \circ ... \circ s_p) = \phi(s_1) \diamond \phi(s_2) \diamond ... \diamond \phi(s_p)$$

If we let $s_i = 0_\circ$ for $i = 3, 4, ..., p$, we have the defining property of an algebraic homomorphism:

$$\phi(s_1 \circ s_2) = \phi(s_1) \diamond \phi(s_2)$$

Therefore, all $\{\phi_i\}$ are equal to the semigroup homomorphism ϕ (or, equivalently, to σ^{-1}). $\surd$

We have shown that, under the assumptions that the decoding mapping σ is one-to-one, and that each encoding mapping ϕ_i maps the identity element of S to the identity element of H, the mappings $\{\phi_i\}$ turn out to be the same semigroup homomorphism, which we will call ϕ. Moreover, ϕ is simply the inverse of σ. Note that the derivation requires neither the error model to be additive nor the operation $\diamond$ to be abelian.

The question that still remains to be answered is whether these assumptions are reasonable:

- By requiring that the decoding mapping σ be one-to-one, we essentially require that it is not many-to-one (since it does not make sense to have one-to-many or many-to-many "decoding" mappings). If σ was many-to-one, then different operands that arrive at the same result s ($s \in S$) would produce different results r_H ($r_H \in H_V$, where H_V is the set of valid results). In such a case, redundancy would be utilized in an inefficient way, since each way of arriving at a result would involve different redundancy conditions. Therefore, the requirement that σ is a one-to-one mapping is essentially a requirement that the fault-tolerant arithmetic code is efficient.

- Now suppose that we need to operate on less than p operands. We would like to treat such a calculation in exactly the same way as a calculation with p operands. The natural way to do that would be to use the identity element for operands that do not appear in the calculation. For example, if we want to operate on only 3 operands, we would actually be calculating the following:

  $$s_1 \circ s_2 \circ s_3 \circ 0_\circ \circ 0_\circ \circ ... \circ 0_\circ$$

  Therefore, it seems reasonable to require that

  $$\phi_1(s_1) \diamond \phi_2(s_2) \diamond ... \diamond \phi_i(s_i) = \phi_1(s_1) \diamond \phi_2(s_2) \diamond ... \diamond \phi_i(s_i) \diamond \phi_{i+1}(0_\circ) \diamond ... \diamond \phi_p(0_\circ)$$

  for each $i \in \{1, 2, ..., p\}$. A stronger requirement (but one that simplifies things a lot) would be $\phi_i(0_\circ) = 0_\diamond$. This requirement does restrict the number of mappings; on the other hand, however, it is efficient because the redundancy in the mapping is independent of the number of operands.

3.4 Redundancy Requirements

In this section we adopt the *additive error model* in order to analyze the redundancy requirements on the encoding mapping. The elements of the original semigroup S are mapped through ϕ to elements of the redundant semigroup H. We denote detectable errors by $e_d \in E_d$ and correctable errors by $e_c \in E_c$. Note that correctable errors are also detectable.

Using the same approach as the approach in Chapter 2 and in [1], we conclude that in order for an error e_d to be detectable, we need the following condition to be satisfied:

$$r_H \diamond e_d \neq h \text{ for all } h \neq r_H,\ h, r_H \in H_V,\ e_d \in E_d$$

which essentially requires that any detectable error e_d corrupts the actual result r_H in an *additive* fashion such that it takes us out of the set of valid results (otherwise, we would think that no error took place). In order for an error e_c to be correctable, the following condition needs to be satisfied:

$$r_H \diamond e_c \neq h \diamond e_d \text{ for all } h \neq r_H,\ h, r_H \in H_V,\ e_c \in E_c,\ e_d \in E_d$$

which essentially requires that any correctable error corrupts the actual result in a unique way that no other error shares (remember that detectable errors are also correctable).

Now, suppose that $\mathcal{E} = \{e_i\}$ is the set of possible single errors (including the identity $0_\diamond$) and that our objective is to detect a total number of D errors and to correct a total number of C errors ($C < D$). Then, $e_d \in \mathcal{E}^{(D)}$ and $e_c \in \mathcal{E}^{(C)}$. The above redundancy conditions can then be combined into one:

Redundancy Condition: In order to detect D errors and correct C errors, the structure of the redundant semigroup H has to satisfy the following requirement:

$$h_1 \diamond e_d \neq h_2 \diamond e_c \text{ for all } h_1 \neq h_2,\ h_1, h_2 \in H_V,\ e_c \in \mathcal{E}^{(C)},\ e_d \in \mathcal{E}^{(D)}$$

where H_V (defined in (3.2)) is a subsemigroup of H[2].

If we use the set notation $x \diamond A = \{xa \mid a \in A\}$, we can simplify the above expression:

$$(h_1 \diamond \mathcal{E}^{(D)}) \cap (h_2 \diamond \mathcal{E}^{(C)}) = \emptyset \text{ for all } h_1 \neq h_2, h_1, h_2 \in H_V \tag{3.4}$$

Because semigroups have a weaker structure than groups, the analysis cannot follow the exact same path as the analysis for the group case in [1]. In particular, the coset-based error detection and correction that was developed in the group case (Figure 2-5) is typically not possible now. However, under the assumption that *both* detectable and correctable errors are invertible, we can transform the above into a

[2]The fact that H_V is a subsemigroup of H can be proved easily. The proof is straightforward and uses the fact that the mapping ϕ is a homomorphism.

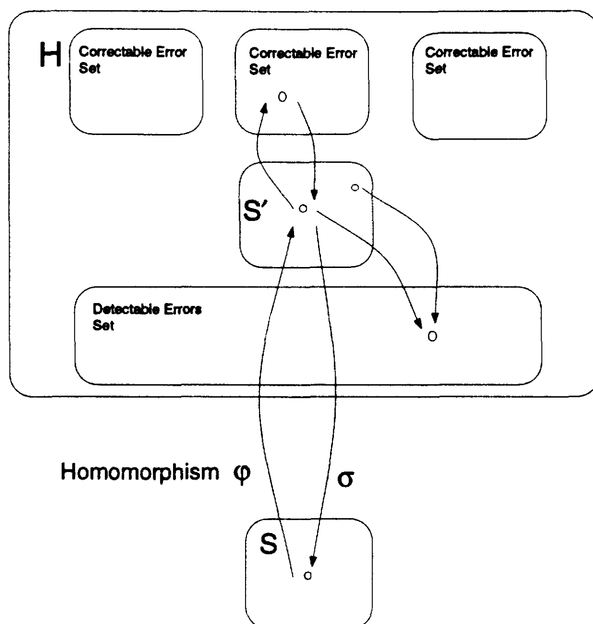


Figure 3-2: Structure of the redundant semigroup for error detection and correction.

more intuitive form as follows:

$$
\begin{array}{rcll}
h_1 \diamond e_d & \neq & h_2 \diamond e_c & \text{for all } h_1 \neq h_2 \in H_V,\ e_c \in \mathcal{E}^{(C)},\ e_d \in \mathcal{E}^{(D)} \\
h_1 \diamond e_d \diamond e_c^{-1} & \neq & h_2 & \text{for all } h_1 \neq h_2 \in H_V,\ e_c \in \mathcal{E}^{(C)},\ e_d \in \mathcal{E}^{(D)} \\
h_1 \diamond e_d \diamond e_c^{-1} & \neq & h_2 & \text{for all } h_1, h_2 \in H_V,\ e_c \in \mathcal{E}^{(C)},\ e_d \in \mathcal{E}^{(D)} \text{ so that } e_d \diamond e_c^{-1} \neq 0_\diamond \\
h_1 \diamond e_d \diamond e_c^{-1} & \neq & h_2 & \text{for all } h_1, h_2 \in H_V,\ e_c \in \mathcal{E}^{(C)},\ e_d \in \mathcal{E}^{(D)},\ e_c \neq e_d
\end{array}
$$

Using set notation we can write the above expression as:

$$(H_V \diamond e_c) \cap (H_V \diamond e_d) = \emptyset \text{ for all } e_c \in \mathcal{E}^{(C)},\ e_d \in \mathcal{E}^{(D)},\ e_c \neq e_d \tag{3.5}$$

Note that the above equation is quite different from Equation (3.4): under the assumption that all errors are invertible, we only need to check that, for each pair of *different* correctable and detectable errors, each error "shifts" the set of valid results H_V to a different set in the redundant space. In Equation (3.4) we do a similar check for every pair of different operands. The two situations are quite different since, in general, we expect that the set of operands contains many more elements than the set of errors.

Equation (3.5) parallels the result that was shown in Chapter 2 for the group case. However, since the concept of a coset cannot be extended to the general semigroup case[3], the coset-based error detection and correction cannot be extended to the semigroup case. Figure 3-2 shows the structure of the redundant semigroup. The major difference of the semigroup from the group case is that error correction is now based on *disjoint sets* of elements (one for each different error), whereas in the group case

[3]The lack of inverses causes this problem.

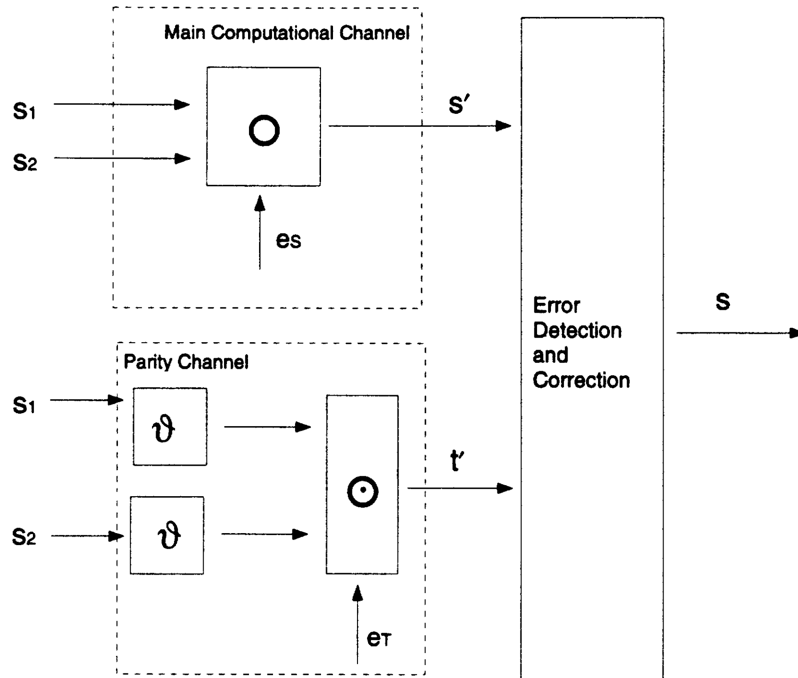


Figure 3-3: Model of a fault-tolerant system that uses separate codes.

error correction is based on having a different *coset* for each error. This evidently makes things a lot more complicated in checking for sufficient redundancy requirements in the semigroup case. However, as the examples that we present in Chapter 4 show, we now have more flexibility in choosing arithmetic coding schemes.

3.5 Separate Codes for Semigroups

In this section, we focus on the special case of separate codes. We will show that, by restricting ourselves to these codes, we can obtain extremely interesting results that parallel the ones obtained for separate codes in the group case:

- A complete characterization of separate codes will be possible.
- Moreover, we will be able to outline a systematic algebraic procedure that can generate all possible such codes.

As explained in Chapter 2, separate codes are a class of arithmetic codes that protect computation using an independent "parity" channel. No interaction between the original codeword and the parity information occurs during computation. Figure 3-3 shows a typical example of a computational unit that achieves fault tolerance through a separate scheme. For simplicity of presentation, in this figure, as well as in the discussion that follows, we focus on separate codes with two operands. The analysis can be extended easily to the general case of a system with p operands.

3.5.1 Description of the Model for the Codes

The codeword of a separate code consists of two parts: the original unencoded data (which is operated upon by the main computational channel in the exact same way as the original data) and the "parity" information (which provides the necessary redundancy for fault tolerance, and which is operated upon by the "parity" channel). As shown in Figure 3-3, errors can now take place during computation in the main channel (as for example e_S), or the parity channel (for example e_T), or in both[4]. As was the case in the general analysis of a semigroup computation, an implicit assumption of the system in Figure 3-3 is that the parity encoders (denoted by the mapping θ), as well as the error detector and corrector, are error-free[5].

It has already been shown in this chapter, that any arithmetic code for a computation with an underlying semigroup structure corresponds to a homomorphism ϕ. For the special case of a separate code, the homomorphism ϕ is as follows:

$$S \overset{\phi}{\longmapsto} H = S{\times}T$$

where the only difference now is that H is the *cartesian product* of two other semigroups, the original semigroup S and the *parity* semigroup T. Therefore, we can denote the mapping ϕ as the pair:

$$\phi(s) = [s, \theta(s)] \tag{3.6}$$

where $[s, \theta(s)]$ is an element of H (in fact, an element of $H_V \subset H$, which was defined as the subsemigroup of valid results).

3.5.2 Analysis of the Parity Encoding

Let us now analyze the structure of the mapping θ that is used to construct the parity of a codeword. We use the symbol $\circ$ to denote the binary operation in the semigroup S, the symbol $\diamond$ to denote the binary operation in the semigroup H and the symbol $\odot$ to denote the binary operation in the semigroup T. The relation between these three binary operations is straightforward. To see this, let

$$\begin{aligned} h_1 &= \phi(s_1) = [s_1, \theta(s_1)] \\ h_2 &= \phi(s_2) = [s_2, \theta(s_2)] \end{aligned}$$

where $h_1, h_2 \in H$, $s_1, s_2 \in S$, and $\theta(s_1), \theta(s_2) \in T$. By definition of separate codes, the computations in the main and the parity channels do not interact. Therefore, under error-free operation we require that

$$h_1 \diamond h_2 = [s_1, \theta(s_1)] \diamond [s_2, \theta(s_2)] = [s_1 \circ s_2, \theta(s_1) \odot \theta(s_2)] \tag{3.7}$$

[4]Although the errors in Figure 3-3 are modeled as additive, this assumption is again unnecessary for the analysis in this section.

[5]If these subsystems are not reliable enough, we can use N-Modular redundancy to protect them.

We now have the necessary tools to prove the following claim:

Claim: If the mapping ϕ is a homomorphism, then the mapping θ has to be a homomorphism as well.

Proof: Using the fact that the mapping ϕ is a homomorphism and the defining Equation (3.7), we get:

$$\begin{aligned} \phi(s_1 \circ s_2) &= \phi(s_1) \diamond \phi(s_2) \\ &= [s_1 \circ s_2, \theta(s_1) \odot \theta(s_2)] \end{aligned}$$

Moreover, by definition of θ in Equation (3.6):

$$\phi(s_1 \circ s_2) = [s_1 \circ s_2, \theta(s_1 \circ s_2)]$$

Therefore, for all s_1, s_2 in S:

$$\theta(s_1 \circ s_2) = \theta(s_1) \odot \theta(s_2) \tag{3.8}$$

Equation (3.8) is the defining property of a homomorphism, so we conclude that θ is a homomorphism. $\surd$

3.5.3 Determination of Possible Homomorphisms

Having established that θ is a homomorphism, we realize that in order for the corresponding separate code to be efficient, we need to require that θ is onto T. The reason is simply to ensure that the arithmetic code makes use of all elements of the parity semigroup T. Therefore, this essentially requires that efficient use of the parity symbols is made and none of them is wasted.

An onto homomorphism is called a *surjective homomorphism*[6]. When we restrict θ to be a surjective homomorphism, we have a systematic algebraic way of generating all possible separate codes. In the process of developing this method, we make use of the following definitions and theorems[7] in [18]:

Definition: An equivalence relation $\sim$ on the elements of a semigroup S is called a *congruence relation* if it is compatible with $\circ$ (the binary operation of S), that is:

$$\text{For } a, b, a', b' \in S \text{ we have : If } a \sim a', b \sim b' \Rightarrow a \circ b \sim a' \circ b'$$

An equivalence class under a congruence relation is called a *congruence class.* Let the set $S/\!\sim$ denote the set of congruence classes of S under the congruence relation $\sim$.

