Order Number 9521248

The radius of Sheffer functions over E(3)

Beckman, Jeffrey, Ph.D.

City University of New York, 1995

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

THE RADIUS OF SHEFFER FUNCTIONS OVER E(3)

by

JEFFREY BECKMAN

A dissertation submitted to the Graduate Faculty in Computer Science in partial fullfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

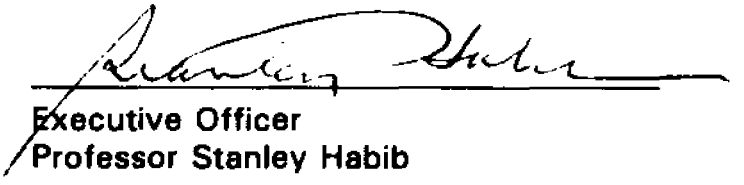
1995

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

1 Nov 1994
Date


Chair of Examining Committee
Professor T. C. Wesselkamper

23, Nov '94
Date


Executive Officer
Professor Stanley Habib

Supervisory Committee
Professor Ravi Kulkarni
Professor John Loustau

THE CITY UNIVERSITY OF NEW YORK

ABSTRACT

THE RADIUS OF SHEFFER FUNCTIONS OVER E(3)

by

Jeffrey Beckman

Adviser: Dr. T.C. Wesselkamper

Through the operation of composition, a two-place Sheffer function ranging over a set E(k) = {0,1,2...,k-1} is a function capable of generating all other two-place functions ranging over the same set. This dissertation introduces the *radius* of the Sheffer function as a measure of the efficiency with which it performs this task. The radius *r* is defined as the least natural number such that all functions may be formed with *r* or fewer compositions of the Sheffer function.

The case of two-place functions over E(3) is examined in detail. It is demonstrated that the 3,774 Sheffer functions may be partitioned into 322 mutually exclusive classes, each having a single value of the radius. Measurements of the radius are given for all 322 classes, with values ranging from 7 to 25.

A set of 5 conditions on the operation table of a Sheffer function is presented that serves to distinguish the efficient functions from the inefficient ones.

ACKNOWLEDGMENTS

Thanks to Professor Ravi Kulkarni and Professor John Loustau for their helpful suggestions; to my fiancé Dianne for her invaluable help in preparing this thesis, and for her understanding during this long process; to my sister Bliss for her assistance in obtaining research documents, and for her frequent pep talks that served to keep me on track; to my parents Sam and Frieda for their interest and concern. Finally, a special heartfelt thanks to Professor Tom Wesselkamper, my mentor in the truest sense of the word. Eternally upbeat, supportive, encouraging, an ever-available source of technical advice, without Professor Wesselkamper this thesis would never have been brought to fruition.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1 Introduction

Traditionally, digital electronic circuits have been synonymous with the binary number system because the high and low voltage levels available offer a natural correspondence with the binary digits 0 and 1. The elements used to synthesize these circuits have two inputs and single output each taking values from the set {0, 1}. It has proven convenient to select a single element as a universal building block, and then for designers to devise a way of interconnecting copies of this element in order to construct a circuit satisfying a particular objective. Obviously, through interconnection of copies of the universal element, it must be possible to produce a circuit for every function of two arguments taking values from {0,1}. It is well-known that the two functions commonly referred to as *nand* and *nor* have this property.

In recent times, with the manufacture of multiple-level logic devices such as tri-level transistors, and with the replacement of electronic circuits by optical circuits where intensity levels need not be limited to high and low values, the restriction of two-valued inputs and outputs has been removed. Consequently, for designing multi-level logic circuits where inputs and

outputs can take on values from some set {0, 1, 2,..., k}, it is desirable to identify a circuit element with which, through interconnection, a circuit can be constructed for every two-place function. And, if there are many choices for such an element it would be helpful to identify those that are in some sense most efficient. It is with these problems that this dissertation is concerned.

1.2 Terminology and Notation

For each natural number k let $E(k)$ represent the finite set {0, 1, 2,..., k-1}, and let $E^n(k)$ represent the n-fold Cartesian product of $E(k)$ with itself, i.e.,

$$E^n(k) = E(k) \times E(k) \times \ldots E(k).$$

We are concerned with functions of n variables where the domain of each function is $E^n(k)$ and the range of each is in $E(k)$. Such a function is termed an n-place function, and is denoted by $f: E^n(k) \rightarrow E(k)$. If for the assignment $(x_1, x_2, \ldots, x_n)$, the function takes the value y, then the familiar notation $f(x_1, x_2, \ldots, x_n) = y$ is used, where no distinction is made between the constants x_i and the constant functions which assume the x_i as their values.

The constant functions are a special set of functions over $E(k)$, defined as: $C(k) = \{c_j: E^n(k) \rightarrow E(k) \mid \text{for all } x_i \in E(k),\ c_j(x_1, x_2, \ldots x_n) = j\}$.

The fundamental operation on functions is the familiar one of composition, where it is allowed to substitute a function for one of the

variables of another function. If $f = f(x_1,x_2,\ldots,x_n)$ and $g = g(u_1,u_2,\ldots,u_m)$, substituting g for any of the variables in f will result in a function f' which is function of $n + m - 1$ variables (although it may be less if for any values of i, $u_i = x_i$). Muzio and Wesselkamper (1986) present an axiomatic way of defining legal compositions through the repeated application of three functions, P, Q and R, and a single two-place operator *. Operating on a function $f(x_1,x_2,\ldots,x_n)$, Pf performs a cyclic permutation of the variables, and Qf performs a transposition of the first two variables. Using the two, any permutation of the variables can be effected. As Rf identifies the first two variables and moves all other variables one place to the right, P, Q, and R are sufficient to permute and identify variables. The operation * handles the special case where the first variable in f is replaced by a function g, i.e.,

$$(f*g)(x_1,x_2,\ldots,x_j) = f(g(x_1,x_2,\ldots,x_m), x_{m+1},\ldots,x_j), \text{ where } j = n + m - 1.$$

As an example, consider two functions defined by the rules $f(x,y,z) = x + yz$ and $g(u,v) = uv$. The composite function $z + uvx$ can be defined as follows.

$$Pf = f(y,z,x),$$

$$Pf*g = f(g(u,v),z,x),$$

$$Q(Pf*g) = f(z,(g(u,v),x) = z + uvx.$$

Suppose $g_i(x_1,x_2,\ldots,x_n)$, $(i = 1, 2,\ldots, n)$ is a given set of functions (where any or all of the x_i may be missing from any g_i). If in the function $f(x_1,x_2,\ldots,x_n)$ each of the x_i is restricted to be one of the g_i, then

$f(x_1,x_2,\ldots,x_n)$ becomes a function $h(x_1,x_2,\ldots,x_n)$, where $h(x_1,x_2,\ldots,x_n) = f[g_1,g_2,\ldots,g_n](x_1,x_2,\ldots,x_n)$, or with the dependence on the x_i suppressed, $h(x_1,x_2,\ldots,x_n) = f(g_1,g_2,\ldots,g_n)$. We say that h is *generated* by the functions f and the g_i.

Using a two-place function as an illustration, starting with $f(x,y)$ the functions $f(x,y)$, $f(x,x)$, $f(y,x)$, $f(f(x,y),x)$, etc., can be formed. If $h(x,y) = f(f(x,y),y)$, then h is generated by f, and the expression $f(f(x,y),y)$ is called the *composition sequence* of g in terms of f. A more detailed discussion of composition sequences is postponed until Chapter 3.

DEFINITION 1: A set of functions A is *complete* if for each natural number n, every function $f: E^n(k) \rightarrow E(k)$, may be generated by some composition of functions in A.

DEFINITION 2: A set of functions A is *complete with constants* if the set $A \cup C(k)$ is complete.

Obviously, any set of functions that is complete is also complete with constants.

If a complete set of functions consists of a single function, that function is called a *Sheffer* function. If a complete with constants set of functions consists of a single function, that function is called a *Sheffer with constants* function.

1.3 Function Specification

There are k^{k^n} n-place functions over $E(k)$. Each function can be defined by listing its value for every possible assignment of values for its variables. This listing can be organized in the form of an n-dimensional table termed the operation table of the function. For a two-place function over $E(k)$ the operation table is a square matrix with k rows and k columns, and is defined so that the following

$$\begin{vmatrix} a_{00} & a_{01} & \cdots & a_{0\,k-1} \\ a_{10} & a_{11} & \cdots & a_{1\,k-1} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ a_{k-1\,0} & a_{k-1\,1} & \cdots & a_{k-1\,k-1} \end{vmatrix}$$

represents the function $f(x,y)$ that takes the value a_{ij} when x takes the value i and y takes the value j. *The diagonal* of the operation table refers to that consisting of the elements $f(i,i)$, $(i = 0, 1, 2, \ldots, k-1)$.

Under a permutation P_i of the values $0, 1, 2, \ldots, k-1$, the operation table of a function f is transformed into the operation table of a function f_i. If for all $k! - 1$ permutations (excluding the identity permutation) the f_i are distinct, the function f is described as *fully-conjugated*.

The value sequence is an alternative representation to the operation table which is often convenient for one and two-place functions. For a one-place

function, the value sequence is written $f = \langle a_0\ a_1\ a_2 \ldots a_{k-1} \rangle$, where $f(i) = a_i$, $(0 \leq i \leq k - 1)$, $a_i \in E(k)$. For a two-place function, the value sequence is written $f = \langle a_{00}, a_{01}, a_{02}, \ldots, a_{k-1\,k-1} \rangle$, where $f(i,j) = a_{ij}$, $(0 \leq i,j \leq k - 1)$, $a_{ij} \in E(k)$.

When referring to two-place functions over *E(2)* the prefix notation of Bochenski (1959) is used, where each function is represented by a single letter. The letter designations, value sequences and common designations of each of the 16 two-place functions over *E(2)* is given in the table below.

Prefix Symbol	Value Sequence	Common Designation
O	<0 0 0 0>	constant 0
K	<0 0 0 1>	and
L	<0 0 1 0>	non-implication
I	<0 0 1 1>	x
M	<0 1 0 0>	non-implication
H	<0 1 0 1>	y
J	<0 1 1 0>	exclusive or
A	<0 1 1 1>	inclusive or
X	<1 0 0 0>	nor
E	<1 0 0 1>	equivalence
G	<1 0 1 0>	$\overline{y}$
B	<1 0 1 1>	implication
F	<1 1 0 0>	$\overline{x}$
C	<1 1 0 1>	implication
D	<1 1 1 0>	nand
V	<1 1 1 1>	constant 1

Figure 1. Bochenski's notation for functions over *E(2)*

1.4 Background

The first description of a Sheffer function is in a paper written by the eponymous author H.M. Sheffer (1913), published in 1913. Sheffer begins the paper with a statement of five independent postulates for Boolean algebras, which include one binary rule of combination of elementary propositions, symbolized $p|q$. He demonstrates that his set of postulates is equivalent to the traditional set of ten postulates, enunciated by Huntington, posed in terms of two binary rules of combination, $\oplus$ and $\odot$ (commonly written as $+$ and $\cdot$).

In the latter part of the paper, Sheffer discusses the application of his five postulates to logic, and shows that any system based on the logical operations of disjunction and negation is equivalent to a system based on his operation, which he named *rejection* (commonly known today as *nor*). He defines rejection in terms of negation (N) and disjunction (A) as

$$p|q = NApq,$$

and defines negation and disjunction in terms of rejection as

$$Np = p|p$$

$$Apq = (p|q)|(p|q).$$

In a footnote, Sheffer indicates that by "the principle of duality" his function could also be interpreted as the logical constant *either not-p or not-q*, commonly known today as the *nand* function.

Sheffer does not, in fact, show that his function is complete, only that it can be used to define negation and disjunction. With the subsequent work of Post, demonstrating that the set $\{A, N\}$ is complete, however, the completeness of Sheffer's function follows.

In 1921, Post (1921) shows that in *E(2)* the set of functions $\{A, N\}$ is complete, and, additionally, that a single function whose truth table is either of the following

+	+	-
+	-	+
-	+	+
-	-	+

+	-	-
+	-	-
-	-	-
-	+	+

is also complete, identifying the two truth tables as those of the functions introduced by Sheffer.

In the latter part of his paper, Post considers the extension of these results to systems with m truth values and functions of n arguments, which he says, "seem to have the same relation to ordinary logic that geometry in a space of an arbitrary number of dimensions has to the geometry of Euclid." He defines two primitives. The first, a generalization of negation, is

$$g(x) = (x - 1)_m ,$$

and the second, a generalization of disjunction, is

$$f(x,y) = max(x,y).$$

(In Post's paper, the truth values considered are $t_0, t_1, \ldots t_m$, with the largest value t_0 and the smallest value t_m. His two primitives are actually specified in terms of their affect on the subscripts of those truth values.

The above formulations are a translation in terms of the logical variables x, and y.)

Using these two primitives, Post constructs a function able to generate the truth table of any given function in the space, and thereby demonstrates the completeness of this set of two functions. His work is of fundamental importance as after Post, the task of proving completeness of a set of functions is simplified to that of demonstrating that Post's primitives are definable in the set.

Continuing the study in the specific case of *E(2)*, Zylinski (1925), in 1925, by actually considering all 16 two-place functions in the space, shows that *D* and *X* (nand and nor) are the only Sheffer functions.

The first identification of a Sheffer function over *E(k)* is by Webb (1935) in 1935 and later modified in 1936 (1936). In the latter paper, Webb's function is given as

$$h(x,y) = (max(x,y) - 1)_m$$

(as in the discussion of Post's result, the above is a translation of the actual function that is given in terms of the subscripts on the truth values). Webb proves completeness by showing that Post's primitives are definable in terms of his function. Using the notation $g^2(x) = g(g(x))$, $g^3(x) = g(g^2(x))$, etc., Post's primitives are defined in terms of Webb's function as

$$g(x) = h(x,x)$$

$$max(x,y) = g^{k-1}h(x,y).$$

Słupecki (1939), in 1939, proves that if a set of functions over *E(k)* is able to generate all the one-place functions, and if one of the functions is a binary function whose truth table contains at least one row and one column in which not all elements are identical (non-degenerately binary) and in which all *k* truth values appear at least once, then the set of functions is complete.[1] Słupecki's result allows other investigators to prove completeness of a set of functions that include a binary function of the form indicated, by demonstrating the ability of the set to generate all the one-place functions.

N. M. Martin (1950), in 1950, defines a number of Sheffer functions over *E(k)*, including

$$f_{1,1,1}(p,q) = \begin{cases} \min(p,q) + 1 \\ 1, \; p = q = n \end{cases}$$

In a review of the above paper, Rose (1951) suggests a necessary and sufficient condition for a Sheffer function over *E(k)*. His condition is an extension of the result of Kalicki (1950), in 1950, which is here described first.

Kalicki introduces the idea of classes of value sequences, wherein value sequences generated by a single function $f: E^2(k) \rightarrow E(k)$ are classified as follows.

---

[1] Słupecki's paper had been printed and readied for distribution shortly before Hitler's invasion of Warsaw in 1939. The building in which the papers were stored was destroyed during the invasion, so that only three or four copies of the original paper are in existence today.

1. The first class CL_1 of value sequences consists of the single element x.
2. The $(t + 1)$th class CL_{t+1} consists of all value sequences generated by f from functions, one of which has a value sequence in CL_t and the other a value sequence in CL_n $(n \leq t)$.

   That is,

   $CL_1 = \{x\}$,

   $CL_2 = \{f(x,x)\}$,

   $CL_3 = \{f(x,f(x,x)),\ f(f(x,x),x),\ f(f(x,x),f(x,x))\}$,

   etc.

The set of value sequences of the elements in CL_n is denoted by $|CL_n|$ and let

$$|DL_n| = \bigcup_{i=1}^{n} |CL_n|$$

A value sequence is not included in $|CL_i|$ if it is already contained in $|DL_{i-1}|$, and, therefore if $|CL_i|$ is empty then so is $|CL_j|$ for all $j \geq i$. Since x is the only variable appearing as an argument, and x takes values from $E(k)$, the value sequences consist of k digits, and there are at most k^k different sequences. Therefore, there is a finite $k_0 \leq k^k$, such that all the value sequences generated by f are contained in $|DL_{k_0}|$.

Rose's technique is to modify the procedure of Kalicki so that the classes are defined as

$CL_1 = \{x, y\}$,

$CL_2 = \{f(x,x), f(x,y), f(y,x), f(y,y)\}$,

etc.

Now the value sequence consists of k^2 digits and there is a finite $k_0 \leq k^{k^2}$ such that all the value sequences generated by $f(x,y)$ will be contained in $|DL_{k_0}|$. Therefore, a k-valued function is a Sheffer function if and only if the value sequence of Martin's $f_{1,1,1}(x,y)$ appears in $|DL_{k_0}|$.

Using Rose's approach, the number of classes that may have to be constructed is $k_0 = k^{k^2} - 1$, which even for small values of k is an extremely large number. Fortunately, the technique can be improved using the results of Martin, to be described shortly.

The important works which follow concern themselves with the formulation of necessary and sufficient conditions for completeness.

In 1952, J.D. Swift (1952) presents a number of necessary conditions for a two-place function $f(x,y)$ defined over $E(k)$ to be a Sheffer function, among which are:

1. $f(x,y)$ must have $k!$ distinct conjugates (fully conjugated).
2. $f(x,y)$ cannot be both commutative and associative.
3. $f(i,i) \neq i$, $(i = 0, 1, 2, \ldots k-1)$.

4. No value can be "trapped" within a set of values. For example, at least one of the values of $f(a,a)$, $f(a,b)$, $f(b,a)$, $f(b,b)$ must be a value other than a or b.

The latter two conditions can be combined into a single condition: A Sheffer function cannot be closed on any non-empty, proper subset of the k truth values. (The property of being closed an a non-empty, proper subset of the k truth values has been called *properly closing* by other authors.)

Applying these conditions, Swift is able to identify all 90 symmetric two-place functions over $E(3)$.

In 1954, N. M. Martin (1954), concerned only with two-place functions, proves that in addition to not being properly closing, there are three other conditions necessary for a function to be Sheffer over $E(k)$. In order to state Martin's conditions a number of terms must first be defined.

DEFINITION 3: A one-place function is a *t-function* if $t^k(j) = j$ for all $j \in E(k)$, and $t^i(j) \neq j$ for every i $(1 \leq i \leq k-1)$, and $j \in E(k)$.

DEFINITION 4: A function $f(x,y)$ is *t-closing* if there is a t-function $t(p)$ such that for every i and j, there is an m such that $f(t^i(p),t^j(p)) = t^m(p)$.

DEFINITION 5: If the truth values in $E(k)$ are partitioned into two or more disjoint, non-empty classes and i and j are elements of the same class then one writes

$i \sim j(D)$. A two-place function satisfies the *substitution law* for a partition D if for any truth values $h, i, j, k,$ whenever $h \sim j(D)$ and $i \sim k(D)$, then $f(h,i) \sim f(j,k)(D)$.

DEFINITION 6: If there is a partition D of the truth values into less than k classes, for which $f(x,y)$ satisfies the substitution law, then $f(x,y)$ has the *proper substitution property*.

DEFINITION 7: A function $f(x,y)$ satisfies the *co-substitution law* for D if for any truth values $h, i, j, k,$ whenever $f(h,i) \sim f(j,k)(D)$, then $h \sim j(D)$, or $i \sim k(D)$.

DEFINITION 8: If there is a partition D such that $f(x,y)$ satisfies the co-substitution law, then $f(x,y)$ has the *co-substitution property*.

Martin proves that a necessary condition for a function to be a Sheffer function is that it has none of the properties of proper substitution, co-substitution, t-closing or properly closing. Then, using Slupecki's result, he shows that a two-place function is a Sheffer function if and only if there is an $i > 0$ such that all i-place functions can be defined by a finite iteration of $f(x,y)$. Applying this result to the case $i = 1$, he proves that a two-place function over $E(k)$, $(k > 2)$, is a Sheffer function if and only if by repeated composition it can generate all one-place functions.

Using a result previously obtained by S. Piccard (1935), who defined three one-place functions over *E(k)* sufficient for generating all one-place functions, Martin then refines his criterion so that a function is a Sheffer function if and only if it can generate the three one-place functions of Piccard (actually Martin's set of three functions is a slight modification of Piccard's). In *E(3)*, this criterion allows Martin to demonstrate that a necessary and sufficient condition for a function to be a Sheffer function is that it does not possess any of the properties of proper substitution, co-substitution, t-closing or properly closing. Finally, applying this condition, Martin specifies that there are 3,774 two-place Sheffer functions over *E(3)*.

Foxley (1962), in 1962, translates Martin's conditions into logical expressions describing the conditions that must be satisfied by the nine entries in the operation table of a two-place Sheffer function over *E(3)*. After programming these conditions on a "logical computer" and entering all possible truth value assignments, he determines that there are 3,774 two-place functions over *E(3)*, in agreement with Martin. Upon relaxing the condition of not possessing the co-substitution property, he finds no change in the number of functions, indicating that this property is not independent of the other three. He then proves that in *E(3)*, if *f(x,y)* has none of the properties of proper closure, t-closure or proper substitution, then it cannot have the co-substitution property.

Among the consequences of this paper is that after Martin's result, to prove a function Sheffer over *E(k)* it is sufficient to show that the function generates the three one-place functions he specified. This result, used in conjunction with Rose's method for proving a function Sheffer, essentially reverts to Kalicki's original method, where only value sequences of a one-place function are generated. The classes CL_i are constructed beginning with $CL_1 = \{x\}$. If the value sequences of Martin's three one-place functions are contained in any $|DL_i|$ then the function is Sheffer. The upper bound on the calculation is now CL_r, with $r = k^k$, so that the method is a lot more practical than Rose's original. (J.C. Muzio (1970), in 1970, finds that for two place functions over *E(3)* the actual bound is CL_6.)

A. Salomaa (1960), in 1960, proves that any function $f(x_1,x_2,\ldots,x_k)$ that generates all one-place functions is a Sheffer function. He is able to refine the criteria of Martin to show that for $k \geq 3$ any two-place function is a Sheffer function if and only if it generates two one-place functions, the permutations $s_1(x)$ and $s_2(x)$ that form the basis of the symmetric group S_k.

Wheeler (1966), in 1964, is able to formulate a set of necessary and sufficient conditions for an *n*-place function over *E(3)* to be a Sheffer function. He defines a δ-function as a function satisfying two conditions that are equivalent to Martin's conditions of not being properly closing and not satisfying the substitution property. He proves that for $n \geq 2$, an *n*-place function is a Sheffer function over *E(k)* if it is a fully conjugated

δ-function. Using these conditions he is able to enumerate the number of *n*-place Sheffer functions over *E(3)*, and shows that the number tends asymptotically to $\frac{8}{27}$.

The classic paper providing necessary and sufficient conditions for a set of functions over *E(k)* to be complete is that of Rosenberg (1965), in 1965. He proves a theorem that establishes a set of eight conditions required of a complete set of functions. The conditions are relatively complex, requiring a large number of preliminary definitions. Inasmuch as the results do not bear directly upon the concerns of this paper, it would be impractical to say any more than that all further work in the area of necessary and sufficient conditions after 1965 are based upon Rosenberg's results.

For example, Rousseau (1967), in 1967, obtains four necessary and sufficient conditions for a function over *E(k)* to be a Sheffer function by modifying Rosenberg's general results, and Schofield (1969), in 1969, reduces Rousseau's conditions to three, which are effectively the fully conjugated condition of Wheeler and the absence of the proper closure property and the proper substitution property of Martin.

In 1975, Muzio (1975a) studies two-place functions over *E(3)* and, proceeding from Rosenberg's theorem, deduces a set of necessary and sufficient conditions for a function to be a Sheffer with constants function. From these conditions, he calculates that of the 19,683 two-place functions over *E(3)*, 15,201 are Sheffer with constants. Ignoring those two-place

functions that are equal to one-place functions, those whose range is not *E(3)* and those that are constants, this number represents 83% of all two-place functions. Muzio conjectures that as $k \to \infty$, the proportion of two-place functions that are complete with constants approaches 100%.

Finally, also in 1975, Muzio (1975b) publishes a paper with which this dissertation is most directly concerned. Muzio is interested in the two-place Sheffer functions over *E(3)*, and by translating the completeness conditions of Martin into conditions on the nine elements in the operation table, he is able to classify all Sheffer functions as belonging to one of two types. Specifically, since a function cannot be closed on a proper subset of the truth values, a function cannot have a fixed point, that is, $f(i,i) \neq i$, $(i = 0, 1, 2)$; therefore, the diagonal of a Sheffer function must be of one of the following forms (using the notation $[f(0,0), f(1,1), f(2,2)]$):

[1 0 0], [1 0 1], [1 2 1], [2 0 0], [2 2 0], [2 2 1],

[1 2 0], [2 0 1].

Letting *a, b, c* represent any permutation of the values 0, 1, 2 and using the notation *[f(a,a), f(b,b), f(c,c)]*, the first set of diagonals above can all be represented by [*b a a*], and the second set by [*b c a*]. Muzio's Type I consists of all functions whose diagonal is of the form [*b a a*], and Type II consists of all functions whose diagonal is of the form [*b c a*]. More detailed information concerning these classes is given in Chapter 2.

CHAPTER 2

PROBLEM DESCRIPTION

2. The Case of *E(2)*

The problem considered here is most clearly described using the case of *E(2)* as an illustration because in *E(2)*, with only 16 two-place functions, the concepts are easily presented, some of the results are well-known, and the relatively small number of computations required allows for a solution without the aid of the computer.

It is well-known that in *E(2)* the functions commonly identified as nand and nor, and here identified as *D* and *X*, respectively, are the only Sheffer functions. Additionally, four other functions, *B*, *C*, *L* and *M*, although not Sheffer functions, are Sheffer with constants functions. Through the operation of composition, and with the availability of the constant functions, the Sheffer functions and the Sheffer with constants functions are able to generate all of the sixteen two-place functions in *E(2)*. Analogously, this fact can be stated in terms of elements of a logic circuit as follows.

The fundamental element in a combinational switching circuit is a gate. The gate may have many inputs, but a single output. Since the concern here is with two-place functions over *E(2)*, the analog is a gate with

two inputs, x and y, taking values from $E(2)$ and whose single output is also a value in $E(2)$. A gate is said to realize a function f if it transforms its two inputs into the single output $f(x,y)$. A collection of gates realizing different functions f_i may be combined into a combinational switching circuit by connecting the outputs of certain gates to the inputs of others so that the resulting configuration has a single output, with no feedback allowed. (The reordering and identification of inputs is allowed, meaning that if a gate realizing $f(x,y)$ is available, so are gates realizing $g(x,y) = f(y,x)$, and $h(x) = f(x,x)$.) The single output of the combinational circuit is a realization of some unique function f of the primary inputs and is equivalent to the composition of the f_i used in its construction.

As an illustration, the switching circuit below is a realization of the function $f(x,y) = D(D(x,x),B(x,y))$.