[6]Many authors, including [18], like to use the name *epimorphism* for a surjective homomorphism. However, strictly speaking, an epimorphism from a semigroup S to a semigroup H is a homomorphism ϕ such that for any two homomorphisms $\theta_1 : H \mapsto T$ and $\theta_2 : H \mapsto T$, if for all $s \in S$ $\theta_1(\phi(s)) = \theta_2(\phi(s))$, then $\theta_1 = \theta_2$. Any surjective homomorphism is an epimorphism, but the converse is false in the semigroup case [19]. A discussion on this issue as well as counterexamples can also be found in [20]. We will be using the correct term *surjective homomorphism* instead of *epimorphism*, but keep in mind that the two terms are sometimes used interchangeably in the literature.

[7]In [18], the claim and the theorem that we use and prove here are stated without a proof.

For the congruence classes $[a], [b]$ (congruence class $[a]$ is the congruence class that contains the element a) we define the following binary operation:

$$[a] \otimes [b] = [a \circ b]$$

Claim: Operation $\otimes$ is well-defined and $S/\sim$ is a semigroup.

Proof: If $[a] = [a']$ and $[b] = [b']$ then $a \sim a'$ and $b \sim b'$. Therefore, $a \circ b \sim a' \circ b'$ which means that $a \circ b$ and $a' \circ b'$ belong to the same congruence class. Moreover, operation $\otimes$ inherits associativity from $\circ$. We conclude that $S/\sim$ is a semigroup. $\surd$

Theorem: Let $\theta : S \longmapsto T$ be a surjective homomorphism. Let $\sim$ be defined by:

$$x \sim y \Leftrightarrow \theta(x) = \theta(y)$$

Then $\sim$ is a congruence relation and $S/\sim$ is isomorphic to T. Conversely, if $\sim$ is a congruence relation on S, then the mapping $\pi : S \longmapsto S/\sim$ such that for $s \in S$

$$\pi(s) = [s]$$

is a surjective homomorphism.

Proof: First, we prove the first statement. Let the binary operation in S be $\circ$, the binary operation in T be $\odot$, and the binary operation in $S/\sim$ be $\otimes$. Furthermore, let:

$$\begin{aligned} x_1 &\sim y_1 \\ x_2 &\sim y_2 \end{aligned}$$

From the definition of $\sim$ we have:

$$\begin{aligned} \theta(x_1) &= \theta(y_1) \\ \theta(x_2) &= \theta(y_2) \end{aligned}$$

So,

$$\begin{aligned} \theta(x_1) \odot \theta(x_2) &= \theta(y_1) \odot \theta(y_2) \\ \theta(x_1 \circ x_2) &= \theta(y_1 \circ y_2) \\ &\Rightarrow \\ x_1 \circ x_2 &\sim y_1 \circ y_2 \end{aligned}$$

Therefore, $\sim$ is a congruence relation.

Now, we need to prove that $S/\sim$ is isomorphic to T. All we need to do is find a mapping $\psi : S/\sim \ \longmapsto T$ that is a bijective (one-to-one and onto) homomorphism. In fact, the following mapping will do:

$$\psi([x]) = \theta(x) \text{ for some } x \in [x] \tag{3.9}$$

Note that this mapping is one-to-one and onto:

- Let $[x]$ be an element of $S/\sim$. For all $x \in [x]$, $\theta(x)$ is the same (by definition of the congruence relation $\sim$), therefore each $[x]$ maps to one element of $\theta(x) \in T$.
- Since θ is onto, each element of T has a non-empty inverse image in S. That is, for all $t \in T$ there exists $s \in S$ such that $t = \theta(s)$. Therefore, there exists at least one congruence class, namely $[s] \in S/\sim$ such that $\psi([s]) = \theta(s) = t$. The mapping ψ is onto.
- If $[x]$ and $[y]$ map to the same element of T, then $\theta(x) = \theta(y)$. Evidently, in such a case, $[x] = [y]$ (by the definition of $\sim$).

Now, we proceed to show that ψ is a homomorphism. For $[x], [y] \in S/\sim$ we have:

$$\begin{aligned} \psi([x]) \odot \psi([y]) &= \theta(x) \odot \theta(y) \\ &= \theta(x \circ y) \\ &= \psi([x \circ y]) \\ &= \psi([x] \otimes [y]) \end{aligned}$$

where we have used in the first step the definition of ψ from (3.9), in the second step the fact that θ is a homomorphism, in the third step the definition of ψ again, and in the fourth step the defining property of $S/\sim$.

We have proved that ψ is a bijective (one-to-one and onto) homomorphism. Therefore, we may conclude that $T \cong S/\sim$.

The proof of the converse is less work. The mapping π is clearly onto: for any congruence class $[x] \in S/\sim$ we know that $\pi(x) = [x]$, so there is at least one element that maps to $[x]$. Moreover, for $s_1, s_2 \in S$:

$$\begin{aligned} \pi(s_1) \otimes \pi(s_2) &= [s_1] \otimes [s_2] \\ &= [s_1 \circ s_2] \\ &= \pi(s_1 \circ s_2) \end{aligned}$$

Therefore, π is a surjective homomorphism.

At this point, the proof of the theorem is complete. $\surd$

The above theorem states that all surjective homomorphic images T of an abelian semigroup S can be found (up to isomorphism) by finding all congruence relations $\sim$ within S. Therefore, all separate codes that protect a computation in a semigroup S can be enumerated by finding all congruence relations $\sim$ that exist within S. In each case, the semigroup T that provides the parity information is isomorphic to $S/\sim$ and H is isomorphic to $S \times S/\sim$. Of course, the enumeration of all congruence relations of a semigroup might not be an easy task. However, we still have a complete characterization of all separate codes for a computation in a semigroup and, moreover, we have an algebraic framework that we can use in order to enumerate or search through all possible such codes for a given semigroup computation. Examples of such techniques and ways to approach this problem can be seen in Chapter 4.

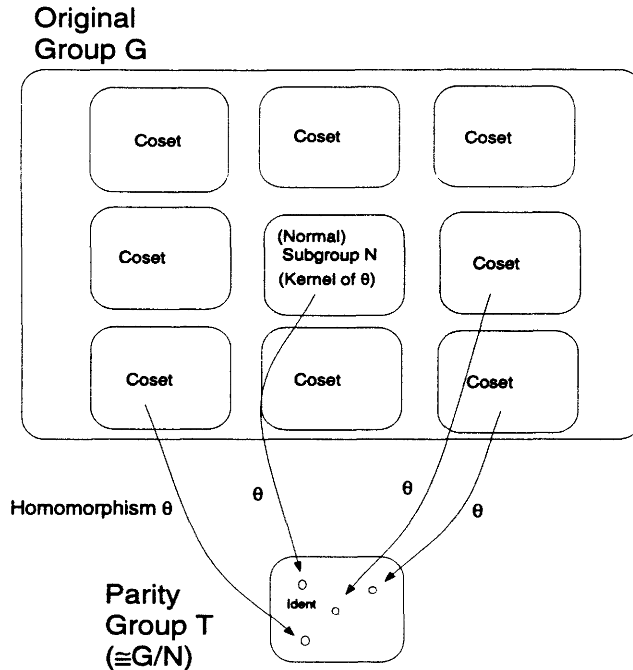


Figure 3-4: Structure of the parity group in separate codes.

3.5.4 Comparison with the Group Case

When comparing the results above with the separate code case for a computation with an underlying abelian group structure, we see a major difference: in the group case, finding a (normal[8]) subgroup N of the original group G completely specifies the homomorphism θ in the sense that the inverse images of the elements of the parity group T are exactly the cosets of G with respect to the subgroup N. Figure 3-4 shows the structure of the parity group T which is isomorphic to G/N. The normal subgroup N corresponds to the inverse image of the identity of T (defined as the *kernel* of the homomorphism θ).

In the more general setting of a semigroup, however, specifying the kernel of a homomorphism θ does not completely determine the structure of the parity semigroup T. In order to define a homomorphism from a semigroup S onto a semigroup T (or, equivalently, in order to define a congruence relation $\sim$ on a semigroup S) we need to specify *all* congruence classes. Each congruence class is the inverse image of an element of the semigroup T under the homomorphism θ.

In [21], it is shown that congruence classes correspond to (normal) *complexes*. The congruence class that is the inverse image of the identity of T (that is, the kernel of the homomorphism θ) is a (normal) *subsemigroup*. The actual definitions of a (normal)

[8]Since we are dealing with abelian groups and semigroups, the term *normal* is not really necessary. We put it in parentheses though, in order to emphasize that in most group and semigroup theory, these facts are stated for normal subgroups, normal subsemigroups and normal complexes. Note also that, even in the abelian monoid case we are dealing with, a subsemigroup is not necessarily a normal subsemigroup.

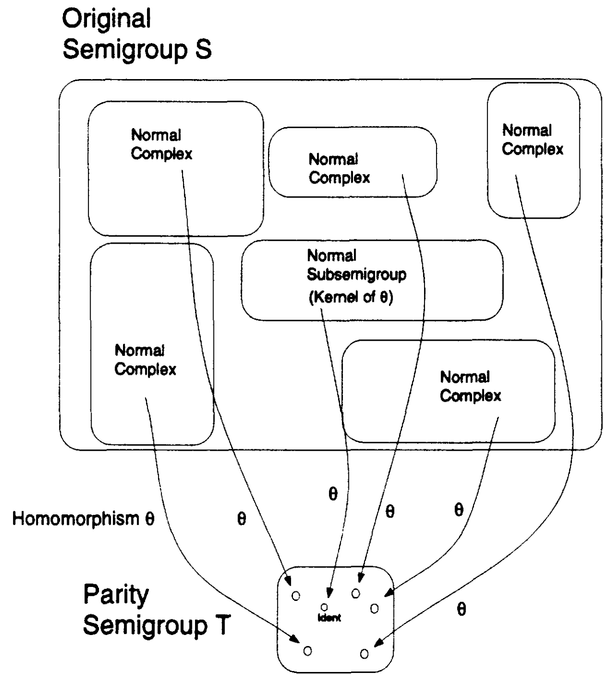


Figure 3-5: Structure of the parity semigroup in separate codes.

complex and a (normal) subsemigroup, adjusted here from [21] for the abelian monoid case, provide a clearer picture of what a congruence class is:
Normal Complex: A nonempty subset C of an abelian monoid $(S, \circ)$ is called a normal complex if, for any $x \in S$ and for any $k, k' \in C$, $x \circ k \in C$ always implies $x \circ k' \in C$.
Normal Subsemigroup: A nonempty subset C of an abelian monoid $(S, \circ)$ is called a normal subsemigroup if, for any $x \in S$ and for any $k, k' \in C$ or being empty symbols, $x \circ k \in C$ always implies $x \circ k' \in C$.
Note that, in the abelian monoid case we are focusing on, a normal subsemigroup is simply a normal complex that contains the identity element of the monoid[9] [21]. However, as we will see in the examples of Chapter 4, a subsemigroup which is a normal complex is not necessarily the normal subsemigroup.

Figure 3-5 shows the structure of the parity semigroup. The normal subsemigroup does not necessarily define the normal complexes; they have to be defined separately. Clearly, this makes the search for separate codes in the semigroup setting much harder than the search for such codes in the group setting. However, as some of the examples that are presented in Chapter 4 show, we actually have a greater variety of possible separate codes and more freedom to choose the kind of code that we want.

[9] In fact, we do not need the monoid to be abelian for this.

3.6 Summary

In this chapter we developed a semigroup-theoretic framework for studying arithmetic codes for semigroup computations. We have thereby successfully extended the group-theoretic framework that was developed in [1] to a much more general setting. The hope is to be able to use this generality to provide fault tolerance to non-linear systems, such as non-linear signal processing applications and matrix multiplication. Due to the lack of inverses, these systems fit more easily into a semigroup, rather than a group framework.

More specifically, in this chapter we have showed that arithmetic codes that provide fault tolerance to a semigroup computation need to be a semigroup homomorphism. Through this result, we established an algebraic framework for the study of arithmetic codes not only at the semigroup level (refer to the examples in Chapter 4) but also for higher semigroup-based algebraic structures (refer to the *semiring-theoretic* framework of Chapter 5). Moreover, in the special case of separate codes we were able to construct a procedure that enumerates all such codes for a given semigroup computation.

As the next chapter shows, there is always a tradeoff. A semigroup is less structured and allows for more possibilities when developing arithmetic codes. On the other hand, exactly because of this flexibility, codes are harder to study in the semigroup-theoretic framework. The examples in the next chapter make this tradeoff more concrete.

Chapter 4

Protecting Semigroup Computations: Some Examples

4.1 Introduction

The analysis in Chapter 3 produced a framework that can be used for constructing arithmetic codes for computations with an underlying semigroup structure. Here, we use the results and methods developed there to construct and analyze arithmetic codes for some typical semigroups, such as the set of non-negative integers under addition $(\mathbf{N}_0, +)$, the set of positive integers under multiplication $(\mathbf{N}, \times)$, and the set of integers under the MAXIMUM operation $(\mathbb{Z}, \text{MAX})$. The objective in studying these simple semigroups is to gain more insight on arithmetic codes for semigroup operations.

This chapter is organized as follows. In Section 4.2 we make extensive use of the results obtained for separate codes in Chapter 3 and give a complete characterization of all possible separate codes for the $(\mathbf{N}_0, +)$ monoid. We also present an extensive list of examples for $(\mathbf{N}, \times)$, as well as a brief (but complete) description of separate codes for the $(\mathbb{Z}, \text{MAX})$ semigroup. In Section 4.3 we discuss possibilities for non-separate codes for various semigroups. Finally, Section 4.4 summarizes the conclusions that we have reached from the study of these examples.

4.2 Examples of Separate Codes

In this section, we develop separate codes for $(\mathbf{N}_0, +)$, $(\mathbf{N}, \times)$, and $(\mathbb{Z}, \text{MAX})$. The analysis for $(\mathbf{N}_0, +)$ and $(\mathbb{Z}, \text{MAX})$ will be complete in the sense that we get a complete characterization of all possible separate code for these semigroups. The analysis for $(\mathbf{N}, \times)$ results in a variety of examples and possibilities for separate codes.

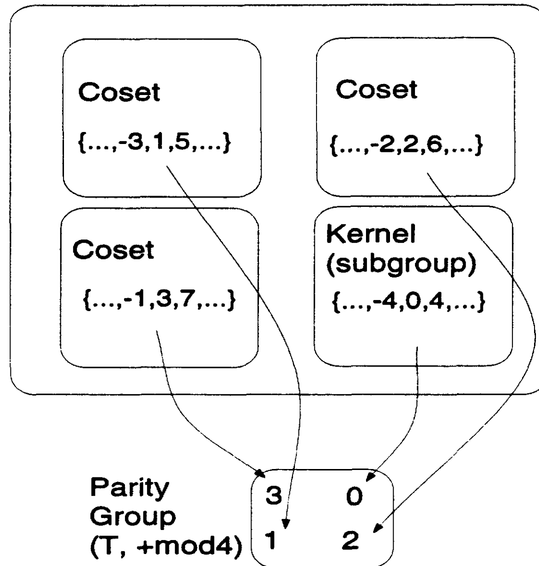


Figure 4-1: Example of a parity check code for the group ($\mathbb{Z}$, $+$).

4.2.1 Separate Codes for (N_0, $+$)

The set of natural integers under the operation of addition forms a very simple abelian semigroup. This semigroup is *cyclic*: it is generated by a single element[1], namely $\{1\}$. In fact, since all other infinite cyclic semigroups have to be isomorphic to it, (N, $+$) is the *unique* infinite cyclic semigroup. If we choose to insert the identity element 0, we have the (N_0, $+$) monoid. In fact, (N_0, $+$) is what we will be dealing with for the rest of this section.

Before focusing on the (N_0, $+$) monoid, let us revisit the corresponding group case: the set of integers (positive and negative) under the operation of addition forms a cyclic abelian group. This group is usually denoted by ($\mathbb{Z}$, $+$). Since all other infinite cyclic groups are isomorphic to it, ($\mathbb{Z}$, $+$) is the *unique* infinite cyclic group [22]. Moreover, as shown in [1] and [22], all possible surjectively homomorphic images of ($\mathbb{Z}$, $+$) (other than itself) are finite cyclic groups. A finite cyclic group of order m is isomorphic to the additive group of integers mod m. Therefore, using the results obtained in [1] and summarized in Chapter 2 of this thesis, we can achieve a complete characterization of the separate codes for the ($\mathbb{Z}$, $+$) group. The parity group (denoted by T) is isomorphic to the group of integers mod m, where m is an arbitrary integer. Therefore, the only possible parity channel operations for protecting addition of integers is addition modulo some integer m. This result was also obtained by Peterson [17] under a different framework.