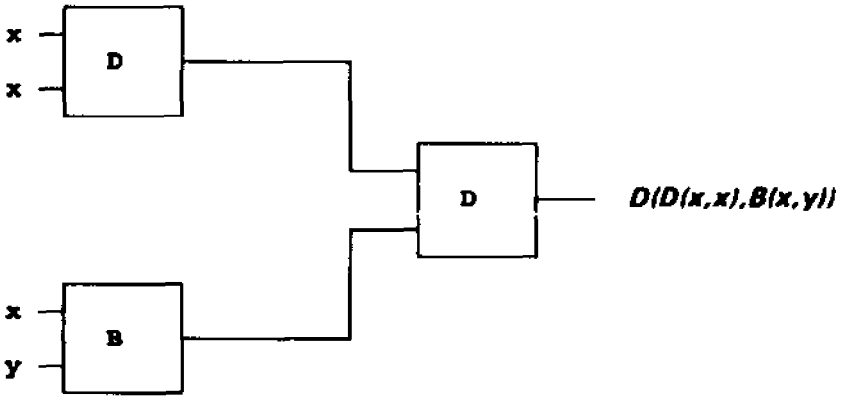


Figure 2. A hardware analog of composition

Now, if a gate realizing a Sheffer function is available, then every function has a hardware realization consisting of some combinational circuit

constructed exclusively from copies of this gate as the circuit elements. Similarly, if a gate implementing a Sheffer with constants function is available, and it is assumed that the constant functions are also available in the form of high and low voltage levels, then every function has a hardware realization as a combinational circuit constructed exclusively from copies of this gate as the circuit elements.

Since any one of the six functions, *D, X, B, C, L, M* is capable of generating all other functions, the question arises as to whether one of them is more efficient than another in performing this task. In order to answer this question, some measure of efficiency must be devised. Towards this end, an illustration using the function *D* follows, where a composition sequence such as *D(1, D(D(x,x), D(y,y)))* is written in a non-parenthesized form as *D1DxxDyy*, and the terms function and gate are used synonymously.

It is assumed that in every circuit the primary inputs $I(x,y) = x$ and $H(x,y) = y$, and the constant functions, $O(x,y) = 0$ and $V(x,y) = 1$ are available *free*, that is without the use of any function. These four functions are then available as the arguments of *D*. With a single application of *D* the following functions can be formed: *Dxx, Dxy, DxO, Dx1, Dyx Dyy, DyO, Dy1, DOx, DOy, DOO, DO1 D1x, D1y, D1O, and D11*. Of course, some of the value sequences of these functions duplicate others of the set, and some of them duplicate the sequences of the four free functions. When all

duplications are eliminated, three unique sequences remain. The functions are:

$$Dxx = \langle 1\ 1\ 0\ 0 \rangle = F = \bar{x}$$

$$Dxy = \langle 1\ 1\ 1\ 0 \rangle = D$$

$$Dyy = \langle 1\ 0\ 1\ 0 \rangle = G = \bar{y}$$

(While there are only three different value sequences, the choice of which three functions to retain is arbitrary. In this example *Dxx* = *Dx1* = *D1x*, and *Dyy* = *Dy1* = *D1y*; consequently, *Dxx* could be replaced above by either *Dx1* or *D1x*, and *Dyy* could be replaced by either *Dy1* or *D1y*.)

With these new functions now available as arguments of the function *D*, the following functions, all requiring two uses of function *D*, may be formed: *DxDxx, DxDxy, DxDyy, DyDxx, DyDxy, DyDyy,* etc. With duplications removed as before, the following functions remain (again arbitrary selections from duplicate values have been made):

$$D1Dxy = \langle 0\ 0\ 0\ 1 \rangle = K$$

$$DyDxx = \langle 1\ 0\ 1\ 1 \rangle = B$$

$$DxDxy = \langle 1\ 1\ 0\ 1 \rangle = C$$

This procedure of using previously generated functions as arguments of *D* to form new functions is continued until all sixteen two-place functions have been generated. The process is summarized as follows.

Zero applications of D (free functions)

$<0\ 0\ 0\ 0> = O = 0$
$<1\ 1\ 1\ 1> = V = 1$
$<0\ 1\ 0\ 1> = H = y$
$<0\ 0\ 1\ 1> = I = x$

One application of D

$Dxx = <1\ 1\ 0\ 0> = F = \overline{x}$
$Dxy = <1\ 1\ 1\ 0> = D$
$Dyy = <1\ 0\ 1\ 0> = G = \overline{y}$

Two applications of D

$D1Dxy = <0\ 0\ 0\ 1> = K$
$DyDxx = <1\ 0\ 1\ 1> = B$
$DxDxy = <1\ 1\ 0\ 1> = C$

Three applications of D

$D1DxDxy = <0\ 0\ 1\ 0> = L$
$D1DyDxx = <0\ 1\ 0\ 0> = M$
$DDxxDyy = <0\ 1\ 1\ 1> = A$

Four applications of D

$D1DDxxDyy = <1\ 0\ 0\ 0> = X$

Five applications of D

$DDxDxyDyDxx = <0\ 1\ 1\ 0> = J$
$DDxyDDxxDyy = <1\ 0\ 0\ 1> = E$

It is helpful to portray the situation of generating functions by the repeated application of a particular function f as follows. A_0 is the set of functions formed without the use of f (this set always consists of the constant functions and the inputs x and y). A_1 is the set of functions formed with one or fewer uses of f. A_2 is the set of functions formed with

two or fewer uses of *f*, etc. In general, A_i is the set of all functions that may be defined with *i* or fewer uses of *f*. Eventually, if the function is Sheffer with constants, there will be some smallest value of *r* such that A_r contains all of the functions in *E(2)*. Visualizing the sets A_i as a series of concentric circles (the set of circles for the function *D* is shown in the figure on the following page) in the space of two-place functions, the minimal index *r* such that A_r contains every two-place function over *E(2)* is termed the radius of the function, *r(f)*. The radius of a function is proposed as the measure of its efficiency in generating all other functions in the space.

The radius of each of the Sheffer functions and each of the Sheffer with constants functions in *E(2)* is given below, along with one function whose generation requires the largest number of uses of the function.

function	radius	worst case function
B	5	*E = BOBBOByxBxy*
C	5	*E = CCCxyCCyx00*
D	5	*E = DDxyDDxxDyy*
L	5	*J = L1LL1LyxLxy*
M	5	*J = MMMxyMMyx11*
X	5	*E = XXxXxyXyXxx*

Figure 3. Radii of Sheffer with constants functions over *E(2)*

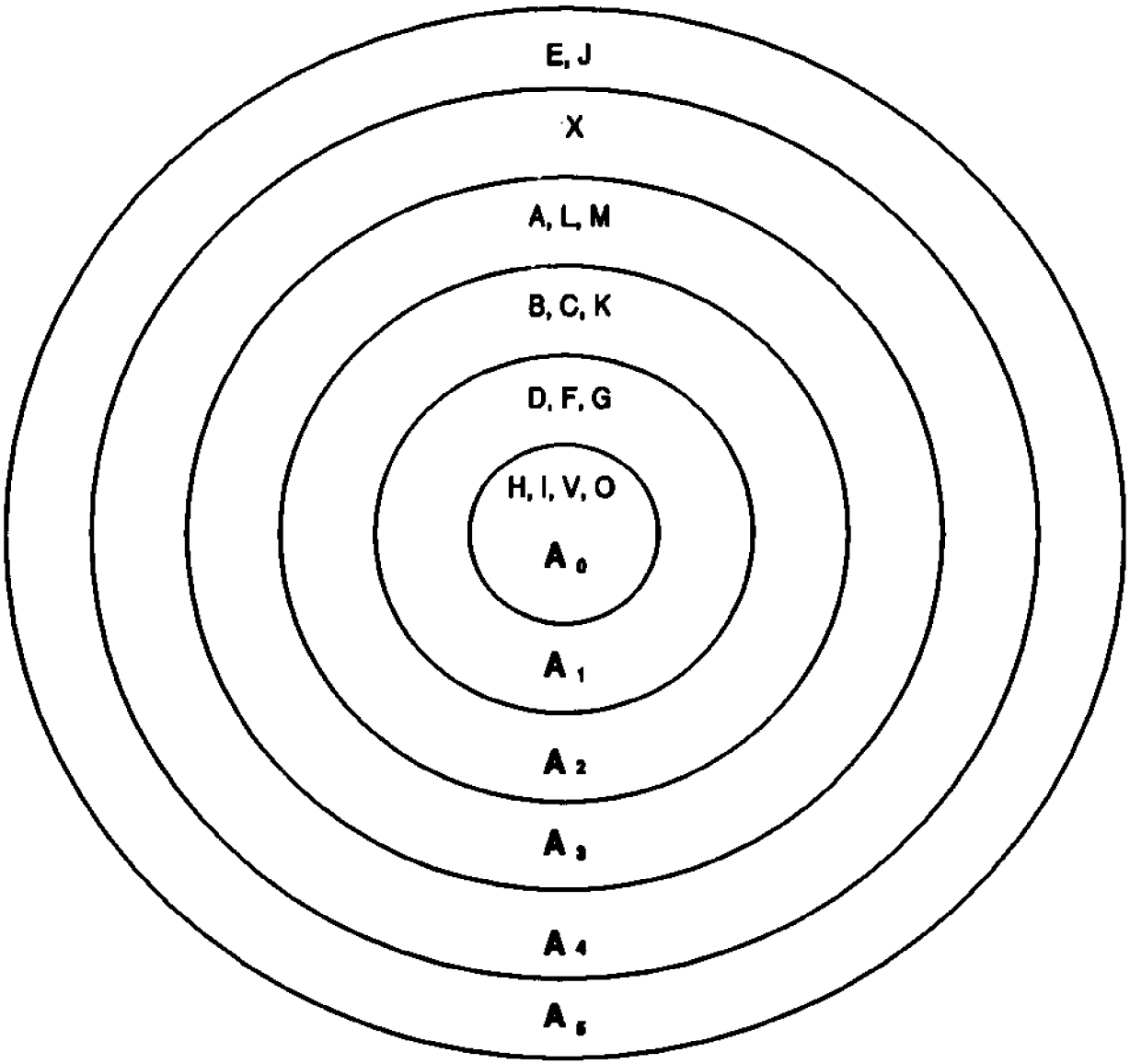


Figure 4. The set of A_i for the function D(x,y)

Clearly, in *E(2)* all of the functions have the same value of the radius, and, therefore, are equally efficient. To the circuit designer, there is no preference as to which should be implemented as the fundamental gate from which all logic circuits may be constructed.

The problem of interest here is to perform a similar analysis of the Sheffer functions over *E(3)*.

CHAPTER 3

COMPUTATIONAL ISSUES

3.1 Introduction

In order to characterize their efficiency, it is necessary to determine the radius of every one of the 3,774 two-place Sheffer functions over *E(3)*. Before establishing the results to be described here, there was no suggestion whatsoever as to what the typical size of such radii might be. As demonstrated in Chapter 2, the radius of each Sheffer functions over *E(2)* is known to be five, but in *E(2)* there are only 16 two-place functions. In *E(3)* there are 19,683 functions, and, therefore, it was believed conceivable that some radii could be of that order of magnitude. Performing 3,774 calculations, each ultimately requiring the generation of 19,683 different functions seems a daunting task, with enormous time requirements, perhaps a practical impossibility. Fortunately, further investigation reveals that the necessary information can be obtained with significantly fewer calculations than originally supposed, thereby making feasible the attainment of the stated goal. The following is a description of the simplification of the problem.

In *E(3)*, the operation table of a two-place function is a square table with three rows and three columns. A typical table is shown on the next page.

		y		
	f(x,y	0	1	2
	0	1	1	2
x	1	2	0	0
	2	2	0	0

There are two manipulations which can be performed on the operation table of a function which generates new functions with important connections to the original function.

3.2 The Conjugate

When a permutation P is applied to the elements {0,1,2} of $E(3)$, the operation table of the function $f(x,y)$ is transformed into the table of another function symbolized by $f_P(x,y)$ which is referred to as the conjugate of $f(x,y)$ under the permutation P. Letting P^{-1} represent the inverse permutation of P, the relationships between the function and its conjugate which will be used subsequently, are:

$$f_P(x,y) = P(f(P^{-1}(x), P^{-1}(y)) \quad (1)$$

$$P^{-1}(f_P(x,y)) = f(P^{-1}(x), P^{-1}(y)) \quad (2)$$

$$f(x,y) = P^{-1}(f_P(P(x), P(y))) \quad (3)$$

$$P(f(x,y)) = f_P(P(x), P(y)) \quad (4)$$

When the rows and columns of the operation table are interchanged, the new table defines another function $\bar{T}(x,y)$, which is referred to as the transpose of $f(x,y)$. The relationship between a function and its transpose is:

$$\bar{T}(x,y) = f(y,x) \quad (5)$$

Shown below are the operation tables of a function, its conjugate under the permutation $P = (01)$, and its transpose.

1	1	2
2	0	0
1	0	0

f

1	2	1
0	0	2
1	0	1

f_P

1	2	1
1	0	0
2	0	0

$\bar{f}$

In the preceding illustration, the operation table of the function is different from that of its conjugate. It is a simple matter to construct a function whose operation table is identical with that of its conjugate as illustrated below, again using $P = (01)$.

0	2	1
2	1	0
0	1	2

If the operation table of a function $f(x,y)$ is identical with the table of its conjugate under some permutation P, the function is said to be self-conjugate under the permutation P.

We focus now on some interesting relationships among Sheffer functions and their conjugates, and return to the question of the transpose shortly. First, it is convenient to introduce an abbreviated notation for composition sequences.

A well-formed composition sequence, in parenthesized format, using a two-place function f is an expression that begins with f and is followed by a

set of parentheses enclosing a pair of arguments separated by a comma. Each of the arguments is either x or y, or is itself a well-formed composition sequence. For example: $f(f(x,y), f(f(y,x),x))$ is a well-formed sequence, while $f(f(x,y), f(y,x), x)$ is not.

The grammar of composition sequences using a function f is described by the productions

$$S \rightarrow f(A,A)$$

$$A \rightarrow x \mid y \mid S$$

If the definition of a composition sequence is expanded to include x and y as legal sequences requiring no uses of the function f, then the grammar is simplified to

$$S \rightarrow x \mid y \mid f(S,S)$$

Let $S^n[f(x,y)]$ be introduced as representing some well-formed composition sequence with n usages of the function f, referred to as an n^{th} order composition sequence. Then

$S^0[f(x,y)] = x$ or y,

$S^1[f(x,y)] = f(S^0[f(x,y)], S^0[f(x,y)]) = f(x,y)$ or $f(x,x)$ or $f(y,x)$ or $f(y,y)$,

$S^2[f(x,y)] = f(S^0[f(x,y)], S^1[f(x,y)])$, or

$S^2[f(x,y)] = f(S^1[f(x,y)], S^0[f(x,y)])$,

$\vdots$

$S^n[f(x,y)] = f (S^r[f(x,y)], S^s[f(x,y)])$,

where $r + s = n - 1$ and $0 \leq r,s \leq n - 1$.

As an example,

$S^4[f(x,y)] = f(\, f(x,y),\, f(f(y,x),x)\,) = f(\, S^1[f(x,y)],\, S^2[f(x,y)]\,)$, with

$S^1[f(x,y)] = f(x,y)$ and $S^2[f(x,y)] = f(f(y,x),x)$.

Clearly, when $r \neq s$, $S^r[f(x,y)]$ and $S^s[f(x,y)]$ represent different composition sequences; however, when $r = s$, $S^r[f(x,y)]$ and $S^s[f(x,y)]$ may or may not be the same sequence. Where it is necessary to refer to a specific n^{th} order composition sequence other alphabetic characters are used in place of S. For example, if $R^3[f(x,y)] = f(\, f(x,y),\, f(x,x)\,)$, then $R^3[g(x,y)] = g(\, g(x,y),\, g(x,x)\,)$.

Additionally, let $R^n[f(y,x)]$ represent the composition sequence formed from $R^n[f(x,y)]$ by reversing the two arguments at every usage of f in the original sequence. As an illustration,

$$R^4[f(x,y)] = f(\, f(x,f(x,y)),\, f(y,x)\,)$$

$$R^4[f(y,x)] = f(\, f(x,y),\, f(f(y,x),x)\,)$$

Finally, if a function h is generated by some n^{th} order composition sequence using function f then we write $h(x,y) = S^n[f(x,y)]$

The following Lemma stated without proof by Salomaa (1960), is required in the proofs of some of the theorems which follow.

LEMMA 1: *If g, f, r, and s are functions satisfying*

$g(x,y) = f(\, r(x,y),\, s(x,y)\,)$ *then,*

$g_p(x,y) = f_p(\, r_p(x,y),\, s_p(x,y)\,)$ *for any permutation P.*

Proof: Given $g(x,y) = f\,(\,r(x,y),\, s(x,y)\,)$

$$g_P(x,y) = P(\,g(P^{-1}(x),\, P^{-1}(y)\,)$$

$$= P(\,f(\,r(P^{-1}(x),\, P^{-1}(y)),\, s(P^{-1}(x),\, P^{-1}(y))\,)\,) \text{ by (1).}$$

$$g_P(x,y) = P(\,f(\,P^{-1}r_P(x,y),\, P^{-1}s_P(x,y)\,) \quad \text{by (2)}$$

$$g_P(x,y) = P(\,P^{-1}f_P(r_P(x,y),\, s_P(x,y)\,) \quad \text{by (2)}$$

$g_P(x,y) = f_P(\,r_P(x,y),\, s_P(x,y)\,)$, as PP^{-1} is equivalent to the identity permutation.

The next theorem demonstrates that if a composition sequence using f generates a function h, then replacing every occurrence of f by f_P results in a sequence which generates h_P.

THEOREM 1: *If* $h(x,y) = R^n[f(x,y)]$, *then* $h_P(x,y) = R^n[f_P(x,y)]$, *for* $n \geq 1$.

Proof: The proof is by induction on the number of uses of f in the composition sequence.

For the basis with $n = 1$: $h(x,y) = R^1[f(x,y)] = f(x,y)$ or $f(x,x)$ or $f(y,x)$ or $f(y,y)$. Select $h(x,y) = f(x,y)$. Then $h_P(x,y) = P(h(\,P^{-1}(x),\, P^{-1}(y)\,) = P(f(\,P^{-1}(x),\, P^{-1}(y)\,)$ by (1), which is equal to $f_P(x,y)$ by (2). Finally, $f_P(x,y) = R^1[f_P(x,y)]$, so $h_P(x,y) = R^1[f_P(x,y)]$. Similarly, for the other three cases.

Assuming the hypothesis is true for all integers $1 \leq j \leq n$, we prove the result for $n + 1$. Suppose $h(x,y) = R^{n+1}[f(x,y)]$, then $h(x,y) = f(\,R^r[f(x,y)],\, R^s[f(x,y)]\,)$, with $r + s = n$, and $0 \leq r,s \leq n$. Let $R^r[f(x,y)] = r(x,y)$ and $R^s[f(x,y)]) = s(x,y)$, so that $h(x,y) = f(\,r(x,y),\, s(x,y)\,)$, and $h_P(x,y) = f_P(\,r_P(x,y),\, s_P(x,y)\,)$ by Lemma 1,

and $h_P(x,y) = f_P(R^r[f_P(x,y)], R^s[f_P(x,y)])$ by the hypothesis. Finally $h_P(x,y) = R^{n+1}[f_P(x,y)]$ using the properties of the value sequence, thus proving the theorem.

COROLLARY 1: *If a function is self-conjugate under a permutation P, then every function generated by this function through the operation of composition will also be self-conjugate under the permutation P.*

Proof: Assume $f(x,y)$ is self-conjugate under a permutation P, and $h(x,y)$ is a function generated by some n^{th} order composition sequence using f, i.e., $h(x,y) = R^n[f(x,y)]$. Then $h_P(x,y) = R^n[f_P(x,y)] = R^n[f(x,y)]$ by Theorem 1 and the fact that $f(x,y)$ is self-conjugate under P. But $h(x,y) = R^n[f(x,y)]$, so $h_P(x,y) = h(x,y)$ and h is proved self-conjugate under the permutation P.

Corollary 1 states that a function which is self-conjugate under a permutation P can never generate a function that it not self-conjugate under a permutation P. Since it is easily demonstrated that for any permutation P there exist functions that are not self-conjugate under P, the important ramification is that a Sheffer function cannot be self-conjugate under any permutation P, and, therefore, in $E(k)$, a Sheffer function has $n!$ distinct conjugates.

THEOREM 2: *All conjugates of a Sheffer function are themselves Sheffer functions.*

Proof: It will be demonstrated that if $f(x,y)$ is a Sheffer function, so is $f_P(x,y)$ for any permutation P.

Since $f(x,y)$ is a Sheffer function, for any function $h(x,y)$ a composition sequence using f exists for the function $h_{P^{-1}}(x,y)$. Assume that this sequence is an n^{th} order sequence, that is $h_{P^{-1}}(x,y) = R^n[f(x,y)]$. Now, if in this composition sequence every appearance of the function f is replaced by f_P, by Theorem 1, the resulting sequence, $R^n[f_P(x,y)]$, generates the conjugate under P of the function originally generated, which in this instance is, $(h_{P^{-1}})_P(x,y) = h(x,y)$ Therefore, $h(x,y) = R^n[f_P(x,y)]$. With $h(x,y)$ being an arbitrary function, this establishes that any function in the space can be generated by a composition sequence using only $f_P(x,y)$, proving that $f_P(x,y)$ is a Sheffer function.

COROLLARY 2: *If $f(x,y)$ is a Sheffer function of radius r, then all of its conjugates are also of radius r.*

Proof: Each function in the space $E(k)$ (19,683 in $E(3)$) is generated by a unique composition sequence with at most r uses of f. If in each of these composition sequences every appearance of f is replaced by f_P, then, by Theorem 2, the new composition sequence generates the conjugate under permutation P of the originally generated function. Since distinct functions have distinct conjugates under the same permutation, there are as many distinct functions generated by the new composition sequences as there are by the original ones. Hence, every function in the space has a unique composition sequence with at most r uses of f_P, establishing the result that every f_P is also of radius r.

3.3 The transpose

If the operation table of a function is different from that of its transpose, the function is called non-symmetric; otherwise, it is called symmetric. Clearly, from (5), a symmetric function has an operation table that is symmetric about the diagonal.

The next Lemma demonstrates that replacing every occurrence of f by $\bar{f}$ in a composition sequence, and reversing the arguments, results in a sequence which generates the same function as the original.

LEMMA 2: If $h(x,y) = R^n[f(x,y)]$, then $h(x,y) = R^n[\bar{f}(y,x)]$ for $n \geq 1$.

Proof. By induction on the number of appearances of f in the sequence.

For the basis with $n = 1$, $h(x,y)$ must be equal to $f(x,y)$ or $f(x,x)$ or $f(y,x)$ or $f(y,y)$. Select $h(x,y) = f(x,y)$. But $f(x,y) = \bar{f}(y,x)$ by the definition of the transpose, and, therefore, $h(x,y) = \bar{f}(y,x) = R^1[\bar{f}(y,x)]$.

Assuming the hypothesis is true for all integers $1 \leq j \leq n$, we prove the result for $n + 1$. Now $h(x,y) = R^{n+1}[f(x,y)] = f(R^r[f(x,y)], R^s[f(x,y)])$, with $r + s = n - 1$, and $0 \leq r,s \leq n - 1$, by the given information and the definition of an n^{th} order composition sequence. But $f(R^r[f(x,y)], R^s[f(x,y)]) = f(R^r[\bar{f}(y,x)], R^s[\bar{f}(y,x)])$, by the hypothesis, and this in turn is equal to $\bar{f}(R^s[\bar{f}(y,x)], R^r[\bar{f}(y,x)])$, by the definition of the transpose. This last expression is replaced by $R^{n+1}[\bar{f}(y,x)]$ by the definition of the notation for composition sequences, showing that $h(x,y) = R^{n+1}[\bar{f}(y,x)]$ and the result is proved.

THEOREM 3: *If f(x,y) is a Sheffer function of radius r, then its transpose* $\overline{f}(x,y)$ *is also a Sheffer function of radius r.*

Proof: Suppose *f(x,y)* is a Sheffer function of radius *r*. Then for any function *h(x,y)* there is some composition sequence using at most *r* occurrences of *f* that generates *h*. In this composition sequence, replacing every occurrence of *f* by $\overline{f}$ and reversing the arguments yields a composition sequence with at most *r* uses of $\overline{f}$, which, by Lemma 2, also generates *h*. Hence $\overline{f}(x,y)$ is a Sheffer function of radius *r*.

Combining Lemma 2 and Theorem 3, we see that the six conjugates and the transpose of a Sheffer function of radius *r* are themselves Sheffer functions of radius *r*. The important remaining question is, can the transpose of a Sheffer function be equal to one of its conjugates? Symbolically, can the following equation be satisfied for some permutation *P*?

$$\overline{f}(x,y) = f_P(x,y) \qquad (7)$$

or, equivalently,

$$f(y,x) = P(f(P^{-1}(x),P^{-1}(y))) \quad (8)$$

Obviously, the equation is trivially true for any symmetric function if *P* is taken to be the identity permutation. For a non-symmetric Sheffer function in *E(3)*, the following theorem, stated and proved by Salomaa, shows that equation (7) cannot be satisfied.

THEOREM 4. *For each non-symmetric Sheffer function in E(3), the transpose and the conjugates over all permutations P are distinct.*

Proof: Suppose equation (7) is true for some non-symmetric function under some permutation P. Since the diagonal elements $f(i,i)$, $(i = 0, 1, 2)$, are unchanged when forming the transpose, in order for the theorem to hold, they must also be unchanged when forming the conjugate. Additionally, one of the requirements of a Sheffer function is that $f(a,a) \neq a$ for any value of a. The two foregoing conditions are satisfied only if the diagonal elements comprise one of the value sequences $<1\ 2\ 0>$ or $<2\ 0\ 1>$ and P is one of the cyclic permutations (012) or (201). Selecting $P = (012)$ and, therefore, $P^{-1} = (021)$, equation (8) implies the following relations.

$$
\begin{aligned}
f(0,1) &= P(f(0,2)) \\
f(0,2) &= P(f(1,2)) \\
f(1,0) &= P(f(2.0)) \\
f(1,2) &= P(f(1,0)) \\
f(2,0) &= P(f(2,1)) \\
f(2,1) &= P(f(0,1))
\end{aligned}
$$

Starting with the first relation, and recognizing that P^3 is the identity permutation, leads to: $f(0,1) = P(f(0,2)) = P(P(f(1,2))) = P^2(1,2) = P^2(P(f(1,0))) = P^3(f(1,0)) = f(1,0)$.

Similarly, it is easily shown the above also lead to $f(0,2) = f(2,0)$ and $f(1,2) = f(2,1)$. The three conditions require that the function f be symmetric, in contradiction with the assumption, thereby demonstrating the result. A similar proof is used in the case of $P = (201)$.