Figure 4-1 presents an example of a parity check for ($\mathbb{Z}$, $+$). (Equivalently, we

[1]An element a is a *generator* of a cyclic semigroup (S, $\circ$) if $S = \{a \circ a \circ a \circ ... \circ a \ (p \ \textit{times}) \ | \ p \in \{1, 2, 3, ...\}\}$.

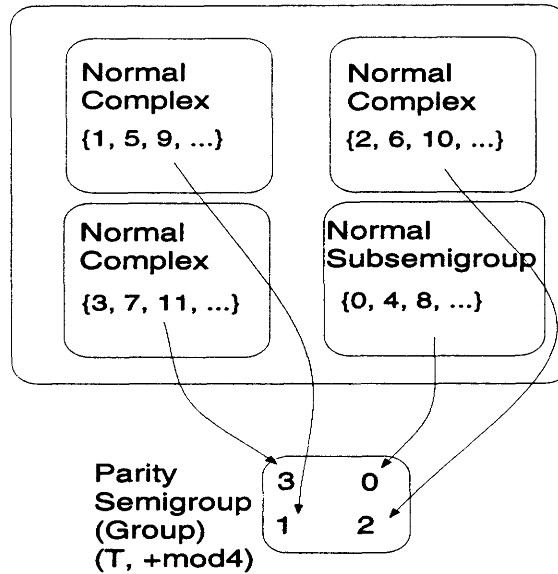


Figure 4-2: Example of a parity check code for the semigroup $(\mathbf{N}_0,\ +)$.

can think of it as an example of a surjective homomorphism for $(\mathbb{Z},\ +)$.) The parity group T is simply the set $\{0,1,2,3\}$ under the operation of addition modulo 4, which is denoted by $+_{\text{mod4}}$. In the figure, we can see which elements of the original group $\mathbb{Z}$ map to which elements of the parity group T. As analyzed in Chapter 2, the (normal[2]) subgroup of multiples of 4, $\{4k \mid k \in \mathbb{Z}\}$, maps to the identity of the parity group T, whereas the three cosets that remain map to the remaining elements of T in a regular way. In terms of the fault-tolerant model, the parity channel simply performs addition mod 4.

Clearly, addition mod m could also serve as a parity check in the case of the monoid $(\mathbf{N}_0,\ +)$. Such an example (for the case when $m = 4$) can be seen in Figure 4-2. The (normal) subsemigroup is the set of positive multiples of 4: $\{4k \mid k \in \mathbf{N}_0\}$, whereas the (normal) complexes consist of "shifted" versions of the subsemigroup: $\{i + 4k \mid k \in \mathbf{N}_0\}$ for $i = 1,2,3$. Clearly, this situation is very similar to the corresponding group case we have just discussed. The inverses (negative numbers) are missing; however, the parity semigroup T is still the exact same finite cyclic group of order 4.

Whereas in the $(\mathbb{Z},\ +)$ case the above parity checks were the only ones possible, for the semigroup $(\mathbf{N}_0,\ +)$ other kinds of surjective homomorphisms exist. Because a semigroup is not as structured as a group, parity checks that protect addition in $(\mathbf{N}_0,\ +)$ and differ from the parity checks for $(\mathbb{Z},\ +)$ can be constructed.

An example of such a parity check can be seen in Figure 4-3. The parity semigroup $(T,\ \diamond)$ is *not* a group anymore. It is simply a finite semigroup of order 8. The

[2]In the abelian case we are dealing with, the term "normal" is redundant.

table that defines how the binary operation $\diamond$ of T takes place under all possible pairs of operands is given in the figure as well[3]. It can be easily checked that the defining table implies the operation $\diamond$ is associative and abelian, so that T is indeed an abelian semigroup. The corresponding surjective homomorphism breaks $(\mathbf{N}_0, +)$ into eight normal complexes. Four of them (including the normal subsemigroup) consist of a single element. The other four consist of an infinite number of elements each.

In terms of the fault-tolerant model, the operation of the parity channel can be explained very simply: if the result of the addition is less than 4, then the parity channel duplicates the computation of the main channel; otherwise it performs addition $\bmod 4$ (just as in the previous examples we have seen). Duplicating the computation if the result is less than 4 was a choice made to simplify the example that we presented. In exactly the same way, we could have duplication of the computation if the result is less than 8, or 12, or any multiple of 4. Of course, that would correspond to a parity semigroup T of higher order.

If we try to apply a parity check like the above for addition in the group $(\mathbb{Z}, +)$ we fail. The problem is that, once we have exceeded the threshold and we have started performing addition $\bmod m$ (where m is the integer that we chose), we do not have enough information to return back to duplicating the result of the computation. However, the existence of the inverses (negative numbers) makes it possible for a computation that starts with the operands lying outside the threshold to result in a value less than the threshold. In such a case, we have to duplicate the result in the parity channel, but we do not have enough information to do so. For example, suppose that we decide to have a threshold of ± 12 and that once the result exceeds this limit we perform addition mod 4. Then $20 + (-16) = 4$ would be represent by something like $(0_{\text{mod4}} + 0_{\text{mod4}})$. Since interaction between the parity and the main channel is not allowed, there is no way for the parity channel to know whether the result of this computation should be 0, or 4, or 8, or 0_{mod4}.

Enumerating all Separate Codes for $(\mathbf{N}_0, +)$

Here, we show that all possible parity checks for the $(\mathbf{N}_0, +)$ semigroup are in either of the two forms (Figures 4-2 and 4-3) that were mentioned above. (The proof is rather long and is included in Appendix A.)

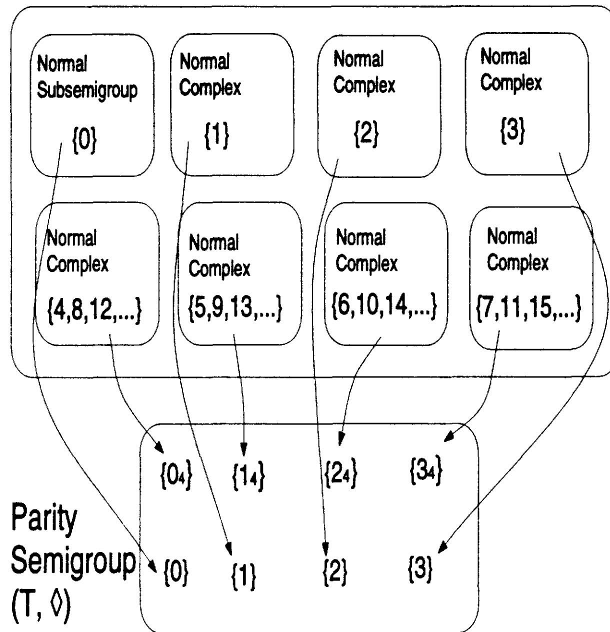
Specifically, we make the following equivalent claim:

Claim: All possible surjective homomorphisms θ from the semigroup $(\mathbf{N}_0, +)$ onto a semigroup T have one of the following two forms:

1. For $n \in \mathbf{N}_0$, $\theta(n) = n \bmod M$, where M is any *finite* integer (if M is infinite, then θ can be thought of as an isomorphism). This kind of homomorphisms map $(\mathbf{N}_0, +)$ onto a finite cyclic *group* of order M[4].

[3] A table is the standard method to define finite semigroups. Note that this defining table cannot be arbitrary: not only has it to imply closedness, but it also has to satisfy the associativity of the binary operation of the semigroup. Moreover, if the semigroup is abelian, the defining table should be symmetric along the diagonal that runs from the top left to the bottom right.

[4] In the special case where $M = 0$, we get a trivial homomorphism from $(\mathbf{N}_0, +)$ onto the trivial



$\diamond$	0	1	2	3	0_4	1_4	2_4	3_4
0	0	1	2	3	0_4	1_4	2_4	3_4
1	1	2	3	0_4	1_4	2_4	3_4	0_4
2	2	3	0_4	1_4	2_4	3_4	0_4	1_4
3	3	0_4	1_4	2_4	3_4	0_4	1_4	2_4
0_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4
1_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4
2_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4
3_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4

Figure 4-3: Example of a parity check code for the semigroup $(\mathrm{N}_0, +)$.

2. For a finite integer M, for $n \in \mathbf{N}_0$ we have:
$\theta(n) = n$ if $n < kM$ (for a fixed positive integer k)
$\theta(n) = n \bmod M$, otherwise
This kind of homomorphisms map $(\mathbf{N}_0, +)$ to a finite *semigroup* of order $M + kM = (k+1)M$[5].

The proof in Appendix A uses extensively the tools that were developed in Chapter 3 and applies them to the special case of this simple cyclic semigroup. We now move to a discussion of the more complicated example of $(\mathbf{N}, \times)$, the set of positive integers under multiplication.

4.2.2 Separate Codes for $(\mathbf{N}, \times)$

The monoid $(\mathbf{N}, \times)$ is far more complicated than $(\mathbf{N}_0, +)$. It has an infinite number of generators, namely the set of prime numbers. As a result, a lot more complexity is involved when attempting to characterize the set of parity check codes for it.

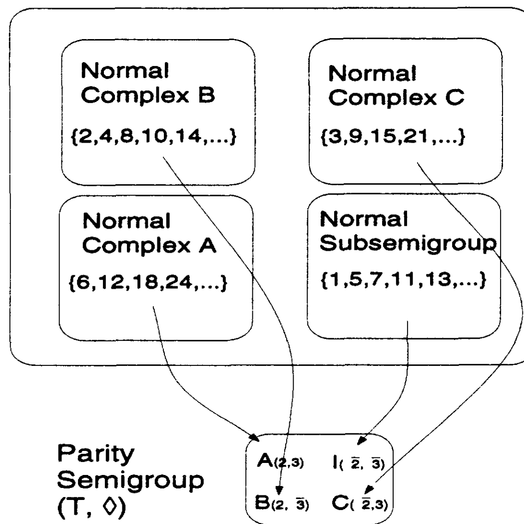
Let us begin our analysis with a simple example of a separate code for $(\mathbf{N}, \times)$. In Figure 4-4, we present a naive parity check. The parity semigroup T is a finite semigroup of order 4 and it is defined by the table in the same figure. We can easily verify that the operation $\diamond$, as defined in the table, is associative and abelian, so that T is indeed an abelian semigroup. The parity check essentially amounts to checking whether the result is a multiple of 2 or of 3 or of neither. The corresponding complexes (or, equivalently, the corresponding congruence classes) of $(\mathbf{N}, \times)$ are also shown in Figure 4-4. Complex A contains multiples of 2 *and* 3 (that is, multiples of 6). Complex B contains multiples of 2 but *not* multiples of 3. Complex C contains multiples of 3 but *not* multiples of 2. The (normal) subsemigroup, denoted by I, contains numbers that are neither multiples of 2 nor multiples of 3. It is interesting to note here that in the parity semigroup T, the element I is the identity element and A is the zero element[6]. Similar examples that check for more than two factors (not necessarily co-prime) can easily be constructed.

Another example, perhaps more familiar to the reader, is the parity check mod m where m is a positive integer. In Figure 4-5, we see such a case for $m = 4$. The parity semigroup T is finite (order 4) and its binary operation $\times_{\text{mod}4}$ is defined by the table in the figure. Note that the elements $0, 1, 2, 3$ of the parity semigroup should be treated as symbols rather than integers. However, this notation is useful because the elements of T under the binary operation $\times_{\text{mod}4}$ behave like integers under multiplication modulo 4. Although the parity semigroup T looks very similar to the one that was used in the previous example, it is in fact different. A parity

semigroup of a single identity element.

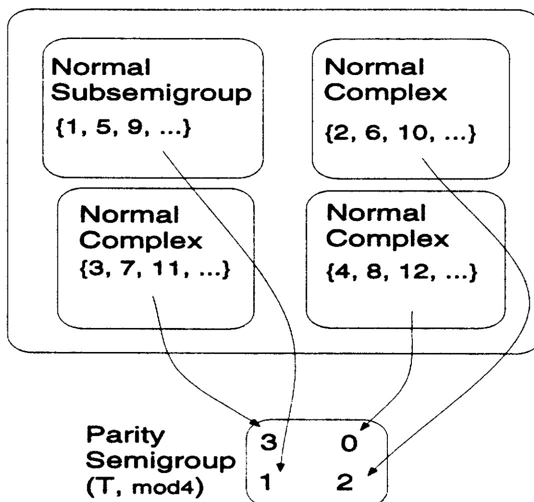
[5]When $M = 0$ the mapping reduces to:
$\theta(n) = n$ if $n < k$ (for a fixed positive integer k)
$\theta(n) = 0_0$, otherwise
where 0_0 is an element in the homomorphic image of $(\mathbf{N}_0, +)$ that "almost" behaves as the zero element.

[6]A zero element is the *unique* element z of S (if it exists) such that for all $s \in S$, $s \circ z = z \circ s = z$.



$\diamond$	I	A	B	C
I	I	A	B	C
A	A	A	A	A
B	B	A	B	A
C	C	A	A	C

Figure 4-4: Example of a parity check code for the semigroup $(\mathbf{N}, \times)$.



$\times_{\text{mod4}}$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Figure 4-5: Example of a parity check code for the semigroup (**N**, $\times$).

check such as the above can be constructed for any integer m. In such a case, the parity channel essentially performs multiplication mod m. In the special case when m is a prime number p, the parity semigroup T has a very special structure: if we discard the zero element, $(T - \{0\},\ \times)$ is a group[7].

Another interesting example is the following. Let P denote the set of prime numbers, that is, $P = \{2, 3, 5, 7, 11, 13, ...\}$. Then, (N, $\times$) can be partitioned into a set of infinite congruence classes as follows:

$$\begin{aligned}
C_0 &= \{1\} \\
C_1 &= P = \{2, 3, 5, ...\} \\
C_2 &= P^2 = \{p_1 \times p_2 \mid p_1, p_2 \in P\} \\
C_3 &= P^3 = \{p_1 \times p_2 \times p_3 \mid p_1, p_2, p_3 \in P\} \\
&\ \ \vdots \\
C_i &= P^i = \{p_1 \times p_2 \times ... \times p_i \mid p_1, p_2, ..., p_i \in P\}
\end{aligned}$$

[7] In fact, T under integer addition and multiplication mod p is the *unique* finite field of order m.

.

.

The corresponding parity semigroup T will be isomorphic to $(\mathbf{N}_0,\ +)$. The congruence class C_0 maps to 0, C_1 maps to 1, and in general $C_i \stackrel{\theta}{\mapsto} i$. The mapping θ is clearly a homomorphism:

$$\theta(C_i \times C_j) = \theta(C_{(i+j)}) = i + j = \theta(C_i) + \theta(C_j)$$

Essentially, the above parity computation checks the total number of primes that are used in producing a result (including multiple uses of the same prime).

More examples are possible, including cases when the parity channel performs the exact same computation if the result is less than a threshold (just like the examples we presented in the previous section for the $(\mathbf{N}_0,\ +)$ monoid[8]). However, it is not our concern here to enumerate all possible separate codes for $(\mathbf{N},\ \times)$ (although that was easy for $(\mathbf{N}_0,\ +)$). The purpose was rather to demonstrate the use of the framework developed in Chapter 3 for generating separate codes for $(\mathbf{N},\ \times)$.

4.2.3 Separate Codes for $(\mathbb{Z} \cup \{-\infty\},\ \text{MAX})$

We conclude our examples of separate codes with a brief analysis of codes for $(\mathbb{Z} \cup \{-\infty\},\ \text{MAX})$, the semigroup of integers under the MAX operation. MAX is the binary operation that returns the largest of its operands.

We will characterize the various forms that a congruence class in $(\mathbb{Z} \cup \{-\infty\},\ \text{MAX})$ can take, by following similar steps as in the analysis of the separate codes for $(\mathbf{N}_0,\ +)$. For ease of notation, we will use the familiar notion of inequalities:

$$x \geq y \Leftrightarrow \text{MAX}(x, y) = x$$

From the definition of a congruence class, we can easily conclude the following: If two elements k, k' (without loss of generality assume $k \geq k'$) belong to a congruence class C, then the set $\{x \mid k \geq x \geq k'\}$ is a subset of C. Therefore, we conclude that any congruence class consists of consecutive integers in an interval. This immediately yields a complete characterization of the separate codes for $(\mathbb{Z} \cup \{-\infty\},\ \text{MAX})$. The (normal) subsemigroup is, of course, the congruence class that includes the identity (which in this case is the symbol $-\infty$).