Taken together, Lemma 2, Theorem 3 and Theorem 4 imply that in *E(3)* the Sheffer functions can be partitioned into a set of mutually exclusive classes, to be referred to as isotopic classes, each comprised of the

conjugates and transposes of the conjugates of a function. Classes with non-symmetric functions consist of 12 members, and those with symmetric functions consist of six members. All members of a class have the same radius.

Based upon the foregoing, in order to calculate the radii of all 3,774 Sheffer functions in *E(3)*, it is only necessary to determine the radius of a representative from each of the isotopic classes.

CHAPTER 4

CALCULATIONS

4.1 Generating the isotopic classes

4.1.1 Type I

Muzio (1975b) presents the Sheffer functions in *E(3)* by dividing them into two classes, designated I and II, distinguished by the values along the diagonal. The Type I functions are partitioned into four classes, and Muzio uses his theorem characterizing Sheffer functions in *E(3)* to designate certain elements in the operation table in each class. The operation table of each class, as presented by Muzio, is reproduced below, with *a*, *b*, *c* representing any permutation of the values 0, 1, 2, and with dashes signifying positions whose values are arbitrary. It is assumed that the transpose and all functions defined by a permutation of the values a, b, c are also included in the class.

b	c	a
-	a	-
-	-	a

Ia

b	a	v
c	a	-
-	-	a

Ib

b	u	a
c	a	-
-	-	a

Ic

b	c	-
-	a	x
-	z	a

Id

where $v \in \{b,c\}$, $u \in \{b,c\}$, and $(x,z) \neq (a,a)$.

The classes are not disjoint, and Muzio constructs a table, using the Karnaugh map technique for drawing Venn diagrams, enumerating the total number of distinct functions in each of the 16 possible intersections of the classes and their complements. For the goal here of determining the number of isotopic classes in Type I, it is necessary to analyze the operation tables and construct a diagram similar to Muzio's, but differing in that each cell contains the number of distinct symmetric classes (designated by an *S* subscript), and the number of distinct non-symmetric classes (designated by an *N* subscript) in each intersection. This diagram is shown in the figure below.

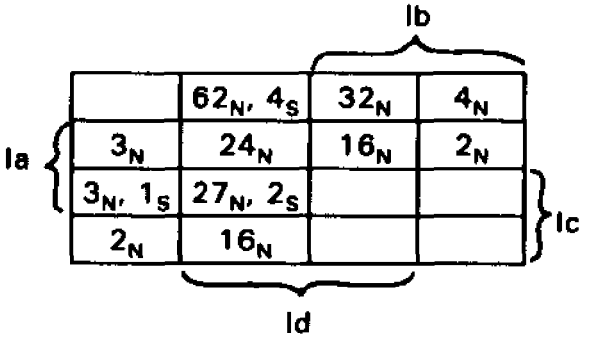


Figure 5. Distribution of isotopic classes: Type I functions

The brackets indicate those columns or rows included in the class. For example, there are 16 non-symmetric isotopic classes in the intersections of classes Ia, Ib, and Id; there are 62 non-symmetric and 4 symmetric isotopic classes in Id but not in Ia, Ib, or Ic.

Summing the numbers in each cell, shows the total number of isotopic classes for Type I functions to be 198. There are 191 non-symmetric

classes, with 12 members in each, and 7 symmetric classes, with six members in each, for a total of 2,334 distinct functions, which agrees with Muzio's calculation.

A program, not reproduced here because of its relative simplicity, was written to generate a representative value sequence for each of the isotopic classes of Type I. Arbitrarily, the representative selected was that corresponding to the assignments $a = 0$, $b = 1$, $c = 2$ in the operation table.

The program takes each operation table in turn and fills in the undesignated positions with every possible permutation (with repetition) of the values *a*, *b* and *c*. After each assignment is made, the function created is checked against a list of functions generated-so-far. If newly-generated, the function is added to the list and its value sequence printed out; otherwise, it is ignored. While straightforward, it must be noted that for classes Ia, Ic, and Id there exist pairs of dual functions, functions f_1 and f_2 such that $f_1 = \overline{f}_2$. The operation tables of such a pair of functions from class Ia are shown below.

b	c	a
c	a	b
a	a	a

b	c	a
c	a	a
a	b	a

The 12 members of the isotopic class comprised of the six permutations of f_1 along with their transposes, are exactly the same as the

12 members of the class comprised of the six permutations f_2 along with their transposes. Letting $a = 0$, $b = 1$, and $c = 2$, either the value sequence of f_1 or the value sequence of f_2 could be used as the representative of their common isotopic class, but not both. The program guards against keeping both sequences by checking to see that neither the function generated, nor its transpose, is on the list of functions generated-so-far before adding the function to the list and printing out its value sequence.

4.1.2 Type II

The calculation of the number of isotopic classes for Type II is done in an identical fashion as for Type I. Type II is partitioned into three classes (Muzio defines three classes, IIa, IIb and IIc, but IIb includes two separate cases, denoted by IIb1 and IIb2 here) whose operation tables are shown below.

b a a	b b -	b - b	b a b
w c x	- c a	a c -	- c x
y z a	- - a	- - a	- - a
IIa	IIb 1	IIb 2	IIc

where $(w,x,y,z,) \neq (bbcc)$ and $x \in \{a,c\}$

The analysis of these three classes yields the map on the following page, with the interpretation of entries as before.

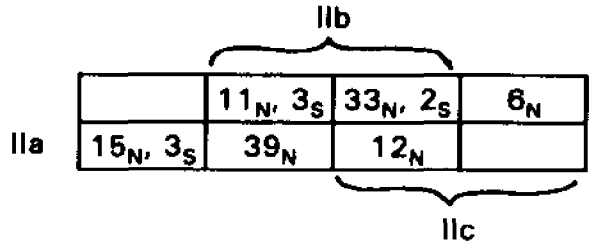


Figure 6. Distribution of isotopic classes: Type II functions

There are 116 non-symmetric classes and 8 symmetric classes of Type II, resulting in a total of 1,440 functions, again in agreement with Muzio's values.

The value sequences for representative functions of the isotopic classes of Type II functions were computed by a second program, structured similarly to the first. The difference lies in the observation that for Type II the two cyclic permutations (a b c) and (a c b) leave the diagonal unchanged. Consequently, if P is one of these permutations, there are functions f_1 and f_2 such that $f_1 = f_{2P}$ (this situation only is possible in classes IIa and IIb). The operation tables of such a pair of functions from class IIa are shown below.

b	c	a
b	c	c
b	c	a

b	a	a
b	c	a
a	c	a

The 12 members of the isotopic class that consists of the six permutations of f_1 and their transposes are identical with the 12 members of the isotopic class that consists of the six permutations of f_2 and their

transposes. Only one of the value sequences should be retained in the list of representatives of distinct isotopic classes. The program guards against keeping both sequences by checking if either the function formed, its conjugates under the three-cycle permutations, or their transposes is already on the list of functions generated-so-far before adding the function to the list and printing out its value sequence.

4.2 The Calculation of the Radii Over E(2)

The listing of the program that calculates the radius for Sheffer functions in *E(3)* is given in Appendix 1. Some of the features of the program can best be understood in the context of its evolution. The program was originally constructed to calculate the radius of a complete with constants set of functions over *E(2)*, as this case would serve as a test case for *E(3)*, one whose validity could be verified by hand calculations. This program for *E(2)* is described here, followed by a discussion of those modifications which were necessitated by the requirements of *E(3)*.

The program calculates the radius of a Sheffer with constants function or the radius of a complete set of functions having two members. In addition, for each selection of a Sheffer with constants function, or a set of functions, a minimum radius composition sequence using that function, or set of functions, is output for each of the 16 two-place functions in *E(2)*. To facilitate debugging, the original form of the program was interactive, querying the user to input the number of functions in the complete set

(either one or two), and then the letter designation (using the notation of Bochenski (1959)) and value sequence of each. Once debugged, the program could have been modified to accept input from a file, but the original version proved so convenient to use that it was decided to leave it in the original form.

For the sake of clarity, the explanation of the workings of the program is offered for the case of a single function in the complete set, with the explanation in the case of two functions being virtually identical.

The operation of the program centers around the inter-relationships and the manipulations of its major data structures, which are usefully visualized as shown in the figure on the next page.

The *values* array holds the value sequence of each function generated by compositions using the input Sheffer with constants function. The key feature is that the functions are generated, and then entered in the array, in increasing order of the radius. All functions obtained with zero compositions (radius = 0) appear first; all functions obtained through a single composition (radius = 1) appear next; and so on. Of course there is no ordering among functions having the same radius. The array is initialized with the value sequences of the *free* functions, x, y, 0 and 1. The figure depicts *values* at a stage where the computation of functions of radius two is in progress.

pointer

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	4	7													

values

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	0	1	1	1									
0	1	1	0	1	1	0									
1	0	1	0	0	1	1									
1	1	1	0	0	0	0									

name

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x	y	1	0	Dxx	Dxy	Dyy									

tally

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0			x		y					Dyy		Dxx		Dxy	1
0	-1	-1	0	-1	0	-1	-1	-1	-1	1	-1	1	-1	1	0

letter [D] count [9]

Figure 7. Data structures

The *name* array is keyed to *values*, with each location containing the composition sequence, in non-parenthesized form, of the function appearing in the same location of *values*. In the situation shown in the figure, using *D* as the Sheffer function, *name* [6] = *Dyy*, which is the composition sequence yielding the value sequence <1 0 1 0>, appearing in column 6 of *values*.

Tally is an array that contains the radius and composition sequence of each function generated by the compositions formed. If a function has not yet been generated, its radius appears as -1 and its composition sequence is null. The location within the tally array that holds the information for a specific function is the decimal equivalent of the value sequence of that function, treated as a binary number, with the most significant digit being the 0-0 entry in the operation table. As shown, the information for the function with value sequence <1 0 1 0> is found at location 10 in tally (1010_2 = 10), where the radius is shown as one and the name is *Dyy*.

The k^{th} location in the *pointer* array contains the number of the column within *values* where the value sequence of a function of radius *k* first appears. That is, *pointer*[0] holds the column number of the first appearance of a zero radius function (column 0); *pointer*[1] contains the column number of the first appearance of a function with radius one (column 4); and so on. Functions of radius *k* appear in *values* in columns lying between *pointer*[*k*] and *pointer*[*k*] - 1, inclusive.

The most important of the remaining variables are *letter*, which contains the letter designation of the Sheffer with constants function, and *count*, which registers the number of functions yet to be generated by composition.

The mechanics of the program are based upon the recognition that an n^{th} order composition sequence must be of the form:

$$S^n(x,y) = f(S^u(x,y), S^v(x,y)),$$

with $u + v = n - 1$ and $0 \leq u,v \leq n - 1$.

If all the distinct functions generated by u^{th} order sequences and v^{th} order sequences are known, then all functions generated by n^{th} order sequences may be obtained by forming the compositions,

$$f(S^u(x,y), S^v(x,y)) \text{ and } f(S^v(x,y), S^u(x,y)),$$

for

$u = 0, v = n - 1,$
$u = 1, v = n - 2,$
.
.
$u = k, v = n - k - 1,$ for $k \leq (n - 1)/2.$

If such functions duplicate previously generated functions they are ignored.

After obtaining its input, the program initializes *name*, *values* and *tally* with the information corresponding to the *free* functions x, y, 0, and 1, and also sets *count* to 12. Compositions are begun using these zero radius functions as arguments. After each composition, the index of the new value sequence formed is computed and used as a key into the *tally* array, where it is determined whether the function has already been generated by a composition of lower order or by a previous composition of the same order.

If this is the case, the new function is ignored; otherwise, *tally* is updated with both the radius of the new function and its composition sequence.

The process continues, always generating functions with a minimum radius composition sequence (although there may be more than one, in which case the program keeps the first one), until either *count* = 0, signifying all functions in *E(2)* have been generated, or a maximum number of compositions have been attempted. The latter condition is a safeguard that halts the program in the event a non-complete with constants function has inadvertently been entered.

If all functions have been generated, the largest entry in the *tally* array is output as the radius of the Sheffer with constants function. In addition, all information in *tally* is printed as shown in the typical output shown below.

```
function #0,  radius is 0,  name is 0
function #1,  radius is 2,  name is D1Dxy
function #2,  radius is 3,  name is D1DxDxy
function #3,  radius is 0,  name is x
function #4,  radius is 3,  name is D1DyDxx
function #5,  radius is 0,  name is y
function #6,  radius is 5,  name is DDxDxyDyDxx
function #7,  radius is 3,  name is DDxxDyy
function #8,  radius is 4,  name is D1DDxxDyy
function #9,  radius is 5,  name is DDxyDDxxDyy
function #10, radius is 1,  name is Dyy
function #11, radius is 2,  name is DyDxx
function #12, radius is 1,  name is Dxx
function #13, radius is 2,  name is DxDxy
function #14, radius is 1,  name is Dxy
function #15, radius is 0,  name is 1
largest number of operations = 5
```

4.3 The Calculation of Radii Over E(3)

The main problems encountered in making the transition from *E(2)* to *E(3)* concern the storage and time requirements for the radius calculation.

In *E(3)*, with 19,683 two-place functions, a naming convention similar to Bochenski's is impractical. A function is identified by its index, the decimal equivalent of the function's value sequence, treated as a ternary number whose most significant digit is the 0-0 entry in the operation table. This detail aside, before the values of the radii had been computed, it seemed conceivable that composition sequences for functions over *E(3)* could be 40 or 50 characters long. (This conjecture is subsequently confirmed by the large radii of certain functions.) Allowing for such a possibility would mean that the *name* array would have to be large enough to told 19,683 strings, some potentially 40 or 50 characters long. As the total capacity for such an array would be in the order of megabytes, this demand was obviously untenable. So, after being used in some short duration test runs to verify that compositions were being formed as expected, the *name* array was abandoned, along with the concomitant entry in the *tally* array that pointed to the composition sequence of the function.