A simple example would be the following pair of congruence classes:

$$\begin{aligned} C_0 &= \{..., -2, -1\} \cup \{-\infty\} \\ C_1 &= \{0, 1, 2, ...\} \end{aligned}$$

The corresponding parity semigroup T is of order 2 and its operation $\diamond$ can be defined in the following table:

[8]For example, this can be achieved in exactly the same way for the parity check mod m. See the analysis for the $(\mathbf{N}_0, +, \times)$ semiring in Chapter 5 for a concrete example.

$\diamond$	0	1
0	0	1
1	1	1

Intuitively, all that the parity channel is doing is checking that the sign of the result comes out correctly.

Other examples do exist. In fact, any subdivision of the $[-\infty, +\infty]$ interval into consecutive subintervals will work. For example, consider the following set of congruence classes for some fixed positive integer M:

$$C_i = \{iM, iM+1, ..., iM+(M-1)\} \text{ for all } i \in \mathbb{Z}$$

In this case, the parity semigroup T is isomorphic to the $(\mathbb{Z} \cup \{-\infty\},$ MAX) semigroup. However, its "range" and its "precision" are smaller by a factor of M. For example, if the elements of $\mathbb{Z}$ are represented as binary numbers and $M = 16$, then this parity test ignores the 4 least significant bits and then performs the same comparison as the main channel.

Before we move to non-separate codes, let us note that, if we slightly modify the above analysis, we can arrive at similar results for the $(\mathbb{Z} \cup \{+\infty\},$ MIN), $(\mathbb{R} \cup \{-\infty\},$ MAX), and $(\mathbb{R} \cup \{+\infty\},$ MIN) semigroups.

4.3 Examples of Non-Separate Codes

Just as with the study of non-separate codes for group-based computations, the study of non-separate codes for semigroup-based computation is hard. There is no way of associating subgroups or congruence relations to them and, in general, the development of non-separate arithmetic codes for these computations is more of an art than a systematic procedure. Of course, the objective would be to homomorphically map the original group/semigroup to a larger group/semigroup, but this is not a trivial task.

For example, the only non-separate codes that are well known for the $(\mathbb{Z}, +)$ group are the αN codes which were mentioned in Chapter 1 (see Figure 1-3). These codes definitely comprise a class of non-separate codes that can be used for protecting $(\mathbb{N}_0, +)$, although a natural extension of them for the semigroup case is not clear. A brief analysis of αN codes can be found in [1] and a more extensive treatment can be found in [16]. In that case, they are used as arithmetic codes to protect the $(\mathbb{Z}, +)$ group but they can certainly be used in the same way to protect the $(\mathbb{N}_0, +)$ monoid.

4.4 Summary

The main emphasis of this chapter was to demonstrate (through a variety of examples) the use of the semigroup-theoretic framework for the development of separate codes for semigroup-based computations. This demonstration focused mainly on simple semigroups like the semigroup of non-negative integers under addition or multiplication and the semigroup of integers under the MIN and MAX operations.

Having presented a variety of codes for these semigroups (sometimes completely characterizing all possible separate codes for them), we are now in position to extend the framework and the tools that we have to higher group- or semigroup-based structures. In Chapter 5, we develop two such frameworks and use them to analyze the arithmetic codes for various computations with an underlying *ring* or *semiring* structure[9]. Among others, we consider the ring of integers under addition and multiplication and the semiring of non-negative integers under addition and multiplication. The analysis will result in complete characterizations of all possible separate codes for both of these structures.

[9] Definitions are provided in Chapter 5.

Chapter 5

Frameworks for Higher Algebraic Structures

5.1 Introduction

Up to this point, we have dealt exclusively with algebraic structures that permit a single operation. The focus of the theoretical analysis in Chapters 2 and 3 and of the examples in Chapter 4 was on group or semigroup computations. In this chapter, we are concerned with higher algebraic structures that are derived from these simple ones. Specifically, we rigorously extend the group framework to a *ring* framework (as outlined in [1]) and extend the semigroup framework to a *semiring* framework. Much of the classical work on fault-tolerant computation is developed for the case of rings, but the connections to group computations have not been highlighted prior to [1]. Our systematic work on protecting semiring computations appears to be novel.

These extensions to higher structures allow us to protect more complex and more common computations that comprise two operations associated by some distributive laws. Moreover, different possibilities are opened up for the resolution of certain important issues. For example, the error model need not be purely additive anymore; it can have both an additive and a multiplicative component. By having more options for modeling the error, we should be able to better approximate the actual faults that take place in the computational system. This can lead to more efficient arithmetic codes and to error correction techniques that were not possible in the group/semigroup case.

In this chapter, we limit ourselves to setting the foundations of a ring/semiring framework for studying arithmetic codes. We formulate the problem mathematically and, through examples, we study the design of arithmetic codes. The application of this framework to the analysis of arithmetic codes, the design of efficient error correction techniques, and so on, is left to future work.

This chapter is organized as follows. In Section 5.2 we formalize and develop the ring-theoretic framework that was mentioned in [1]. Then, in Section 5.3, we provide a few examples that fit in this framework. The semigroup framework is extended to

a semiring framework[1] in Section 5.4. A few examples of separate codes for semirings are presented in Section 5.5. Finally, Section 5.6 summarizes the results and concludes the chapter.

5.2 Ring-Theoretic Framework

In this section we extend the results of Chapter 2 for the group case to a higher group-based structure known as a *ring*. Because our analysis closely follows the steps that we took in Chapter 2, we avoid detailed explanations when possible. We begin with the definition of a ring, and of the associated model for computations in a ring.

5.2.1 Computation in a Ring

The definition of a ring (taken from [23]) is as follows.

Definition: A **ring** $\mathcal{R} = (R, +, \times)$ is a non-vacuous set of elements R together with two binary operations $+$ and $\times$ (called *addition* and *multiplication* respectively), and two distinguished elements 0_+ and $1_\times$ such that:

- R forms an abelian group under the operation of addition. The identity element of the group is 0_+.
- R forms a monoid under the operation of multiplication. The identity element of the monoid is $1_\times$.
- For all $a, b, c \in R$, the following *distributive laws* are satisfied:
  1. $a \times (b + c) = (a \times b) + (a \times c)$, and
  2. $(b + c) \times a = (b \times a) + (c \times a)$.

The most familiar example of a ring is the set of integers under the operations of addition and multiplication (this ring is denoted by $(\mathbb{Z}, +, \times)$). Another familiar example is the ring $(\mathbb{R}[z], +, \times)$ of polynomials in the indeterminate z. Both of these rings are *abelian* rings, because the multiplicative operation is also an abelian operation. An example of a non-abelian ring, perhaps less familiar, is the set of $N \times N$ matrices under the operations of addition and multiplication (matrix multiplication is not an abelian operation).

The defining properties of a ring force the additive identity 0_+ to act like the multiplicative zero ([15]), that is, for all elements $r \in R$:

$$r \times 0_+ = 0_+ \times r = 0_+$$

This is a very important property of a ring. In fact, as we will see later on, this is what forces the kernel of a homomorphism to have the form of an *ideal*.

[1] A *semiring* is the natural extension of a ring when we relax the requirement that the set forms a group under the additive operation and, instead, only require it to form a monoid. A formal definition of the semiring structure is given in Section 5.4.

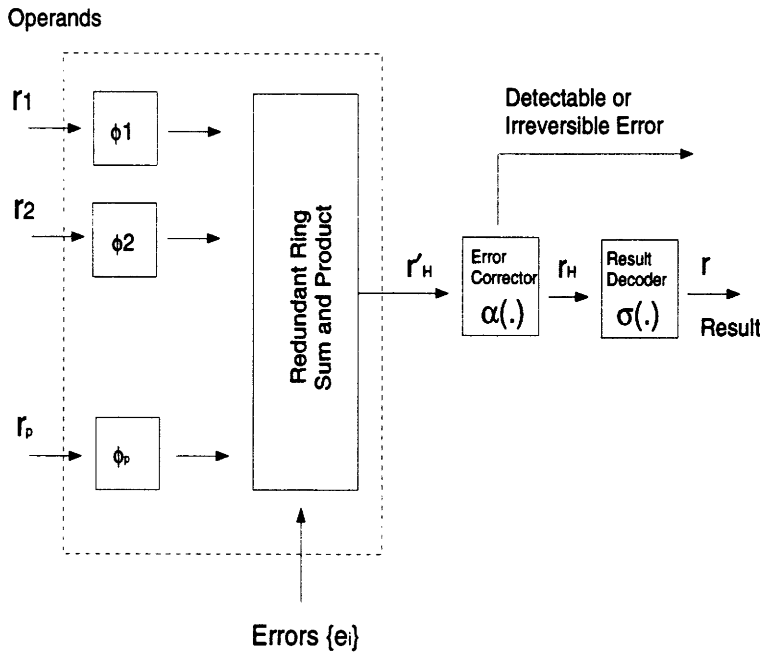


Figure 5-1: Fault-tolerant model for a ring computation.

Fault-Tolerant Model

The model for fault-tolerant ring computation can be seen in Figure 5-1. Just as with the previous models we have seen, the operands $r_1, r_2, ..., r_p$ are encoded via the encoders $\{\phi_i\}$. The computation takes place in the redundant computational unit that performs operations in a higher (redundant) ring. Errors $\{e_i\}$ are introduced during this redundant computation.

The rest of the fault-tolerant system remains exactly the same as the model for group computations of Chapter 2. The output of the redundant computation unit is a possibly corrupted result, which we denote by r'_H. The error corrector maps r'_H to the fault-free redundant result r_H. In the case of an error that is irreversible or only detectable, the error corrector signals so. The decoder maps (through the decoding mapping σ) the fault-free redundant result r_H to its desired, unencoded form, r. As usual, an implicit assumption in this figure is that the encoders, the error corrector and the result decoder are fault-free.

Error Model

Despite the fact that the choice of error model does not affect the limited analysis that we present in this thesis, a few remarks on this choice are in order. It should be clear by now that, in the case of a ring, the *additive* error model (defined with respect to the + operation) is not necessarily the best. A *multiplicative* error model (that is, an error model that is "additive" with respect to the $\times$ operation[2]), or perhaps a

[2]The use of a *multiplicative* error model is more sensible if the ring is abelian, that is, the multiplicative operation is abelian. If the ring is not abelian, then we should be more careful: we

combination of a multiplicative and an additive error model, could be a better choice. Of course, it all depends on the particular computation at hand, the actual hardware faults that can take place in our computational system, and how efficiently these faults can be represented by our error model.

Since the choice of the error model does not affect the analysis of arithmetic codes that we present in the rest of this section, we are content to leave further study of the error model to future work.

5.2.2 Use of Ring Homomorphisms

Let us now formulate the requirements that emerge from our model in terms of the original ring $(R, +, \times)$, the redundant ring $(H, \circ, \diamond)$, and the decoding mapping σ. The desired result r is found by decoding the uncorrupted result r_H of a fault-free redundant computation:

$$r = r_1 \odot r_2 \odot \dots \odot r_p = \sigma(r_H)$$

where each $\odot$ could independently be either the additive $(+)$ or the multiplicative $(\times)$ operation of the original ring R, and $r_H = \phi_1(r_1) \star \phi_2(r_2) \star \dots \star \phi_p(r_p)$ with $\star$ being either the additive $(\circ)$ or the multiplicative $(\diamond)$ operation of the redundant ring H.

We are now in position to state the following claim:

Claim: When both $(R, +, \times)$ and $(H, \circ, \diamond)$ are rings, under the assumptions that:

1. The mapping σ is one-to-one, and
2. $\phi_1(1_\times) = \phi_2(1_\times) = \dots = \phi_p(1_\times) = 1_\diamond$ (that is, all encodings map the multiplicative identity of R to the multiplicative identity of H),

all the ϕ_i's have to be the same ring homomorphism $\phi = \sigma^{-1}$.

Proof: By simply considering R as a group under the additive operation, we can show (Chapter 2) that all the ϕ_i have to be the same ($\phi_i(r) = \sigma^{-1}(r)$ for all $r \in R$) and satisfy:

$$\phi_i(r_1 + r_2) = \phi_i(r_1) \circ \phi_i(r_2) \text{ for all } r_1, r_2 \in R$$

Moreover, by simply considering R as a monoid under the multiplicative operation, we can show (Chapter 3) that for each encoding, $\phi_i(r) = \sigma^{-1}(r)$ for all $r \in R$, and also[3]:

$$\phi_i(r_1 \times r_2) = \phi_i(r_1) \diamond \phi_i(r_2) \text{ for all } r_1, r_2 \in R$$

The above two equations, together with the condition $\phi_i(1_\times) = 1_\diamond$ that we have assumed, are exactly the definition of a ring homomorphism. $\surd$

In arriving at this result, we only had to make the same set of basic assumptions that we made earlier in Chapters 2 and 3 when considering the simpler group and semigroup structures. The only extra choice we had was whether to associate the

need to define a *left* and a *right* multiplicative error and distinguish between the two.

[3]In Chapter 3 we assumed that we were dealing with an abelian monoid. However, the proof of the following relation does not require the monoid to be abelian.

additive operation of R with the multiplicative operation of H (instead of its additive operation). However, this could not guarantee that the mapping would be compatible with the associative laws that hold in the original ring R. In fact, the multiplicative operation in the redundant ring H is not even known to be abelian, whereas the additive operation of R is. Therefore, a way to overcome these problems is to associate the additive (respectively, multiplicative) operation of R with the additive (multiplicative) operation of H. Once we decide on this, the mappings $\{\phi_i\}$ are forced to be the exact same ring homomorphism ϕ.

By studying ring homomorphisms, we can develop and analyze arithmetic codes for computational tasks that can be modeled as operations in a ring. Moreover, just as in the group case, there is a systematic way of studying separate codes for ring computations. This is the theme of the next section.

5.2.3 Separate Codes for Rings

In the special case of separate codes, the ring homomorphism ϕ maps the original computation in a ring R to a redundant computation in a higher order ring H, where H is the cartesian product of R and a parity set (which is also a ring) that we will call T, that is:

$$R \stackrel{\phi}{\longmapsto} H = R \times T$$

The homomorphic mapping satisfies $\phi(r) = [r, \theta(r)]$, where θ is the mapping that creates the parity information from the operands. The set of valid results H_V in H (that is, the set of results obtained under fault-free computation) is the set of elements of the form $[r, \theta(r)]$, for all $r \in R$.

Let us now show that θ is a ring homomorphism from R to T. Let the symbols $+, \times$ denote the additive and multiplicative operations of R, the symbols $\circ, \diamond$ denote the corresponding operations of H, and the symbols $\oplus, \otimes$ denote the corresponding operations of the parity ring T. Also let:

$$\begin{aligned} h_1 &= \phi(r_1) = [r_1, \theta(r_1)] \\ h_2 &= \phi(r_2) = [r_2, \theta(r_2)] \end{aligned}$$

where $h_1, h_2 \in H$, $r_1, r_2 \in R$, and $\theta(r_1), \theta(r_2) \in T$. Since the computations of the main and parity channels do not interact, we require that under error-free operation:

$$\begin{aligned} h_1 \circ h_2 = \phi(r_1) \circ \phi(r_2) &= [r_1 + r_2, \theta(r_1) \oplus \theta(r_2)] \\ h_1 \diamond h_2 = \phi(r_1) \diamond \phi(r_2) &= [r_1 \times r_2, \theta(r_1) \otimes \theta(r_2)] \end{aligned}$$

Since the mapping ϕ is a homomorphism, we also have:

$$\begin{aligned} \phi(r_1) \circ \phi(r_2) &= \phi(r_1 + r_2) = [r_1 + r_2, \theta(r_1 + r_2)] \\ \phi(r_1) \diamond \phi(r_2) &= \phi(r_1 \times r_2) = [r_1 \times r_2, \theta(r_1 \times r_2)] \end{aligned}$$

We conclude that

$$\begin{aligned}\theta(r_1)\oplus\theta(r_2) &= \theta(r_1+r_2)\\ \theta(r_1)\otimes\theta(r_2) &= \theta(r_1\times r_2)\end{aligned}$$

The above properties, together with $\theta(1_\times) = 1_\otimes$ (a fact that can be shown easily), establish that the mapping θ is a ring homomorphism from R to T.

If we require that θ is onto T, then the problem of finding suitable separate codes reduces to the problem of finding suitable *surjective homomorphisms* θ from R onto T. Here is where a standard theorem from ring theory (see for example [15]) almost solves our problem completely. Before we state it, however, we need to define what an *ideal* is.