Originally, in *E(2)*, the *values* array was implemented as a two-dimensional array where each column contains the value sequence of a function, stored one digit per row. In *E(3)*, such an implementation requires an array of 19,683 columns and 9 rows. Even with the entries

stored as characters (one byte), this requirement calls for an array of approximately 180,000 bytes, a number balked at by Turbo C++. Instead, the array was made one-dimensional, with the entry being the decimal index corresponding to the value sequence.

While solving the storage problem, this change increased the overhead of performing the operation of composition—the basic computation in the program—and hence the time of execution. Under the new implementation, after retrieving the two arguments for the composition, the indices have to be converted back into value sequences before the composition can be performed. To offset this increased overhead, the subroutine *compute* was modified so that after it computes the value sequences of the two arguments, call them *x* and *y*, it performs both compositions *fxy* and *fyx*, and then checks to see if they have been generated already.

The following is a brief description of the operation of the final version of the program as shown in Appendix A.

The initialization is identical with that for *E(2)* except the constant function 2 increases the number of *free* functions to 5. Next, the Sheffer function index is input at the keyboard. (Again, while the interactive mode was selected in the beginning so as to get a quick estimate of the typical time frame of a calculation, and while a second version of the program exists where the input is read from a file, the original version retains the

advantage of convenience.) There is the capability of entering a set of two complete functions, but this capability has never been utilized.

The computations are carried out within the main *while* loop, using *count* $\leq$ 0 or *level* $\geq$ MAX_LVLS as criteria for exiting the loop. In this loop, first, all functions generated by value sequences with a *level* number of compositions are determined. Specifically, for level = k, *compose* selects every combination of arguments of radius u and v such that $u + v = k - 1$, and calls *compute* to actually perform the composition. As previously mentioned, $f(R^u(x,y), R^v(x,y))$ and $f(R^v(x,y), R^u(x,y))$ are computed during the same call. If the functions are newly-generated, *tally* and *count* are updated accordingly. In this manner, all functions of radius = 1 are formed first; all functions of radius = 2, next, etc. This insures that the composition sequence for a particular function is always a minimum radius sequence.

Upon exiting the main *while* loop, if all functions in *E(3)* have been generated, the maximum radius is printed out (as is the average radius, a statistic easily obtained and therefore put in the program in case it might be of future interest. Its calculation has not proven of value.) If not all functions have been generated, a message is output stating the fact that the input function failed to form all functions within MAX_LVLS compositions.

One final note: The print statement within the *while* loop was inserted to allay the anxiety engendered by viewing a blank screen while the program was in operation. As some of the computations were hours in

duration, printing the number of functions yet to be generated at the completion of each level served to affirm that the program was indeed running. The output was originally directed to the screen, but was changed so that a hard copy record of each run could be obtained. This decision to print out a *history* of the generation of functions at each level turned out to be a fortuitous one, as is explained in the discussion of future research.

4.4 Results

The data, taken over a period of months on a 386, 20 Mhz machine, are summarized in Appendix B. For Type I and Type II functions, the minimum radius is seven and the maximum twenty-five. By actual timed runs, the minimum radius calculations took approximately three minutes, performing approximately 800,000 compositions, while the maximum radius calculation required 6 hours and 33 minutes, performing approximately 126,000,000 compositions.

CHAPTER 5

THE TAXONOMY OF THE SPACE OF SHEFFER FUNCTIONS

5.1 Introduction

Appendix B partitions Sheffer functions into classes distinguished by a common value of the radius. Those classes of small radius are comprised of what are termed *efficient* functions, while those of large radius are comprised of what are termed *inefficient* functions. A number of factors have been analyzed in an attempt to characterize how the value sequences of efficient functions differ from those of their inefficient counterparts with the thought of using similar criteria to predict the efficiency of Sheffer functions over $E(k)$, with $k > 3$.

5.2 Factors Affecting Efficiency

5.2.1 Type

Since the most obvious distinction among operation tables of Sheffer functions is the different distribution of values along the diagonal, Type I being of the form [1 0 0] and Type II of the form [1 2 0], the effects of this factor are examined first. The following graph shows the percentage of functions in each radius class for functions of Type I, functions of Type II, and for all functions. (Classes of radii 17 through 25 have been merged into

a single class, labelled 17+, as the number in each individual class is too small to allow meaningful comparisons.)

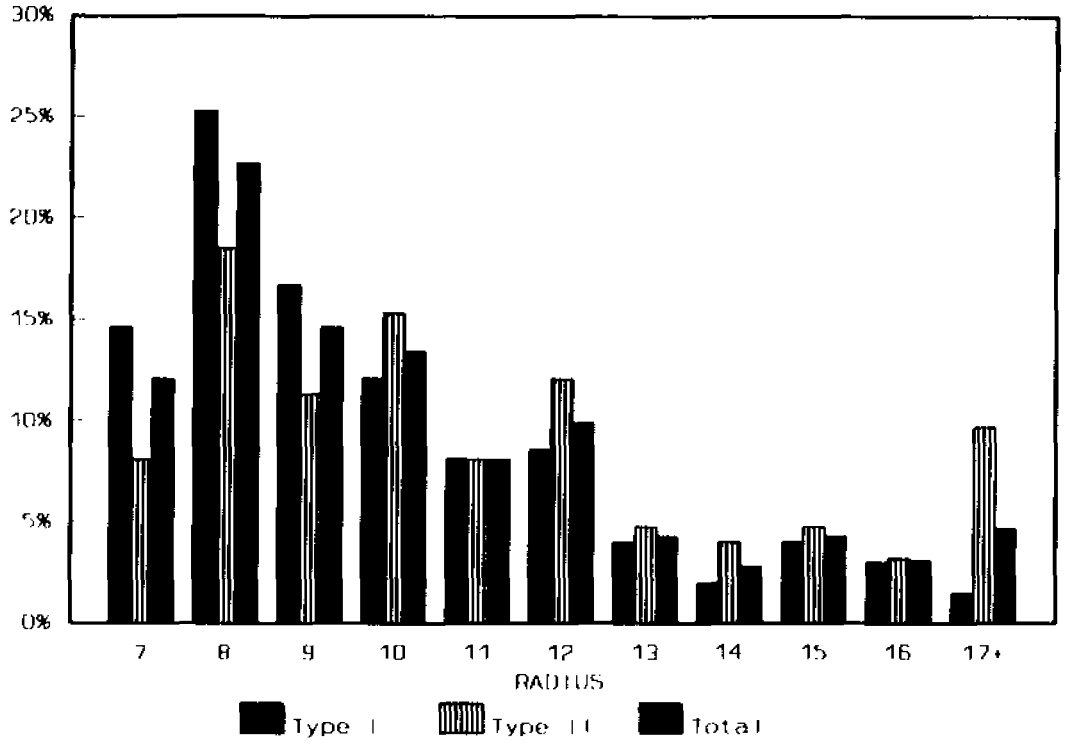


Figure 8. Distribution of functions among radius classes

The percentage of Type I functions in the efficient classes (arbitrarily taken to be $7 \leq r \leq 10$) is slightly larger than that of Type II, while the percentage of Type II functions in the inefficient categories ($r \geq 17$) is slightly higher than that of Type I, with the percentage in the intermediate classes being approximately the same for each. The differences in percentages are small, however, and recognizing that there are significant numbers of Type I and Type II functions in both efficient and inefficient categories, *type* is abandoned as a discriminator of efficiency.

5.2.2 Off-diagonal Distribution

If, instead of comparing sequences with different diagonals, a comparison is made of functions with the same diagonal, so that the focus is either solely on functions of Type I or solely on functions of Type II, it seems reasonable to assume that efficiency depends on the number of appearances of the values 0, 1, 2 among the six off-diagonal positions of the operation table. The two tables at the end of this section display for each possible distribution of the values of 0, 1, 2, the number of functions in each radius class. The three-tuple at the left side of the table represents the number of zeroes, ones and twos in the distribution, so that 006 is a distribution with 0 zeroes, 0 ones and 6 twos. For Type I functions, because the only values along the diagonal are 0 and 1, and because one of the requirements of a Sheffer function is that its operation table contain each of the values 0, 1, 2, at least one value of 2 must appear in an off-diagonal position. This is not a requirement of Type II functions, where all three values appear along the diagonal.

Unfortunately, the tables offer little aid in identifying a characteristic of efficient functions, except to confirm that value sequences with either all twos or all zeroes in the off-diagonal positions (all ones is not a possible assignment) are very inefficient functions. Surprisingly, skewed distributions such as 015, 105, 150, 501, 510, do not necessarily create inefficient functions, as radii for these distributions range over the entire spectrum of

possible values, and relatively uniform distributions such as 123, 132, 213, 222, 231, 312, 321 do not necessarily create efficient functions, as radii for these distributions also range over the entire spectrum of values.

	7	8	9	10	11	12	13	14	15	16	17+	Total
006									1			1
015	1		1			1						3
024	2	3	2			1						8
033	4	2	1			1						8
042	1	2	1		1							5
051				1								1
105				1	1				1			3
114	3	5	1	2	1	2						14
123	4	7	7	4		1	1					24
132	5	5	2	1		3	1		1			18
141			2	1		1			1			5
204		1		2	1	2	1					7
213	3	8	3	4	3	2						23
222	4	6	8		3	2					1	24
231		4		1	2			1		1		9
303	1		1	2		1			1	1	1	8
312	1	5	1	2	3		2		1	3		18
321		2	1	1	1		1	2	1	1		10
402			1	1			1		1			4
411			1	1			1	1				4
501											1	1

Figure 9. Off-diagonal distribution: Type I

	7	8	9	10	11	12	13	14	15	16	17+	Total
114									1			1
123	1	1	2									4
132	2	1		2			1					6
141	1		1	1							1	4
150				1				1				2
213		1	2		1	1					1	6
222	2	4	3	1		3	3	1				17
231	2	3	1	3	1	2					2	14
240		1		2				1	1	1	1	7
303		1										1
312	1	2	1	3		2	1		1			8
321	1	6	3		3	2	2		2			19
330		1		3	1	1				1	2	9
402					1			1				2
411		2		3	1	2		1			1	10
420			1		2	2		1		1	2	9
501								1				1
510							1		1		1	3
600											1	1

Figure 10. Off-diagonal distribution: Type II

5.2.3 Algebraic Properties

Of the fifteen symmetric functions, none appears among the two most efficient classes, radius = 7 and radius = 8, while four appear in the least efficient class, radius = 17+ (actual radii are 17, 18, 19, and 21), with the remainder exhibiting radii ranging from 9 to 15. Evidently, symmetry militates against efficiency. In light of this fact, other algebraic properties were examined to see if they impact upon efficiency.

It has been shown by Salomaa (Sa60) that two-place Sheffer functions cannot be associative, so each Sheffer function was tested to see how many of the following weaker conditions it might satisfy.

power associative: $f(a, f(a,a)) = f(f(a,a), a)$

left alternative: $f(a, f(a,b)) = f(f(a,a), b)$

flexible: $f(a, f(b,a) = f(f(a,b), a)$

right alternative: $f(a, f(b,b)) = f(f(a,b), b)$

The results of this testing are that, except for the symmetric functions, which must be both power associative and flexible, there are only nine functions that are power associative, and no functions that satisfy any of the other conditions, and so it may be possible to strengthen Salomaa's result. Clearly, these algebraic properties strongly militate against completeness, but have little bearing upon efficiency.

5.2.4 Stability

When viewing large numbers of operation tables and their corresponding radii, the difference in *stability*, the sensitivity of the radius to small perturbations among the off-diagonal values, is striking. For some functions changing a certain value of an off-diagonal entry results in a function of dramatically different radius, while changing the value of a different entry produces little if any change; for other functions, the change in radius is small no matter which entry is changed.

To investigate this phenomenon, the value sequence of a function is

interpreted as the coordinates of a point in nine-dimensional space. The Hamming distance between a point representing a function f_1 and a second point representing a function f_2 is defined as

$$d = \sum_{i=1}^{9} \delta_i, \quad \text{where} \quad \delta_i = \begin{cases} 0, & (x_i)_1 = (x_i)_2 \\ 1, & (x_i)_1 \neq (x_i)_2 \end{cases}$$

The Hamming distance is equal to the number of positions within the value sequences where f_1 and f_2 have entries that disagree. The set of all functions on a *sphere* of radius one centered at a particular function consists of all functions obtainable from that particular function by a single change in the value sequence. Each set of functions consists of at most 12 functions, accounting for two changes for each of the six off-diagonal elements. (While not all the functions in the set are Sheffer when the off-diagonal elements are changed, virtually none of the resulting functions are Sheffer if the diagonal elements are changed. Even if these functions are Sheffer with constants, the radius values are not available, making such functions useless for this comparison.)

The range of radii among the functions on the sphere centered about a particular function is a measure of the stability of that particular value sequence. The following table shows the maximum range and minimum range of radii in the spheres centered at functions within each radius class.

radius	minimum variation	maximum variation
7	7 - 8	7 - 15
8	7 - 9	8 - 18
9	7 - 8	8 - 21
10	7 - 9	8 - 25
11	7 - 8	7 - 17
12	7 - 10	8 - 25
13	8 - 15	8 - 25
14	7 - 12	9 - 18
15	9 - 13	8 - 25
16	10 - 12	9 - 21
17+	13 - 17	8 - 25

Figure 11. Stability of radius

While not directly correlated to efficiency, stability leads to the following.

5.2.5 Criteria For Efficient Functions

Examining the changes that produce dramatic increases or decreases in the radius, leads to a recognition of certain factors that make for efficient functions. Let us define the following five conditions regarding the operation table of a function.