Definition: A nonempty subset U of a ring $(R, +, \times)$ is said to be an **ideal** of R if[4]:

1. U is a subgroup of R under the additive operation.

2. For every $u \in U$ and $r \in R$, both $u\times r$ and $r\times u$ are in U.

Theorem: Let R, T be rings and θ a homomorphism of R onto T with kernel U. Note that U is an ideal of R. Then T is isomorphic to R/U. Conversely, if U is an ideal of R, then R/U is a ring and it is a surjectively homomorphic image of R.

The above theorem states that there is an isomorphism between surjective homomorphisms θ of the ring R onto T, and ideals U of R: the quotient ring R/U provides an isomorphic image of T. Therefore, by finding all possible ideals of R, we can find all possible surjective homomorphisms θ of R onto another ring T. Note that once we decide upon the ideal U, then the construction of the quotient ring R/U only uses the additive group structure of the ring: R/U is formed by the standard coset construction for groups. The ring structure is such that the multiplicative operation only needs to be considered when finding an ideal for the ring R. After that, we only need to worry about the additive operation.

The problem of finding all possible separate codes for a computation with an underlying ring structure has been reduced to the well formulated algebraic problem of finding all possible ideals of a ring. Of course, finding the ideals of a ring is not a trivial task, but now we at least have a well-defined procedure that can be used to construct arithmetic codes. Moreover, it results in a complete characterization of the possible separate codes for computation in a ring.

5.3 Examples in the Ring-Theoretic Framework

In this section we present some examples (taken from [1]) of already existing arithmetic coding schemes in the ring framework. We start with examples of non-separate codes and then move to examples of separate ones.

[4]Some authors use the term "two-sided ideal" for what we define here as an ideal.

5.3.1 Examples of Non-Separate Codes

The dominant example of a non-separate arithmetic code comes from the ring of $N \times N$ matrices. The example was presented in detail in Chapter 1 as an instance of Algorithm-Based Fault Tolerance (ABFT). The set of $N \times N$ matrices forms a group under the operation of matrix addition. If we include matrix multiplication, it forms a non-abelian ring (known as the *ring of matrices*). Assuming an additive error model (with respect to the additive group operation), errors take place as:

$$R' = R + E$$

where R' is the possibly corrupted result, R is the error-free result, and E is a matrix that represents the additive error.

Under these assumptions, a homomorphism ϕ that maps the matrix ring to a larger redundant ring will correspond to an arithmetic code. Therefore ϕ needs to satisfy the following requirements:

$$\begin{aligned} \phi(A+B) &= \phi(A) \circ \phi(B) \\ \phi(A \times B) &= \phi(A) \diamond \phi(B) \\ \phi(I_\times) &= I_\diamond \end{aligned}$$

where the symbols $\circ$ and $\diamond$ have been used to denote the operations that take place in the homomorphic ring and are not necessarily the same as the operations of the original ring. The symbol $I_\times$ represents the multiplicative identity (identity matrix) in the original ring, whereas $I_\diamond$ represents the multiplicative identity in the redundant ring.

In the ABFT example that was presented in Chapter 1, the homomorphic mapping ϕ maps the $N \times N$ matrix to a larger $(N+1) \times (N+1)$ matrix by adding to it an extra checksum row and an extra checksum column[5]. In order to satisfy the homomorphism equations above, we need to ensure that we perform the $(N+1) \times (N+1)$ matrix product $\phi(A) \diamond \phi(B)$ in the following way: we ignore the checksum column of A and the checksum row of B and then perform regular matrix multiplication of an $(N+1) \times N$ matrix with an $N \times (N+1)$ matrix. It can be easily verified that the result will be an $(N+1) \times (N+1)$ matrix that satisfies the requirements for the mapping to be homomorphic. The operation of addition on the homomorphic images $\circ$ remains the same as regular matrix addition. The last requirement (that the multiplicative identity $I_\times$ maps to $I_\diamond$) is also satisfied by the mapping ϕ and the multiplicative operation $\diamond$ as defined above.

This scheme was introduced in [3] and provided sufficient error correction as long as the error matrix E had only a single non-zero entry. When E has more than one non-zero entry, we need more redundancy. ABFT schemes for exactly this purpose were developed later in [4]. They were basically a natural extension of the method described above.

[5]The element in the lower right corner of the matrix turns out to simply be the sum of all the elements of the original $N \times N$ matrix.

More examples of codes in the ring framework can be found in [1]. Such codes include linear transformations, codes for finite fields[6] and others.

5.3.2 Examples of Separate Codes

Residue codes are separate codes that can be used to protect computation in $(\mathbb{Z}, +, \times)$, the ring of integers under addition and multiplication.

The ideals of $(\mathbb{Z}, +, \times)$ are known to be of the following form:

$$U = \{0, \pm M, \pm 2M, \pm 3M, ...\}$$

for a non-negative integer M. If $M = 0$ then $U = \{0\}$ and this corresponds to the surjective homomorphism θ being an *isomorphism* that maps $(\mathbb{Z}, +, \times)$ onto itself, i.e. the parity information is just a repetition of the element being coded. If $M = 1$, then $U = \mathbb{Z}$ and θ corresponds to the trivial homomorphism that maps $\mathbb{Z}$ onto the trivial ring of a single element[7].

However, for all other $M > 1$ we get a non-trivial separate code. Given our results in Chapter 4 on separate codes for $(\mathbb{Z}, +)$, it is not hard to convince oneself that when $M > 1$, the code corresponds to a parity channel that performs addition and multiplication modulo M. Therefore, we have arrived at the interesting conclusion that the only possible non-trivial separate codes for protecting computation in the ring of integers are residue codes that perform operations modulo an integer M. In fact, this same result was obtained by Peterson [17].

More examples of separate codes for ring-based computations can be found in [1]. They include examples in the ring of polynomials, linear transformations, real residue codes, and others.

5.4 Semiring-Theoretic Framework

5.4.1 Computation in a Semiring

In this section we are interested in studying higher semigroup-based structures. More specifically, we study fault-tolerant computation in a *semiring*, a simple algebraic structure that admits two operations. We begin with the following definition of a semiring, which is a slightly altered version of the definition found in [24]:

Definition: A **semiring** $\mathcal{R} = (R, +, \times)$ is a non-vacuous set of elements R together with two binary operations $+$ and $\times$ (called *addition* and *multiplication* respectively), and two distinguished elements 0_+ and $1_\times$ such that:

- R forms an abelian monoid under the operation of addition. The identity element of the additive monoid is 0_+.

[6] A *field* is a special case of a ring where the multiplicative operation is abelian and the set of non-zero elements forms a group under the multiplicative operation.

[7] Technically speaking, our definition of a ring does not allow for this trivial ring.

- R forms a monoid under the operation of multiplication. The identity element of the multiplicative monoid is $1_\times$.
- All $a, b, c \in R$ satisfy the *distributive laws*:
  1. $a \times (b + c) = (a \times b) + (a \times c)$, and
  2. $(b + c) \times a = (b \times a) + (c \times a)$.

Clearly, every ring is a semiring. The most natural example of a semiring that is not a ring is the set of *non-negative* integers under integer addition (additive operation) and multiplication (multiplicative operation). This semiring is usually denoted by $(\mathrm{N}_0, +, \times)$. One can easily check that the conditions of a semiring are satisfied: integer addition is abelian and associative, integer multiplication is associative, and multiplication distributes over addition. In fact, $(\mathrm{N}_0, +, \times)$ will be the focus of our examples in the next section.

Other examples of semirings are $(\mathbb{Z} \cup \{-\infty\}, \mathrm{MAX}, +)$, the set of integers under the operations MAX (additive operation) and $+$ (multiplicative operation)[8], and $(\mathbb{R} \cup \{-\infty\}, \mathrm{MAX}, +)$ (the set of real numbers under the same operations). Clearly, we can replace the MAX operation with MIN (and include the symbol $+\infty$ instead of $-\infty$) and still get a semiring. Another interesting example of a semiring is $(\mathbb{Z} \cup \{\pm\infty\}, \mathrm{MAX}, \mathrm{MIN})$ that is, the set of integers under the MAX and MIN operations. In fact, $(\mathbb{Z} \cup \{\pm\infty\}, \mathrm{MIN}, \mathrm{MAX})$, $(\mathbb{R} \cup \{\pm\infty\}, \mathrm{MAX}, \mathrm{MIN})$, $(\mathbb{R} \cup \{\pm\infty\}, \mathrm{MIN}, \mathrm{MAX})$ are also examples of semirings. The reader can verify that these structures satisfy the requirements of our semiring definition.

Before we present the general framework for protecting computations with an underlying semiring structure, we make a comment on the definition of a semiring. Some authors assume that a semiring is a bit more structured, requiring that in a semiring $(R, +, \times)$ the additive cancellation law is satisfied, that is:

$$\text{For all } a, b, c \in R, \text{ if } a + c = b + c \text{ then } a = b$$

Interestingly enough, this assumption forces 0_+, the additive identity of R, to behave as a multiplicative zero[9]:

$$\text{For any element } r \in R\text{: } r \times 0_+ = 0_+ \times r = 0_+$$

Some other authors (including [24] and [25]) simply assume that a semiring satisfies the equation above (and not necessarily the additive cancellation law). As we will see in our analysis, this assumption simplifies our task slightly. However, we keep the definition and the analysis of a semiring as general as possible. We begin the analysis of the fault-tolerant model in the next section.

[8] This is an interesting example because integer addition behaves like a multiplicative operation.
[9] The proof makes use of the distributive and additive cancellation laws. We leave it to the reader.

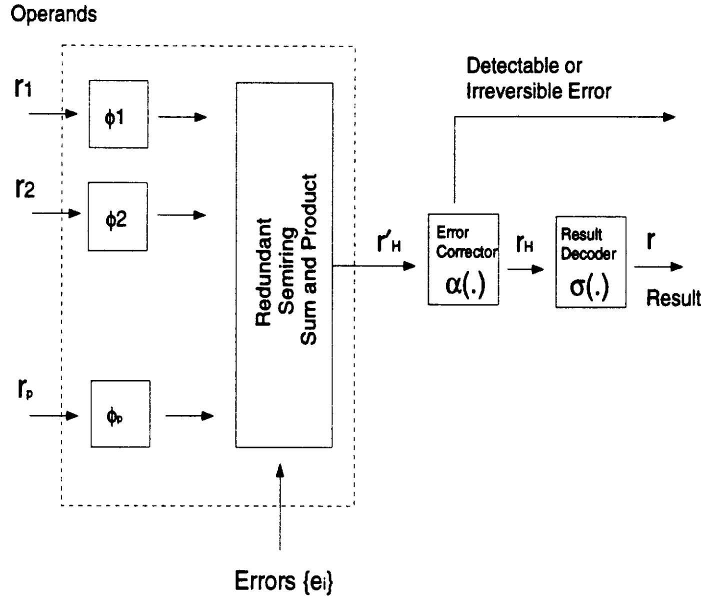


Figure 5-2: Fault-tolerant model for a semiring computation.

Fault-Tolerant Model

In order to protect a computation with an underlying semiring structure, we use the computational model shown in Figure 5-2. We protect a computation in a semiring R by mapping it to a larger (redundant) semiring H. The operands $r_1, r_2, ..., r_p$ are encoded via the encoders $\{\phi_i\}$ and the redundant computation takes place in semiring H. Errors $\{e_i\}$ might be introduced during this redundant computation. Once again, we assume that the encoders, error corrector and decoder are error-free.

The rest of the fault-tolerant system remains exactly the same as in the model for semigroup computations in Chapter 3. Let us denote the possibly corrupted output of the redundant computational unit by r'_H; then the error corrector tries to map r'_H to the fault-free redundant result r_H. In the case of an irreversible or a detectable error, the error corrector signals so. The decoder, as usual, performs the decoding of the fault-free redundant result to its original form, which we denote by r.

5.4.2 Use of Semiring Homomorphisms

Before we start our analysis, we make a note on semiring homomorphisms: a *semiring homomorphism* from $(R, +, \times)$ to $(H, \circ, \diamond)$ is, by definition, a mapping ϕ that satisfies the following rules:

- For all $a, b \in R$:

$$\begin{aligned} \phi(a+b) &= \phi(a) \circ \phi(b) \\ \phi(a \times b) &= \phi(a) \diamond \phi(b) \end{aligned}$$

- The mapping should also map the additive and multiplicative identities of R to the additive and multiplicative identities of H respectively[10]:

$$\phi(0_+) = 0_\circ$$
$$\phi(1_\times) = 1_\diamond$$

By following the steps of Section 5.2, we can easily arrive at the following conclusion:

Claim: Suppose $(R, +, \times)$ and $(H, \circ, \diamond)$ are both semirings, with identities $0_+, 1_\times$ and $0_\circ, 1_\diamond$ respectively. Then, under the assumptions that:

1. The mapping σ is one-to-one,

2. $\phi_1(0_+) = \phi_2(0_+) = ... = \phi_p(0_+) = 0_\circ$ (that is, all encodings map the additive identity of R to the additive identity of H), and

3. $\phi_1(1_\times) = \phi_2(1_\times) = ... = \phi_p(1_\times) = 1_\diamond$ (all encodings map the multiplicative identity of R to the multiplicative identity of H),

then all ϕ_i's have to be equal and have to be a semiring homomorphism $\phi = \sigma^{-1}$.

Proof: The proof is virtually identical to the proof in Section 5.2 and is left to the reader. $\surd$

At this point we have established that the study of arithmetic codes for a given semiring computation is equivalent to the study of semiring homomorphisms. In what follows, we use this result to study separate codes for semirings.

5.4.3 Separate Codes for Semirings

In the ring case that we analyzed in Section 5.2, there was enough structure so that a simple specification of the kernel of a ring homomorphism (which really corresponds to what was defined earlier as an ideal) was able to determine completely the whole homomorphism. In the semiring case, we are not as fortunate. As we will see, the analysis breaks down at two points:

- The kernel of a semiring homomorphism, being a normal subsemigroup under the additive operation, is not even strong enough to completely specify the normal complexes of the semiring under the additive operation.

- Even if we are able to specify all possible ways of partitioning the semiring into normal complexes under the additive operation (given a kernel for a homomorphism), it is not guaranteed that all of these partitions will comply with the homomorphic requirement on the multiplicative operation.

[10]If the semiring H satisfies the additive cancellation law, we can easily show that the mapping *has to map* the additive identity of R to the additive identity of H. Therefore, in this case, the first of the two statements is not necessary.

The theoretical analysis that follows generalizes the notion of a semigroup congruence relation/class to a *semiring congruence relation/class*, and the notion of a normal complex/subsemigroup to the notion of a *semiring ideal/complex.* The attempt is to provide definitions, tools and methods that enable us to study semiring homomorphisms.

Parity Encoding for Separate Codes

The model for fault-tolerant computation using separate arithmetic codes remains exactly the same as the model for a computation that takes place in a ring. By following the steps we followed in Section 5.2, we can show that the encoding mapping θ has to be a semiring homomorphism from $(R, +, \times)$ to $(T, \oplus, \otimes)$, because it satisfies:

- For all $a, b \in R$:

$$\begin{aligned} \theta(a+b) &= \theta(a) \oplus \theta(b) \\ \theta(a \times b) &= \theta(a) \otimes \theta(b) \end{aligned}$$

- Also:

$$\begin{aligned} \theta(0_+) &= 0_\oplus \\ \theta(1_\times) &= 1_\otimes \end{aligned}$$

Determination of Possible Homomorphisms

In this section, just as we did in the previous cases, we assume that the semiring homomorphism is surjective, and outline a systematic approach for enumerating all such homomorphisms for a given semiring R. In order to achieve that, we look for *semiring congruence relations* and attempt to partition the original semiring into classes based on these relations:

Definition: An equivalence relation $\sim$ on the elements of a semiring $(R, +, \times)$ is called a **semiring congruence relation** if it is compatible with the two operations of the semiring in the following sense:

$$\text{For } a, b, a', b' \in R \text{ we have: If } a \sim a', b \sim b' \Rightarrow a+b \sim a'+b' \text{ and } a \times b \sim a' \times b'$$

Now, let the set $R/\sim$ denote the set of equivalence classes of the semiring R under the semiring congruence relation $\sim$ (we call these classes *semiring congruence classes*). For equivalence classes $[a], [b]$ ($[a]$ is the equivalence class that contains element a) we define the following binary operations:

$$\begin{aligned} [a] \oplus [b] &= [a+b] \\ [a] \otimes [b] &= [a \times b] \end{aligned}$$

Claim: Both of the above operations are well-defined and $R/\sim$ (the set of equivalence classes of R under the congruence relation $\sim$) is a semiring.

Proof: If $[a] = [a']$ and $[b] = [b']$ then $a \sim a'$ and $b \sim b'$. Therefore, $a + b \sim a' + b'$ and $a \times b \sim a' \times b'$, which means that $a + b$ belongs to the same semiring congruence class as $a' + b'$, and $a \times b$ belongs to the same semiring congruence class as $a' \times b'$. Therefore, operations $\oplus$ and $\otimes$ are well-defined. Operations $\oplus, \otimes$ are clearly associative (they inherit associativity from $+, \times$ respectively). Moreover, operation $\oplus$ is evidently abelian and the distributive laws for operations $\oplus$ and $\otimes$ are inherited from the distributive laws of R. The additive identity of $R/\sim$ is $[0_+]$ and its multiplicative identity is $[1_\times]$. Therefore, $(R/\sim, \oplus, \otimes)$ is a semiring. $\surd$

Now, we are ready to state the following theorem:

Theorem: Let $\theta : (R, +, \times) \longmapsto (T, \oplus, \otimes)$ be a surjective semiring homomorphism. Let relation $\sim$ be defined by:

$$x \sim y \Leftrightarrow \theta(x) = \theta(y)$$

Then, $\sim$ is a semiring congruence relation and $R/\sim$ is isomorphic to T. Conversely, if $\sim$ is a semiring congruence relation, then the mapping $\theta : R \longmapsto R/\sim$ such that for all $r \in R$

$$\theta(r) = [r]$$

is a surjective semiring homomorphism.

Proof: The proof is exactly the same (other than considering two operations instead of one) as the proof of the similar statement for a semigroup in Chapter 3. We leave the details of the proof to the reader. $\surd$

The above theorem states that all surjective homomorphic images T of a semiring R can be found (up to isomorphism) by finding all semiring congruence relations $\sim$ that exist in R. Therefore, all possible separate codes that protect a computation in a semiring R can be enumerated by finding all semiring congruence relations $\sim$ of R. In each case, the semiring T that provides the parity information is isomorphic to $R/\sim$ and H is isomorphic to $R \times R/\sim$. Of course, the enumeration of all semiring congruence relations might not be an easy task. However, we have obtained a complete characterization of the separate codes for a semiring computation. Moreover, we have an algebraic framework that can be used to study separate codes for semiring computations.

The notion of semiring congruence relations in a semiring R investigates surjective semiring homomorphisms by looking "globally" over the semiring and verifying that certain equivalence classes satisfy the requirements of a semiring congruence class. However, sometimes we would like to address more "local" questions: under what conditions could a given subset of elements of R correspond to a semiring congruence class (or, equivalently, to the complete preimage of an element under some semiring homomorphism of R). The remainder of this section answers this question by introducing the concepts of a *semiring complex* and a *semiring ideal* (which are really extensions of the normal complex and normal subsemigroup respectively).

Definition: A non-empty subset N of a semiring $(R, +, \times)$ is called a **semiring complex** if:

- For any $z, l, r \in R$ and for any $n_1, n_2 \in N$,

if $z + (l \times n_1 \times r) \in N$, then $z + (l \times n_2 \times r) \in N$.

The following theorem establishes the equivalence between a semiring complex and a semiring congruence class.

Theorem: For the subset N of a semiring R to be a complete preimage of one element under some surjective homomorphism of R, it is necessary and sufficient that N be a semiring complex.

Proof: The proof is rather long and is deferred to Appendix A. $\surd$

If we look at the special case of the congruence class that contains the additive identity 0_+ of R (that is, the kernel of the homomorphism) we get the following:

Definition: A non-empty subset U of a semiring $(R, +, \times)$ is called a **semiring ideal** if:

- $0_+ \in U$.
- For any $z, l, r \in R$ and for any $n_1, n_2 \in U$
  If $z + (l \times n_1 \times r) \in U$, then $z + (l \times n_2 \times r) \in U$.

Theorem: In order that the subset U of a semiring R should be, under some surjective semiring homomorphism $\theta : R \longmapsto T$, the complete preimage of the additive identity $0_\oplus$ of T, it is necessary and sufficient that U be a semiring ideal.

Proof: U has to be semiring complex and has to contain the additive identity (because for any semiring homomorphism $\theta(0_+) = 0_\oplus$). $\surd$

Before closing this section, let us make an observation about the form of the semiring ideal in the special case of a semiring in which the additive identity acts as a multiplicative zero that is, for any element $r \in R$:

$$0_+ \times r = r \times 0_+ = 0_+$$

Since $0_+ \in U$ we see that for any $u_1, u_2 \in U$ we need:

$$u_1 + (l \times u_2 \times r) \in U \text{ for all } l, r \in R$$

(this is easy to see — just express $u_1 = u_1 + l \times 0_+ \times r$ for any $l, r \in R$). If we set $r = l = 1_\times$, we see that U is a subsemigroup under the additive operation (in fact, it has to be a normal subsemigroup, as can be seen easily). If we set $u_1 = 0_+$ we see that for any $u \in U$ and all $l, r \in R$

$$l \times u \times r \in U$$

We conclude that in this special case a semiring ideal is a subset U of R such that:

- U is normal subsemigroup that contains 0_+ under the additive operation.
- For any $l, r \in R$ the set $l \times U \times r \subseteq U$.

Under these circumstances, the concept of a semiring ideal is exactly the same as in [24] and is very close to the concept of the ideal in the ring case that we studied in Section 5.2.

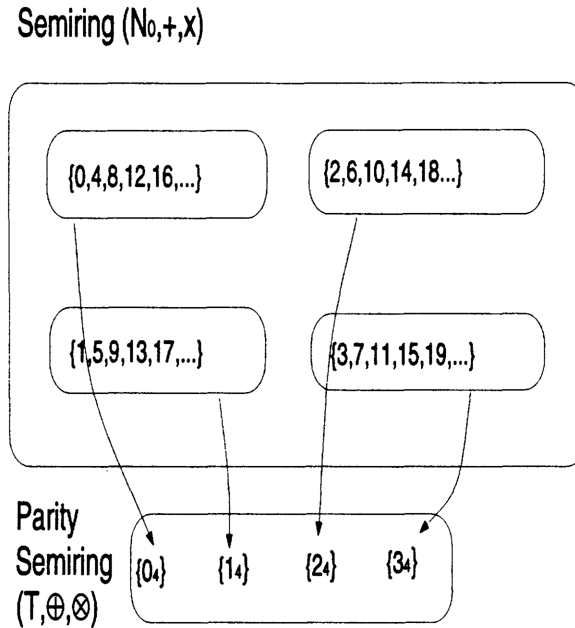


Figure 5-3: Example of a residue check mod 4 for the semiring $(\mathbf{N}_0, +, \times)$.

At this point, we conclude the theoretical analysis for the semiring case. In order to demonstrate these results and ways to use them, we present a few examples.

5.5 Examples in the Semiring-Theoretic Framework

5.5.1 Separate Codes for $(\mathbf{N}_0, +, \times)$

In this section we present examples of separate codes for $(\mathbf{N}_0, +, \times)$, the semiring of non-negative integers under addition and multiplication. Our analysis results in a complete characterization of the separate codes for this semiring. However, this is in some ways an unfortunate example, because the analysis is relatively easy. The elaborate theoretical analysis in the previous section can be verified, but those results are not really necessary in finding the possible separate codes.

By comparing $(\mathbf{N}_0, +, \times)$ with $(\mathbb{Z}, +, \times)$ we can easily see that a residue check can be used to protect computation in this semiring. Such a check corresponds to a parity channel that performs integer addition and multiplication modulo some positive integer M. Figure 5-3 shows an example of such a residue check when $M = 4$. The semiring congruence classes under this surjective homomorphism are shown in the figure. The operations $\oplus$ and $\otimes$ that take place in the parity semiring T are simply addition and multiplication modulo 4.

From the results of the previous section and from the examples in Chapter 4, we expect that this kind of separate code might not be the whole story. Any par-

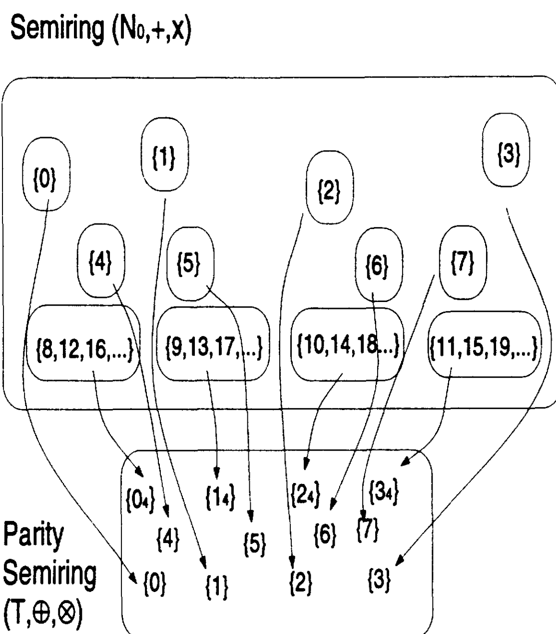


Figure 5-4: Example of a parity check code for the semiring $(\mathbf{N}_0, +, \times)$.

tition of $(\mathbf{N}_0, +, \times)$ into semiring congruence classes results in a semiring surjective homomorphism. We know that the semiring congruence classes for $(\mathbf{N}_0, +, \times)$ are congruence classes under *both* operations (addition and multiplication), that is, they are semigroup congruence classes for both $(\mathbf{N}_0, +)$ and $(\mathbf{N}_0, \times)$. Since we already have a complete characterization of the congruence classes for $(\mathbf{N}_0, +)$ (Chapter 4), we know that the semiring congruence classes for $(\mathbf{N}_0, +, \times)$ can only correspond to homomorphic mappings of the form:

1. For $n \in \mathbf{N}_0$, $\theta(n) = n \bmod M$, where M is any *finite* integer.

2. For fixed finite positive integers M, k, for $n \in \mathbf{N}_0$ we have:

$$\begin{aligned} \theta(n) &= n \text{ if } n < kM \\ \theta(n) &= n \bmod M, \text{ otherwise} \end{aligned}$$

Fortunately, the above classes also satisfy the congruence requirements under the multiplicative operation. Therefore, they can be used as separate codes for protecting computations in the $(\mathbf{N}_0, +, \times)$ semiring. In fact, this is a complete characterization of all separate codes for this semiring. In Figure 5-4 we present an example of a parity check code for $(\mathbf{N}_0, +, \times)$ that is of the second form mentioned above: if the result is less than 8, then the parity channel duplicates the computation, otherwise it simply performs addition or multiplication mod 4. The parity semiring T has operations $\oplus$ (additive) and $\otimes$ (multiplicative). The defining tables for them can be found in Table 5.1.

OPERATION $\oplus$

$\oplus$	0	1	2	3	4	5	6	7	0_4	1_4	2_4	3_4
0	0	1	2	3	4	5	6	7	0_4	1_4	2_4	3_4
1	1	2	3	4	5	6	7	0_4	1_4	2_4	3_4	0_4
2	2	3	4	5	6	7	0_4	1_4	2_4	3_4	0_4	1_4
3	3	4	5	6	7	0_4	1_4	2_4	3_4	0_4	1_4	2_4
4	4	5	6	7	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4
5	5	6	7	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4
6	6	7	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4
7	7	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4
0_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4
1_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4
2_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4
3_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4

OPERATION $\otimes$

$\otimes$	0	1	2	3	4	5	6	7	0_4	1_4	2_4	3_4
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	0_4	1_4	2_4	3_4
2	0	2	4	6	0_4	2_4	0_4	2_4	0_4	2_4	0_4	2_4
3	0	3	6	1_4	0_4	3_4	2_4	1_4	0_4	3_4	2_4	1_4
4	0	4	0_4	0_4	0_4	0_4	0_4	0_4	0_4	0_4	0_4	0_4
5	0	5	2_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4
6	0	6	0_4	2_4	0_4	2_4	0_4	2_4	0_4	2_4	0_4	2_4
7	0	7	2_4	1_4	0_4	3_4	2_4	1_4	0_4	3_4	2_4	1_4
0_4	0	0_4	0_4	0_4	0_4	0_4	0_4	0_4	0_4	0_4	0_4	0_4
1_4	0	1_4	2_4	3_4	0_4	1_4	2_4	3_4	0_4	1_4	2_4	3_4
2_4	0	2_4	0_4	2_4	0_4	2_4	0_4	2_4	0_4	2_4	0_4	2_4
3_4	0	3_4	2_4	1_4	0_4	3_4	2_4	1_4	0_4	3_4	2_4	1_4

Table 5.1: Defining tables of the operations $\oplus$ and $\otimes$ for the parity semiring T.

In Section 5.3, we saw that the only possible kind of separate code for computations in $(\mathbb{Z}, +, \times)$, the ring of integers under addition and multiplication, is a parity check that performs addition and multiplication modulo M (M being a positive integer) [16] [17] [1]. However, when concentrating on the less structured semiring of non-negative integers $(\mathbf{N_0}, +, \times)$, more possibilities are opened. We hope that, by using the semigroup framework, we can utilize this extra flexibility to discover efficient codes that suit our error detecting and correcting requirements.

5.5.2 Separate Codes for $(\mathbb{Z} \cup \{\pm\infty\}, \text{MIN}, \text{MAX})$

In this section we briefly discuss separate codes for $(\mathbb{Z} \cup \{\pm\infty\}, \text{MIN}, \text{MAX})$ the semiring of integers under the operations MIN (additive) and MAX (multiplicative). The same discussion applies if we switch the two operations (that is we make MAX the additive operation and MIN the multiplicative one).

We already know from the analysis of $(\mathbb{Z}, \text{MIN})$ and $(\mathbb{Z}, \text{MAX})$ in Chapter 4 that the congruence classes for *both* of these semigroups are intervals of consecutive integers. Therefore, we conclude that the semiring congruence classes for $(\mathbb{Z} \cup \{\pm\infty\}, \text{MIN}, \text{MAX})$ will be of the exact same form, that is consecutive intervals of integers. All of the examples of separate codes that we saw in Chapter 4 for $(\mathbb{Z}, \text{MAX}/\text{MIN})$ can be used without any modification to protect this semiring.

In fact, this is a complete characterization of all possible semiring congruence classes of $(\mathbb{Z} \cup \{\pm\infty\}, \text{MIN}, \text{MAX})$. Clearly, the same results apply to the semirings $(\mathbb{R} \cup \{\pm\infty\}, \text{MAX}, \text{MIN})$ and $(\mathbb{R} \cup \{\pm\infty\}, \text{MIN}, \text{MAX})$.

5.5.3 Separate Codes for $(\mathbb{Z} \cup \{-\infty\}, \text{MAX}, +)$

If we consider the congruence classes of the semigroups $(\mathbb{Z}, \text{MAX})$ and $(\mathbb{Z}, +)$ separately (both of them were analyzed in Chapter 4), we see that they have no common intersection (other than the trivial congruence classes[11]). Therefore, we conclude that no non-trivial separate codes can be used to protect computation in the semiring $(\mathbb{Z} \cup \{-\infty\}, \text{MAX}, +)$. Of course, the same results apply to $(\mathbb{Z} \cup \{+\infty\}, \text{MIN}, +)$, $(\mathbb{R} \cup \{-\infty\}, \text{MAX}, +)$, and $(\mathbb{R} \cup \{+\infty\}, \text{MIN}, +)$.

It is interesting to note that the above conclusion is false for the semiring $(\mathbf{N_0} \cup \{-\infty\}, \text{MAX}, +)$. As we saw in Chapter 4, the semigroup $(\mathbf{N_0}, +)$ allows for more kinds of surjective homomorphisms than the group $(\mathbb{Z}, +)$. This extra freedom suggests that there might exist non-trivial surjective homomorphisms for the semiring $(\mathbf{N_0} \cup \{-\infty\}, \text{MAX}, +)$. In fact, one such homomorphism (taken from [24]) is the following:

[11]Each semigroup/semiring can be partitioned into congruence classes in two trivial ways:

1. Take the whole set of elements as one big congruence class. This corresponds to a surjective homomorphism that maps everything to the zero element.
2. Make each element a separate congruence class. This corresponds to an isomorphism.