1. There are no columns or rows that contain only a single value.
2. There are no duplicate columns or rows.
3. There is at least one column or one row containing three different values.
4. There is no symmetry about the diagonal.

5. Each of the values 0, 1, 2, appears at least once in an off-diagonal position.

It is asserted that any function that satisfies the set of conditions above is an efficient function.

Each of the graphs in the following series of five graphs shows the percentage of functions in each radius class satisfying a subset of the five conditions above. For the first graph the subset consists of Condition 1 alone, with each subsequent graph adding a single condition, in numerical order, until for the last graph the subset consists of all five conditions.

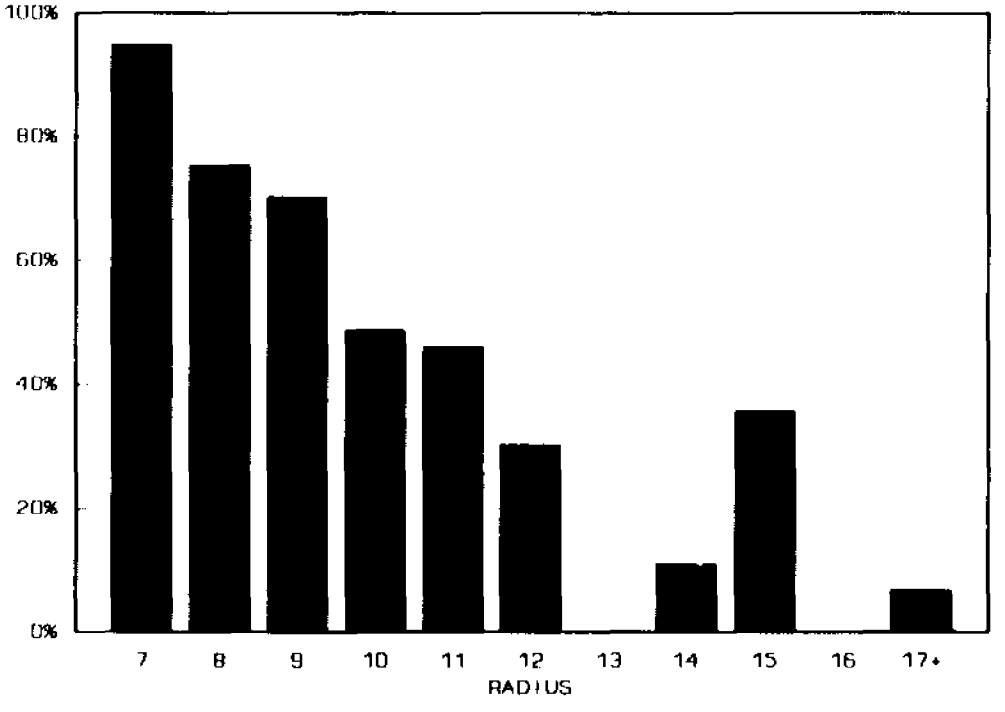


Figure 12. Percentage of functions satisfying Condition 1

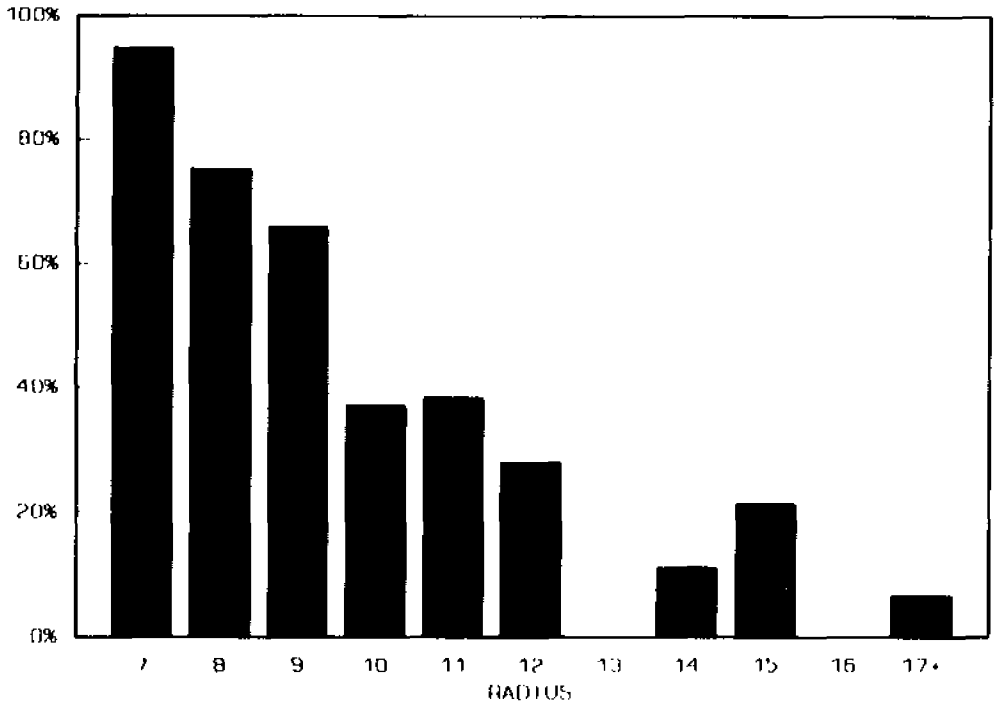


Figure 13. Percentage of functions satisfying Conditions 1-2

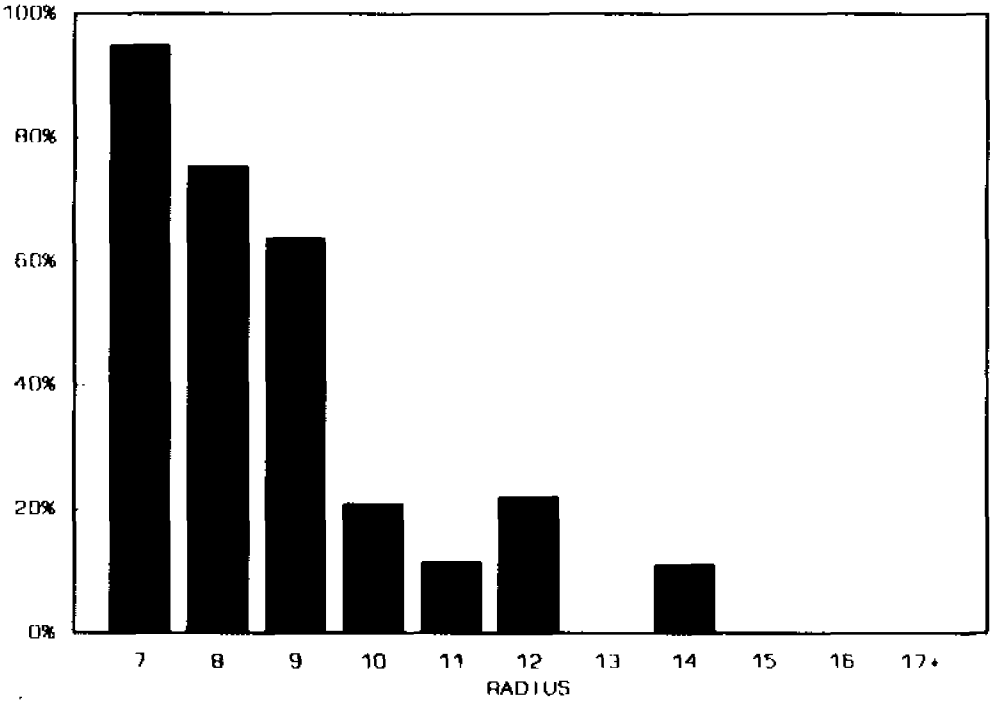


Figure 14. Percentage of functions satisfying Conditions 1-3

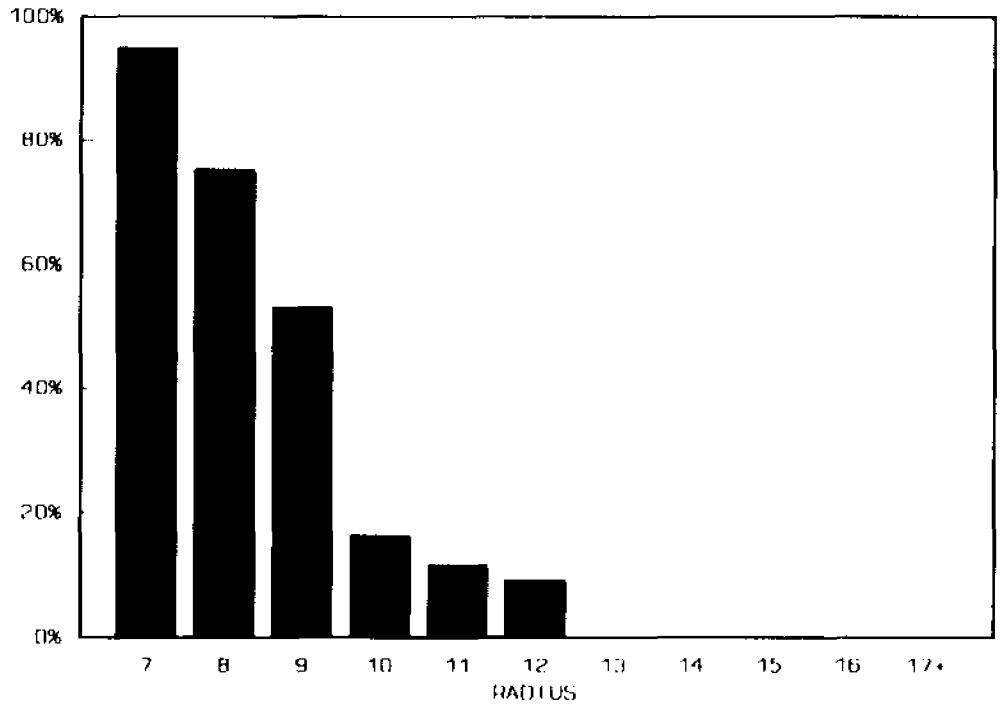


Figure 15. Percentage of functions satisfying Conditions 1-4

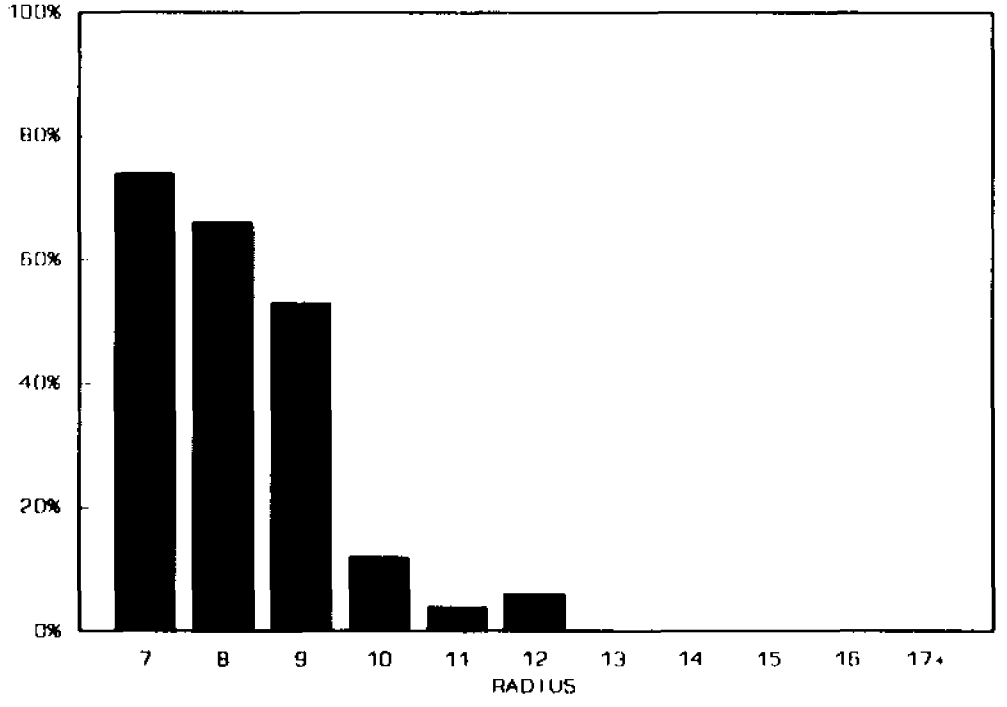


Figure 16. Percentage of functions satisfying Conditions 1-5

An electrical analogy proves useful in interpreting the graphs above. Each condition is considered as a filter of functions, and the set of five conditions as a series circuit of five filters, connected in ascending order of the condition number. The input to the first filter (Condition 1) consists of all Sheffer functions over the entire spectrum of radii, so that, in the format of the graphs above, the graph of the input would show bars of height 100% for each radius class. The five graphs above are interpreted as the *frequency response graphs* of the outputs taken after each successive stage of filters in the circuit.

Viewed in this manner, it is apparent that Condition 1 is the most important in distinguishing efficient functions, as it filters out most of the functions of large radius, while allowing virtually all functions of small radius to pass through. The subsequent conditions act to eliminate totally all large radius functions, and also attenuate most of the functions of intermediate radius. Condition 5 is the least important condition. Whether it is included or excluded, the largest value of the radius for a function satisfying the criteria is 12. Condition 5 has been included because it reduces the number of functions of radius 10, 11, and 12 that satisfy the criteria (at the expense of also reducing the number of efficient functions of radius 7, 8 and 9 that satisfy the criteria).

If inefficient functions are considered to be functions whose radii are roughly in the range 10 - 25, there are virtually no inefficient functions

satisfying these criteria. If a selection of a function is made based upon these criteria, the probability of selecting a *poor* function is effectively zero, with an expected value of the radius being approximately eight.