Let X_n be the finite set $\{-\infty, 0, 1, ..., n\}$, where $-\infty$ is assumed to satisfy the conditions $-\infty \leq i$ and $-\infty + i = -\infty$ for $i \in X_n$. Let $\circ$ be the operation $x \circ y = \text{MAX}(x, y)$ and $\diamond$ be the operation $x \diamond y = \text{MIN}(x + y, n)$. (One can easily check that $(X_n, \circ, \diamond)$ is a finite semiring.) Now, if we let $\delta_n : (\mathbf{N} \cup \{-\infty\}, \text{MAX}, +) \longmapsto (X_n, \circ, \diamond)$ be the mapping:

$$\begin{aligned} \delta_n(i) &= i \text{ , for } i \leq n \\ &= n \text{ , otherwise} \end{aligned}$$

for all $i \in \mathbf{N}_0$, we easily conclude that δ_n is a semiring homomorphism (the verification is left to the reader). In fact, this mapping is very reminiscent of the homomorphisms for $(\mathbf{N}_0, +)$ that we saw in Chapter 4, where up to a certain threshold (in this case n) the homomorphic semigroup duplicates the original one. Once we exceed the threshold, things get radically different.

5.6 Summary

In this chapter we developed frameworks for higher algebraic structures. More specifically, we extended the work of Chapters 2 and 3 to treat computations with an underlying ring or semiring structure. Many examples of arithmetic codes that fit in these frameworks were presented: codes for the ring of matrices, the ring of integers, the semiring of non-negative integers and other structures were presented. More importantly, we demonstrated the use of the tools and the results of our theoretical analysis (for groups/semigroups and rings/semirings) in constructing arithmetic codes for given computations. However, it is our belief that the potential applications of this algebraic framework go beyond what we have demonstrated here: it can be used for developing arithmetic codes that make efficient use of redundancy, for achieving efficient error correction procedures, and for exploiting more fully the error model.

Chapter 6

Summary of Contributions, and Suggestions for Future Research

In this thesis we have dealt with the problem of systematic design of arithmetic codes to protect a given computational task. Starting from a very general setting, in which we assumed that the computation can be modeled as a semigroup operation, we managed to extend a previous group-theoretic framework to encompass a more general set of computations. We were also able to extend our results to more complicated algebraic structures, such as rings and semirings. The end result is to provide an algebraic framework under which the design of arithmetic codes for semigroup and semiring computations can be formulated as a mathematical problem and be solved systematically.

6.1 Contributions and Conclusions

The starting point for this thesis was modeling a fault-tolerant system for a semigroup-based computation as a composition of three subsystems: a unit that performs computation in a larger *redundant* semigroup, an error corrector, and a decoder. We assumed that the error corrector and the decoder were error-free (that is, protected by modular redundancy or some other means), and concentrated on the redundant computation unit.

Under a few elementary requirements on the structure of the redundant computation, we showed that all possible ways of adding redundancy correspond to semigroup homomorphisms. Therefore, the redundant computation unit essentially performs a homomorphic computation in a redundant semigroup of higher order. The above is an important result because it places the problem of designing arithmetic codes into an algebraic framework.

We then used this framework to characterize the error detection and correction requirements for our arithmetic codes. Naturally, such a characterization requires an underlying error model. The choice depends on the actual hardware that is used to implement the computation. However, in order to demonstrate the use of the framework, we adopted an additive error model (as used in [1]) and managed to

characterize the redundancy requirements with respect to the error detecting and correcting capabilities of our codes.

In the special case of separate codes (that is, codes that provide fault tolerance through a completely independent parity channel operating in parallel with the original computational unit), we presented a constructive procedure that generates all possible separate codes for a given semigroup-based computation. This is an interesting result because it generalizes the previous procedure for group-based computations to a more general set of computations.

Having established the algebraic framework, we presented many examples of arithmetic codes that we constructed using the methods and the results mentioned above. The objective was to demonstrate the use of the framework on simple, well-known semigroup computations.

Finally, the framework was extended to the ring and semiring structures, which are higher algebraic structures with one group operation and one semigroup operation respectively. We then presented examples of arithmetic codes for these structures.

6.2 Future Research Directions

This thesis presented an algebraic framework for developing arithmetic codes. The framework is very general and applies to all computations that can be modeled as semigroup operations. The framework is also very theoretical, so there is a variety of practical issues to be considered, as well as a number of directions in which the framework can be extended.

6.2.1 Hardware Implementation and the Error Model

An arithmetic code can provide fault tolerance to a computational system if it protects the parts of the system that are liable to fail. Therefore, there is a definite need to map the abstract semigroup-theoretic formulation to actual hardware implementations, in order to connect the theoretical results that we obtained in this thesis with the actual implementation of our computational system. Hardware failure modes need to be explicitly reflected in our algebraic formulation. This is extremely important: the set of expected errors depends solely upon the specifics of our implementation, and it is against this set of errors that the fault-tolerant system should provide protection. As long as our model for the fault-tolerant computation does not provide direct links to the actual hardware that is used, the error model cannot be specified with complete success.

Choosing an error model should really be a tradeoff between simplicity in the algebraic formulation and effectiveness in reflecting the actual errors that take place in the system. For example, we have already seen that an additive error model was a suitable choice for computational tasks with an underlying group structure. Despite the fact that it might be a somewhat poor or inefficient reflection of the actual faults that can take place in the system, an additive error model for group-based computations results in a coset-based error correction procedure, which is not as complicated

as the general error correction procedure. However, an additive error model can be inefficient or intractable in situations where it does not directly reflect the actual error. For example, while a multiplicative error in a ring can be written as an additive error, the additive error will be operand-dependent even if the multiplicative error is not.

For a semigroup-based computation, the additive error model does not even result in a simplified error correction technique. Therefore, it is not even clear why we should use by default an additive error model (something that was done in the group case for the sake of coset-based error correction). An error model that appropriately reflects the hardware implementation and/or makes the task of error correction well-defined, systematic, and easy needs to be developed. In order to achieve this, the first step is to map the algebraic formulation onto an actual implementation. This was accomplished in the past for some specific computational tasks (for example, for operations of arithmetic processors in [16]). It is our hope that similar results can be achieved for the more general setting we have presented in this thesis.

6.2.2 Non-Abelian Group or Semigroup Computations

Our results regarding semigroup homomorphisms, as well as the results on group homomorphisms in [1], do not really require the underlying semigroup or group operations to be abelian. Therefore, an interesting direction to take is to investigate possibilities for extending this framework to non-abelian computations. Naturally, the analysis would get more complicated: we would need to look for *normal* subgroups and *normal* complexes, and we would have to be especially careful about the error model. For example, a simple additive error model would not be sufficient because a *right* error might behave differently than a *left* error.

6.2.3 Realizability of Arithmetic Codes

A question that was not addressed in this thesis was the realizability of an arithmetic coding scheme. An arithmetic code needs to be efficient in the sense that encoding and decoding of the data should be relatively easy to perform. A complicated arithmetic code is not desirable for the following reasons:

- If such a code is too complicated, then modular redundancy could be more efficient and much easier to implement.
- Complex coding and decoding is more liable to failures and invalidates our assumption that the encoders, error corrector and decoder are fault-free. This again reinforces the need for reflecting implementation in the algebraic formulation.

6.2.4 Development of a Probabilistic Framework

An interesting direction is the inclusion of a probabilistic model in our framework. Under such a model, we would be able to analyze the fault detection and correction

capabilities of a fault-tolerant system based on the prior probabilities of each of the errors. This opens up a number of intriguing possibilities:

- The computational system can be characterized in terms of an *average* or *expected* performance. If error correction is time consuming but errors occur infrequently enough, then we could afford to use an arithmetic code once we know that its overall performance will be adequate.

- Comparisons between fault-tolerant systems can be made on a fair basis.

- When a faulty result originates from more than one valid result, we could use a number of classic methods of detection and estimation to achieve optimal recovery from the error(s). By allowing different errors to reach the same invalid result in the redundant space, we relax the strict requirements on the redundancy of the arithmetic code, and, by taking advantage of our knowledge about the prior probability distributions of the errors, we can make efficient use of the redundancy of the code.

- A hierarchical division of the errors according to their prior likelihoods can make an important difference in the efficiency of the error correction techniques. If error correction guarantees fast recovery from the most probable error(s) then the average performance of the technique could be acceptable, even if certain (infrequent) errors take a lot of time to be identified.

6.2.5 Subsystem Decomposition and Machines

Depending on how one looks at a computational task, there might be different kinds of semigroup computations that can be associated with it. Consider the example of linear time-invariant (LTI) discrete-time (DT) filters of finite order N that operate on finite sequences. For simplicity, assume that the input sequences and the impulse responses of the filters take on only integer values. If we look at LTI DT filters as systems that produce one output at a time, we might think of the operation as a sequence of N additions and N multiplications in a ring. Alternatively, we could look at one multiplication and one addition at a time, which sets things up in the group of integers under addition and the semigroup of integers under multiplication. Yet another point of view is to consider the outputs of LTI DT filters as sequences, so we might associate the underlying computation with the semigroup of finite sequences (with integer values) under the operation of convolution. Similar possibilities exist for median and other nonlinear DT filters. In fact, most DT filters can be seen as computations in the semigroup of finite (or, more generally, infinite) sequences under some desirable operation.

Depending on the level at which we look at the problem, we see that there exists a variety of different approaches to protecting the given system. It would be interesting to study the various tradeoffs between these different approaches, as well as the differences in terms of the hardware overhead and the time delay involved. Since DT filters are so important in a variety of signal processing applications, investigating

ways of providing fault tolerance in such systems seems an extremely interesting research direction.

In fact, a computational machine can usually be regarded as an operation taking place in the set of strings: given a *collection of strings as an input*, there is a rule that specifies how to produce an *output string*. Therefore, with the right choice of an operator, we can model a machine as a semigroup operation taking place in the set of strings. An interesting future direction would be to investigate if and how the semigroup-theoretic framework can be used to provide fault tolerance to a given machine.

6.2.6 Links to the Theory of Error-Correcting Codes

If we restrict ourselves to a unary identity operation, then the arithmetic coding scheme essentially reduces to an error-correcting coding scheme, of the sort considered in communication theory. In fact, if the additive error model is a good description of the interaction between the codewords and the noise in a communication channel, then the framework with this error model (discussed in both Chapters 2 and 3) can be used to generate error-correcting codes.

Considering binary vector spaces as an extension of the group framework, we quickly arrive at the standard class of linear error-correcting codes [2]. In this case, the error-correcting code is really a subspace of a higher dimensional space: redundancy is introduced by mapping the original vector space V (consisting of the set of codewords that we would like to protect) into a higher-dimensional vector space H. Codewords from V get mapped to a subspace V' of H. Note that a subspace forms a group under the operation of addition and functions in the same way as a subgroup in the case of a group computation (Chapter 2). In fact, since the operation in this case is the identity operation, we can use any "coset" of H under V' (which, in this case, is the subspace V' "shifted" by some distance) to map codewords to. Codes created in this fashion are known as *coset* codes [2].

Similarly, the semigroup framework can be used to generate error-correcting codes. However, the additive error model in this case essentially reflects only errors that are unidirectional. In fact, asymmetric codes that have been already developed in some other fashion, like the Constantin-Rao single asymmetric error-correcting (SAEC) codes [2], can be placed in the semigroup framework quite comfortably. The big difference in this case is that, instead of looking at a subgroup and the cosets that it creates (all of which are necessarily sets of the same order of elements), we look at the normal subsemigroup and the corresponding normal complexes (which are *not* necessarily sets of the same order).

In recent years, an even broader definition of a *coset code* is used (see, for example, [26]). This definition allows almost all known coding techniques for band-limited channels, such as lattice codes and trellis codes, to be viewed as instances of a coset code. Under the framework of [26] and [27], a coset code is defined by a lattice[1]

[1] In [26], a real *lattice* is defined as a discrete set of vectors (points, N-tuples) in real Euclidean N-space $\mathbb{R}^N$ that forms a group under ordinary vector addition.

partition Λ/Λ' and by a binary encoder C that selects a sequence of cosets in the lattice Λ'. Moreover, geometric parameters (determined by the partition Λ/Λ' and the encoder C) can be related quite naturally to fundamental coding parameters, such as *distance* and *coding gain.* An interesting future direction would be to investigate whether a framework similar to this can also be used for arithmetic codes. Naturally, this would impose stricter requirements on the framework. However, since the notions of redundancy and distance of an arithmetic code would immediately have geometrical interpretations, it seems worthwhile to pursue research in this direction.

Appendix A

Proofs of Theorems

A.1 Enumerating all Separate Codes for $(\mathbf{N}_0, +)$

Here, we prove the claim made in Chapter 4 about the form of the parity checks for $(\mathbf{N}_0, +)$, the semigroup of non-negative integers under the operation of addition.

Specifically, we prove the following claim:

Claim: All possible surjective homomorphisms θ from the semigroup $(\mathbf{N}_0, +)$ onto a semigroup T have one of the following two forms:

1. For $n \in \mathbf{N}_0$, $\theta(n) = n \bmod M$, where M is any *finite* integer (if M is infinite, then θ can be thought of as an isomorphism). This kind of homomorphisms map $(\mathbf{N}_0, +)$ onto a finite cyclic *group* of order M[1].

2. For a finite integer M, for $n \in \mathbf{N}_0$ we have:
   $\theta(n) = n$ if $n < kM$ (for a fixed positive integer k)
   $\theta(n) = n \bmod M$, otherwise
   This kind of homomorphisms map $(\mathbf{N}_0, +)$ to a finite *semigroup* of order $M + kM = (k+1)M$[2].

Proof: Let $\theta : \mathbf{N}_0 \mapsto T$ be an onto homomorphism and let $\sim$ be the corresponding congruence relation defined on the elements of $\mathbf{N}_0$. Then, θ maps each of the congruence classes of the semigroup $\mathbf{N}_0$ to an element of the semigroup T. Moreover, since θ is onto, any element of T has a non-empty *preimage* under θ which is a congruence class.

The following two theorems from [21] (adjusted for the sake of simplicity for the abelian monoid case) show that congruence classes are equivalent to (normal) *com-*

[1] In the special case where $M = 0$, we get a trivial homomorphism from $(\mathbf{N}_0, +)$ onto the trivial semigroup of a single identity element.

[2] When $M = 0$ the mapping reduces to:
$\theta(n) = n$ if $n < k$ (for a fixed positive integer k)
$\theta(n) = 0_0$, otherwise
where 0_0 is an element in the homomorphic image of $(\mathbf{N}_0, +)$ that "almost" behaves as the zero element.

plexes and (normal) *subsemigroups*[3]:
Theorem 1: In order that the subset C of a monoid M should be a complete preimage of one element under some homomorphism of M, it is necessary and sufficient that C is a (normal) complex.
Theorem 2: In order that the subset C of a monoid M should, under some homomorphism θ, be a complete preimage of the identity element of the monoid $\theta(M)$, it is necessary and sufficient that C is a (normal) subsemigroup[4].

Once we have established an equivalence between the congruence classes of $\mathbf{N}_0$ under the relation $\sim$ and its (normal) complexes and (normal) subsemigroup, we can characterize all surjective homomorphisms $\theta : \mathbf{N}_0 \mapsto T$ by characterizing all ways of partitioning $\mathbf{N}_0$ into sets that comprise the (normal) complexes and the (normal) subsemigroup.

From the definition of a (normal) complex (the subsemigroup) we can easily conclude that there are only two possibilities for a complex (the subsemigroup):

- It consists of a single element (in which case it trivially satisfies the definition of a normal complex) or,
- It consists of an infinite number of elements and it is of the form:

$$\{k + iM\} \text{ for fixed positive integers } k, M \text{ and } i \in \{0, 1, 2, \ldots\}$$

The reason is simple: if a complex C contains at least two elements, say $k, k' \in C$, then we can always write the "largest" one (that is, the one that is generated using the generator more times) in terms of the other. For example, if we assume without loss of generality that k' is the "largest" one, we can write: $k' = k + M$, where M was chosen accordingly. Then, the definition of a complex forces all elements of the form $k + iM$ for $i \in \{0, 1, 2, \ldots\}$ to lie in C. This can be proved easily by induction. Note that M has to be the "smallest difference" between elements in C.

Once the form of the (normal) subsemigroup and the (normal) complexes is known, all that's left to show to complete the proof is the following:

1. The "step" number M is the same for all infinite complexes.
2. If each infinite complex C_i starts at a value k_i, then all k_i have to lie in an interval $[\lambda M, \ldots, (\lambda M + (M-1))]$ for an appropriately chosen positive integer λ.

The first statement can be proved by contradiction: if it is *not* true, then there exist two different complexes C_1 and C_2 such that:

$$\begin{aligned} C_1 &= \{k_1 + iM_1 \mid i \in \{0, 1, 2, \ldots\}\} \\ C_2 &= \{k_2 + jM_2 \mid j \in \{0, 1, 2, \ldots\}\} \end{aligned}$$

[3]These were defined in Chapter 3. A normal complex is a nonempty subset C of an abelian semigroup $\mathcal{S} = (S, \circ)$, such that for any $x \in S$ and for any $k, k' \in C$, $x \circ k \in C$ always implies $x \circ k' \in C$. A normal subsemigroup is a nonempty subset C of an abelian semigroup S such that for any $x \in S$ and for any $k, k' \in C$ or being empty symbols, $x \circ k \in C$ always implies $x \circ k' \in C$.

[4]In the monoid case, this simplifies to a normal complex that contains the identity.

where M_1 and M_2 are *different* integers.

Let's assume without loss of generality that $k_1 = k_2 + d$ (where d is a non-negative integer)[5]. Then, by the definition of a congruence class (Chapter 3), the set given by:

$$\begin{aligned} d + C_1 &= \{d + k_1 + iM_1 \mid i \in \{0, 1, 2, ...\}\} \\ &= \{k_2 + iM_1 \mid i \in \{0, 1, 2, ...\}\} \end{aligned}$$

has to be a subset of a congruence class. Since it intersects the congruence class C_2 (to see this, just let $i = j = 0$) it has to be a subset of it. So, $d + C_1 \subset C_2$. This can only hold if $M_1 = \mu M_2$ where μ is a positive integer (for an infinite complex $M \neq 0$). Therefore, the two congruence classes are as follows:

$$\begin{aligned} C_1 &= \{k_1 + i\mu M \mid i \in \{0, 1, 2, ...\}\} \\ C_2 &= \{k_2 + jM \mid j \in \{0, 1, 2, ...\}\} \end{aligned}$$

where we have set $M \equiv M_2$ for simplicity.

A similar argument can be made the other way: for a large enough positive integer α, we can find an integer d' such that:

$$k_2 + d' = \alpha\mu M + k_1$$

Then, we can conclude that the set given by:

$$\begin{aligned} d' + C_2 &= \{k_2 + d' + jM \mid j \in \{0, 1, 2, ...\}\} \\ &= \{k_1 + \alpha\mu M + jM \mid j \in \{0, 1, 2, ...\}\} \\ &= \{k_1 + (\alpha\mu + j)M \mid j \in \{0, 1, 2, ...\}\} \end{aligned}$$

is a subset of a congruence class. Since it intersects C_1 (for example, let $i = 2\alpha$, $j = \alpha\mu$), it has to be a subset of C_1. Clearly, this is possible only if $\mu = 1$.

Therefore, all infinite congruence classes can only be of the form:

$$C_i = \{k_i + jM \mid j \in \{0, 1, 2, ...\}\}$$

and each of the other congruence classes consists of a single element.

It remains to show that all $k_i \in [\lambda M, ..., (\lambda M + (M - 1))]$ for an appropriately chosen λ. The proof is again by contradiction. If the above was not true, then there would exist two infinite congruence classes C_1, C_2 such that:

$$\begin{aligned} C_1 &= \{k_1 + iM \mid i \in \{0, 1, 2, ...\}\} \\ C_2 &= \{k_2 + jM \mid j \in \{0, 1, 2, ...\}\} \\ k_2 &= k_1 + \alpha M + d \end{aligned} \tag{A.1}$$

where α is a strictly positive integer, and, without loss of generality, we have assumed

[5] Since $(\mathbf{N}_0, +)$ is a cyclic semigroup, we always have either $k_1 = k_2 + d$ or $k_2 = k_1 + d$.

$k_1 < k_2$.

Then, for large enough i, the set $d + C_1$ intersects the congruence class C_2; it is therefore a subset of it. However, that is impossible unless $\alpha = 0$.

At this point, the proof of the claim is complete. We have demonstrated that the only two kinds of separate codes for the $(\mathbf{N}_0, +)$ monoid are as given in the beginning of this section[6]. $\surd$

A.2 Equivalence of Semiring Congruence Classes and Semiring Complexes

In this section we prove the following theorem from Section 5.4:

Theorem: For the subset N of a semiring R to be a complete preimage of one element under some surjective homomorphism of R, it is necessary and sufficient that N be a semiring complex.

Proof: First, we prove that in order for the subset N to be a complete preimage of some element under some surjective homomorphism of R, it has to be a semiring complex.

Let N be the complete preimage of some element $t \in T$ under a surjective homomorphism $\theta : (R, +, \times) \longmapsto (T, \oplus, \otimes)$. Then

$$\theta(n) = t \text{ for all } n \in N$$

For any elements $n_1, n_2 \in N$ and any elements $z, l, r \in R$, if $z + (l \times n_1 \times r) \in N$ then $\theta(z) \oplus (\theta(l) \otimes \theta(n_1) \otimes \theta(r)) = t$. Since $\theta(n_1) = \theta(n_2) = t$, we conclude that $\theta(z) \oplus (\theta(l) \otimes \theta(n_2) \otimes \theta(r)) = t$. Therefore, the element $z + (l \times n_2 \times r) \in N$. This establishes that N is a semiring complex.

Now, we prove the other direction of the theorem: a semiring complex N is the complete preimage of an element under some (surjective[7]) semiring homomorphism θ. All we need to do is construct a homomorphism θ under which N is the complete preimage of an element of $\theta(R)$. Equivalently, we can construct a semiring congruence relation $\sim$ under which N is a complete congruence class. In fact, we follow the later approach[8].

First, we define in R the relation $\sim'$: for elements $r_1, r_2 \in R$ we say that "r_1 is related to r_2 through $\sim'$ " (notation $r_1 \sim' r_2$) if and only if:

- $r_1 = r_2$ *or*
- There exist $n_1, n_2 \in N$ and $z, l, r \in R$ such that we can write:

$$r_1 = z + (l \times n_1 \times r)$$

[6] It is not hard to verify that these two kinds of separate codes in fact work, and we leave that verification to the reader.

[7] The proof does not really require the semiring homomorphism to be surjective.

[8] We basically refine the relations used for semigroups in [21] to account for the fact that we are dealing with a semiring.

$$r_2 = z + (l \times n_2 \times r)$$

Note that the above relation is not quite an equivalence relation. It is clearly reflexive (for all $r \in R$, $r \sim' r$) and symmetric (for all $r_1, r_2 \in R$, if $r_1 \sim' r_2$ then $r_2 \sim' r_1$). To create an equivalence relation, we need the transitivity property. Therefore, we define the relation $\sim$ as follows: $r_1 \sim r_2$ if and only if:

- There exists a finite sequence of elements $\{z_1, z_2, ..., z_n\} \in R$ such that:

$$r_1 \sim' z_1 \sim' z_2 \ldots \sim' z_n \sim' r_2$$

One can easily check that relation $\sim$ is an equivalence relation (it is reflexive, symmetric and transitive). Furthermore, it is a semiring congruence relation. To show that, we need to show two things:

1. For all $a, b, c, d \in R$, if $a \sim b$ and $c \sim d$ then $(a + b) \sim (c + d)$, and
2. For all $a, b, c, d \in R$, if $a \sim b$ and $c \sim d$ then $(a \times b) \sim (c \times d)$.

We start by showing the truth of the first statement. First, we will show that for all $a, b, c, d \in R$, if $a \sim' b$ and $c \sim' d$ then $(a+b) \sim (c+d)$. For simplicity of notation, we assume that multiplication precedes addition whenever parentheses are not used to indicate the order of operations explicitly. We break the problem into four different cases:

Case 1: There exist z_1, l_1, r_1 and z_2, l_2, r_2 in R and n_1, n_2, n_3, n_4 in N such that:

$$\begin{aligned} a &= z_1 + l_1 \times n_1 \times r_1 \\ b &= z_1 + l_1 \times n_2 \times r_1 \\ c &= z_2 + l_2 \times n_3 \times r_2 \\ d &= z_2 + l_2 \times n_4 \times r_2 \end{aligned}$$

By adding a to c and d, we see that:

$$\begin{aligned} (a + c) &= (z_1 + z_2 + l_1 \times n_1 \times r_1) + l_2 \times n_3 \times r_2 \\ &= z' + l_2 \times n_3 \times r_2 \\ (a + d) &= (z_1 + z_2 + l_1 \times n_1 \times r_1) + l_2 \times n_4 \times r_2 \\ &= z' + l_2 \times n_4 \times r_2 \end{aligned}$$

where $z' = z_1 + z_2 + l_1 \times n_1 \times r_1$.

We conclude that $(a + c) \sim' (a + d)$. Similarly, by adding d to a and b, we conclude that $(a + d) \sim' (b + d)$. By the transitivity of $\sim$, we arrive at the important result $(a + c) \sim (b + d)$. By induction, we can easily show that for all $a, b, c, d \in R$, if $a \sim b$ and $c \sim d$ then $(a + c) \sim (b + d)$.

Case 2: $a = b$ and c, d as above. The proof is basically the same as above (in fact, easier).

Case 3: $c = d$ and a, b as in case 1. The proof is the same as in case 2.

Case 4: $a = b$ and $c = d$. Clearly $a + c = b + d$ and therefore, $(a + c) \sim (b + d)$.

Now we show that for all $a, b, c, d \in R$, if $a \sim' b$ and $c \sim' d$ then $(a \times c) \sim (b \times d)$. If $a \sim' b$ and $c \sim' d$ then we have the exact same cases we had above:

Case 1: There exist z_1, l_1, r_1 and z_2, l_2, r_2 in R and n_1, n_2, n_3, n_4 in N such that:

$$\begin{aligned} a &= z_1 + l_1 \times n_1 \times r_1 \\ b &= z_1 + l_1 \times n_2 \times r_1 \\ c &= z_2 + l_2 \times n_3 \times r_2 \\ d &= z_2 + l_2 \times n_4 \times r_2 \end{aligned}$$

Then, by left multiplying c and d by a we see that:

$$\begin{aligned} (a \times c) &= (z_1 \times z_2 + l_1 \times n_1 \times r_1 \times z_2) + (z_1 + l_1 \times n_1 \times r_1) \times l_2 \times n_3 \times r_2 \\ &= z' + l' \times n_3 \times r_2 \\ (a \times d) &= (z_1 \times z_2 + l_1 \times n_1 \times r_1 \times z_2) + (z_1 + l_1 \times n_1 \times r_1) \times l_2 \times n_4 \times r_2 \\ &= z' + l' \times n_4 \times r_2 \end{aligned}$$

where $z' = z_1 \times z_2 + l_1 \times n_1 \times r_1 \times z_2$ and $l' = (z_1 + l_1 \times n_1 \times r_1) \times l_2$.

We conclude that $(a \times c) \sim' (a \times d)$. Similarly, by right multiplying a and b by d, we conclude that $(a \times d) \sim' (b \times d)$. By the transitivity of $\sim$, we arrive at the important result $(a \times c) \sim (b \times d)$. By induction, we can easily show that for all $a, b, c, d \in R$, if $a \sim b$ and $c \sim d$ then $(a \times c) \sim (b \times d)$.

Case 2: $a = b$ and c, d as above. The proof is basically the same as above.

Case 3: $c = d$ and a, b as in case 1. The proof remains the same.

Case 4: $a = b$ and $c = d$. Clearly $a \times c = b \times d$ and therefore, $(a \times c) \sim (b \times d)$.

We conclude that $\sim$ is a semiring congruence relation. The only thing left to do to complete the proof of the theorem is to show that N is the complete preimage of an element $t \in T$ under θ, the homomorphism corresponding to relation $\sim$. This is easy: from the definition of N we see that if $n \sim x$ and $n \in N$ then $x \in N$. Therefore, N is a complete semiring congruence class under $\sim$.

At this point, the proof of the theorem (both directions) is complete. $\surd$

Bibliography

[1] P. E. Beckmann, *Fault-Tolerant Computation Using Algebraic Homomorphisms.* PhD Thesis, EECS, Massachusetts Institute of Technology, Cambridge, MA, 1992.

[2] T.R.N. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[3] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers,* vol. 33, pp. 518-528, June 1984.

[4] J.-Y. Jou and J. A. Abraham, "Fault-tolerant matrix arithmetic and signal processing on highly concurrent parallel structures," *Proceedings of the IEEE,* vol. 74, pp. 732-741, May 1986.

[5] V. S. S. Nair and J. A. Abraham, "Real-number codes for fault-tolerant matrix operations on processor arrays," *IEEE Transactions on Computers,* vol. 39, pp. 426-435, April 1990.

[6] J.-Y. Jou and J. A. Abraham, "Fault-tolerant FFT networks," *IEEE Transactions on Computers,* vol. 37, pp. 548-561, May 1988.

[7] J. A. Abraham, "Fault tolerance techniques for highly parallel signal processing architectures," *Proc. of SPIE,* vol. 614, pp. 49-65, 1986.

[8] 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,* pp. 65-75, July 1987.

[9] C.-Y. Chen and J. A. Abraham, "Fault-tolerance systems for the computation of eigenvalues and singular values," *Proc. of SPIE,* vol. 676, pp. 228-237, August 1986.

[10] P. E. Beckmann and B. R. Musicus, "Fault-tolerant round-robin A/D converter system," *IEEE Transactions on Circuits and Systems,* vol. 38, pp. 1420-1429, December 1991.

[11] P. E. Beckmann and B. R. Musicus, "Fast fault-tolerant digital convolution using a polynomial residue number system," *IEEE Transactions on Signal Processing*, vol. 41, pp. 2300-2313, July 1993.

[12] C. J. Anfinson, R. P. Brent, and F. T. Luk, "A theoretical foundation for the Weighted Checksum scheme," *Proc. of SPIE,* vol. 975, pp. 10-18, 1988.

[13] H. Park, "Multiple error algorithm-based fault tolerance for matrix triangularizations," *Proc. of SPIE,* vol. 975, pp. 258-267, 1988.

[14] C. J. Anfinson and B. L. Drake, "Triangular systolic arrays and related fault tolerance," *Proc. of SPIE,* vol 826, pp. 41-46, 1987.

[15] I. N. Herstein, *Topics in Algebra.* Xerox College Publishing, Lexington, Massachusetts, 1975.

[16] T.R.N. Rao, *Error Coding for Arithmetic Processors.* Academic Press, New York, 1974.

[17] W. W. Peterson and E. J. Weldon Jr, *Error-Correcting Codes.* The MIT Press, Cambridge, Massachusetts, 1972.

[18] R. Lidl and G. Pilz, *Applied Abstract Algebra.* Undergraduate Texts in Mathematics, Springer-Verlag, New York, 1985.

[19] P. M. Higgins, *Techniques of Semigroup Theory.* Oxford University Press, New York, 1992.

[20] G. Lallement, *Semigroups and Combinatorial Applications.* John Wiley and Sons, New York, 1979.

[21] E. S. Ljapin, *Semigroups.* Volume Three, Translations of Mathematical Monographs, American Mathematical Society, Providence, Rhode Island, 1974.

[22] L. Fuchs, *Abelian Groups.* Pergamon Press, Oxford, New York, 1967.

[23] N. Jacobson, *Basic Algebra I.* W. H. Freeman and Company, San Francisco, 1974.

[24] J. S. Golan, *The Theory of Semirings with Applications in Mathematics and Theoretical Computer Science.* Longman Scientific & Technical, Essex, England, 1992.

[25] W. Kuich and A. Salomaa, *Semirings, Automata, Languages.* Monographs in Theoretical Computer Science, Springer-Verlag, New York, 1986.

[26] G. D. Forney, Jr., "Coset Codes — Part I: Introduction and Geometrical Classification," *IEEE Transactions on Information Theory,* vol. 34, pp. 1123-1151, September 1988.

[27] G. D. Forney, Jr., "Coset Codes — Part II: Binary Lattices and related codes," *IEEE Transactions on Information Theory,* vol. 34, pp. 1152-1187, September 1988.