Of course, there are efficient functions that also fail to satisfy the criteria, so the criteria offer a set of sufficient but not necessary conditions for a function to be efficient.

Since symmetric functions simplify the synthesis of switching circuits, and, consequently, are the most likely to be implemented in hardware, identifying efficient symmetric functions is desirable. Obviously, the criteria established above is not usable as all symmetric functions are eliminated through Condition 4. With only 15 functions from which to draw conclusions it is difficult to devise a good test to discriminate the efficient symmetric functions from the inefficient. The following set of conditions on the operation table seems the best discriminator.

1. No column (or row) has a single truth value.
2. Each truth value appears at least twice.

No symmetric function with radius greater than 12 satisfies these two conditions, while every symmetric function with radius less than 12 does. Actually, only one of the three functions of radius 12 satisfies the criteria, and the comparison of this function and one of the radius nine functions, whose operation tables are shown on the following page, remains one of the unexplained puzzles.

1	2	1
2	0	1
1	1	0

radius = 9

1	2	1
2	0	2
1	2	0

radius = 12

Logic seems to indicate that since all other factors are equal, the function with the more uniform distribution of the values {0, 1, 2} would be the more efficient. Yet, this is not the case.

5.2.6 The Density of Members of an Isotopic Class

While the criteria set out above allow one to discriminate between good and poor functions, they do not serve to make the finer distinctions between a radius 7 function and a radius 9 function, for example, or a radius 13 function and a radius 19 function. The inter-relationships of the members of the isotopic classes themselves were studied in the hopes that they would provide such explanations.

Once again, the functions are treated as points in nine-dimensional space, and the Hamming distance is used as the distance measure. The identity permutation is selected to serve as the origin, and the distance between it and every other point in the class is computed. This calculation associates with each isotopic class a sequence of 11 numbers that describe the density of the functions in that class. If the numbers are relatively small, the functions are closely packed; if the numbers are large, the functions are loosely packed. The distances were calculated with the hope that the

patterns obtained could in some way be correlated with the differences in efficiency.

A program was written to calculate these distance sets, and the total of the Hamming distances was also computed as a quick means of comparing one isotopic class with another. Unexpectedly, the results show that the total of the Hamming distances between the identity permutation and its conjugates and transposes is always one of the following numbers: 39, 54, 63, or 66. The following is an explanation of this result.

First, consider the diagonal elements of the operation table for Type I functions under the identity permutation.

00	11	22
1	0	0

Recall that a Hamming distance between two functions is equal to the number of positions in the operation tables where the functions differ. Also, the arbitrary selection of the identity permutation as the origin, means that all comparisons are made against the table under this permutation. The elements along the diagonal are unaffected when taking the transpose of a function; therefore, considering only these elements, a permutation and its transpose make the same contribution to the calculation of the total Hamming distance. The table on the following page shows the diagonal elements for each of the six permutations.

00	11	22
1	0	0
2	0	0
1	0	1
1	2	1
2	2	0
2	2	1

Figure 17. Diagonal elements under six permutations

The identity permutation is the first row in the table, and the total number of entries that do no agree with the entry in this row of the column is 9, which becomes 18 when the transposes are included. The computation for Type II yields the same value.

Next, consider the six off-diagonal elements of an operation table where the identity permutation is symbolized as shown in the table below.

	0	1	2
0	-	A	B
1	C	-	D
2	E	F	-

The new locations of each of these off-diagonal elements corresponding to each of the six permutations and the six transposes is shown in the table on the following page. The entry in each of the off-diagonal positions is shown first for the permutation and then for the transpose, with the identity permutation, as the reference, again appearing

in the first row. The notation, A_{102}, is used to represent the numerical value of A under the permutation $\begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 2 \end{pmatrix}$.

01	02	10	12	20	21
A_{012}	B_{012}	C_{012}	D_{012}	E_{012}	F_{012}
C_{012}	E_{012}	A_{012}	F_{012}	B_{012}	D_{012}
B_{021}	A_{021}	E_{021}	F_{021}	C_{021}	D_{021}
E_{021}	C_{021}	B_{021}	D_{021}	A_{021}	F_{021}
C_{102}	D_{102}	A_{102}	B_{102}	F_{102}	E_{102}
A_{102}	F_{102}	C_{102}	E_{102}	D_{102}	B_{102}
E_{120}	F_{120}	B_{120}	A_{120}	D_{120}	C_{120}
B_{120}	D_{120}	E_{120}	C_{120}	F_{120}	A_{120}
D_{201}	C_{201}	F_{201}	E_{201}	A_{201}	B_{201}
F_{201}	A_{201}	D_{201}	B_{201}	C_{201}	E_{201}
F_{210}	E_{210}	D_{210}	C_{210}	B_{210}	A_{210}
D_{210}	B_{210}	F_{210}	A_{210}	E_{210}	C_{210}

Figure 18. Off-diagonal elements under six permutations and transposes

Every element, A, B, C, D, E, F, appears twice in each column with a value under two different permutations, so each of the elements behaves the same, as A, whose behavior is now described. Under the identity permutation, A appears at position 01, that is, row = 0 and column = 1. Suppose under this permutation that A has a value equal to its row number: $A_{012} = 0$. Then in the second column, which shows the entries at position 02, A appears as A_{021}, and A_{201}, with values zero and two, respectively. Under one permutation, its value agrees with the row number of its position, and under the second permutation its value agrees with the column number. The

same behavior occurs if under the identity permutation, A agrees with its column number. But, if under the identity permutation, A takes on a value different from its row and column numbers, then in each transpose and permutation, its value differs from its row and column position.

To summarize the above: Considering six permutations and six transposes, each element appears twice in each of the off-diagonal positions. If under the identity permutation an element has a value in agreement with either its row or column position, then in one of its appearances at a particular off-diagonal position it agrees with its row number, and in the other appearance it agrees with its column number. If under the identity permutation an element agrees with neither its row or column number, then in each of the two appearances at a particular off-diagonal position it agrees neither with the row nor the column number.

The contribution of the off-diagonal elements to the total of all Hamming distances is the sum of the number of different values observed when each element in a column is compared with the element in the first row of that column.

Assume under the identity permutation that n entries, $0 \leq n \leq 6$, agree with either a row or column position, and, therefore, that $6 - n$ entries do not. In each column in the table there are n entries that agree with a row number, n entries that agree with a column number, and $2(6 - n)$ entries that agree with neither row nor column. Consider a column headed by an entry that agrees

with a row number. There are $2(6 - n)$ entries that differ with this head entry because they agree with neither row nor column, and n entries that differ with this head entry because they agree with a column number. Hence, there are a total of $2(6 - n) + n$ entries different from the value under the identity permutation. The same number is obtained for a column headed by an entry in agreement with a column number. The n columns of this type, contribute $[2(6 - n) + n]n$ to the total Hamming distance.

Consider a column headed by an entry that agrees neither with its row or column position. There are n entries that differ with this head entry because they agree with the column number, n entries that differ with this head entry because they agree with the row number. Hence, there are a total of $2n$ entries different from the value under the identity permutation. The $6 - n$ columns of this type contribute $2n(6 - n)$ to the total Hamming distance.

The off-diagonal contribution to the total Hamming distance is

$$[2(6 - n)]n + 2n(6 - n) = 3n(8 - n)$$

And, the total Hamming distance is

$$3n(8 - n) + 18 = 3[n(8 - n) + 6]$$

For Sheffer functions, the values of n, the number of entries in the identity permutation in agreement with either a row or column number, range from one to five, and, therefore, the total Hamming distance must be one of the values 39, 54, 63, 66, explaining the phenomenon observed.

As far as the individual sets of Hamming distances, there appear to be many different patterns, but the explanation here also is based upon the number of elements in agreement with a row number or a column number under the identity permutation. More importantly, the results do not appear to correlate at all with efficiency of functions.

CHAPTER 6

FUTURE RESEARCH

6. Future Research

The obvious extension of this work would be a determination of efficient functions over *E(4),* ideally a calculation of the radii of complete two-place functions over *E(4).* Any thought of performing such a calculation using a modification of the program in Appendix A, however, is quickly dispelled by the following rough estimate of its storage requirement and possible execution time.

The *compute* subroutine requires approximately 180 microseconds per call. A typical function of radius seven calls the *compute* routine about 50 times for every successful composition (i.e., one that forms a function not generated previously), while the function of radius 25 requires about 5,000 calls for every successful composition.

If we assume that radii in this range are possible in *E(4)* (which seems likely considering the minimum radius in *E(2)* is five and in *E(3)* is seven), and assume that *hit* ratios ranging from 40:1 to 5,000:1 are possible (since these ratios seem to grow exponentially with radius, and are dependent upon the number of functions formed at each value, it seems likely that the actual hit ratios will be significantly higher in *E(4)*), and finally assume the

use of a processor faster by a factor of 5, then, a typical calculation of radius in *E(4)*, where there are 4,294,967,296 functions, would take between 70 days and 2.5 years. That is,

$$\frac{.00018}{5}\ \frac{sec}{call} \cdot 4^{16}\ functs. \cdot 40\ \frac{calls}{funct.} \cdot \frac{1}{86,400}\ \frac{days}{sec} = 70\ days.$$

Even if the execution time can be significantly reduced by rewriting the program to take advantage of some massively parallel computer, and one were willing to invest the time in a single calculation, in its present format the *values* array would now have to hold 4^{16} long integers for a storage capacity of approximately 17 gigabytes, a presently unattainable figure.

Clearly, a totally different approach must be employed. Wesselkamper and Cabrasawan (1994), and Wesselkamper, Lesniak and Cabrasawan (1994) have suggested a modification of the genetic algorithm that seems well-suited to this task. To illustrate the method, the following describes how the genetic algorithm has been used by Wesselkamper and Cabrasawan in identifying efficient functions in *E(3)*.

The method begins with a selection of an initial population of *2N* functions, each represented by its nine ternary digit value sequence. A fitness function *F(f)* is defined so that for two functions f_1 and f_2 if $F(f_1) > F(f_2)$, than in some sense f_1 is a more fit function than f_2. With the results in Appendix B now available, the obvious choice of fitness is the value of the radius. So, if one function is Sheffer with constants and the

other is not, the Sheffer with constants function is deemed more fit; if both functions are Sheffer with constants the one with the smaller radius is deemed more fit; if neither function is Sheffer with constants, than both are deemed unfit.

A random selection of two value sequences, called the *parents*, is made from the population. A new value sequence, called the *child*, is formed so that in those positions where the parents have the same ternary digit, the child inherits that same digit, and in those positions where the parents have different ternary digit, the child inherits a value selected randomly from one of the parent's digits, with equal probability of selection being assigned to each choice. With some small probability a mutation of the digits of the value sequence of the child is introduced. The child is then added to the population and the most fit *2N* value sequences are selected to form a new population P_1, and the process is repeated.

Wesselkamper and Cabrasawan have found that even if the original population does not include a Sheffer with constants function, as soon as two Sheffer with constants functions are formed the method converges rapidly to a population comprised entirely of Sheffer functions with radius 7 and 8.

In *E(4)*, the radii are not known, and, therefore, some other fitness function must be devised. In looking at the output of runs of the program in Appendix A, it is obvious that even with small numbers of compositions,

efficient functions generate significantly larger numbers of functions than inefficient ones. For example, within a radius of 5 (all functions generated with 5 or fewer compositions) a typical function of radius 7 has generated approximately 14,000 functions, a radius 9 function has generated approximately 8,000 functions, a radius 11 function has generated approximately 2,600 functions, and the radius 25 function has generated approximately 660 functions. This observation led Wesselkamper and Cabrasawan to suspect that which of two functions is more fit could be established by determining which of the two generated more functions in a fixed number of compositions. Actually, they have determined that a better measure is the number of functions generated in a fixed amount of *time*—rather than a fixed number of compositions—keeping the processor speed and the generating algorithm fixed.

In *E(4)*, beginning with a population of Sheffer with constants functions, it seems promising that the genetic algorithm technique will be able to identify efficient functions. Obtaining an actual radius calculation for one of them, however, will still require overcoming the storage requirement obstacle.

APPENDIX A

```
/* The calculation of the radius of Sheffer functions over E(3) */


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <alloc.h>
#include <process.h>


#define BASE          3    /* Ternary numbers */
#define FALSE         0
#define FREE          5    /* Number of zero radius functions */

#define MAX_COL 19683      /* Maximum # of columns in values array */

#define MAX_LVLS     25    /* Maximum # of compositions to attempt */

#define MAX_VALS      9    /* Number of digits in value sequence */

#define TOTAL     19683    /* Number of different functions */
#define TRUE          1


                           /* Global variables */

int level = 1;             /*  Number of compositions required to
                           ** generate the function. */

int next_col = FREE;       /*  Next column open in values array */

int pointer[MAX_LVLS+1];   /*  An array indexed to the values array.
                           ** Pointer[0] points to the first column in the
                           ** values array holding zero radius functions;
                           ** pointer [1] points to the first column holding
                           ** radius one functions, etc.
                           */
```

```
char *tally;                    /*  An array recording the number of
                                ** compositions required to generate each of
                                ** the individual ternary functions, indexed by
                                ** the decimal equivalent of the value
                                ** sequence of the function
                                ** (MSB = 0-0 entry).
                                */

int *values;                    /*  An array holding the decimal equivalent of
                                ** the value sequence of each function
                                ** generated, in the order in which it was
                                ** generated.
                                */

int num_functions;              /*  Number of functions in complete set.  Allow
                                ** for a maximum of two functions.
                                */

FILE *prn_ptr;

    /* Program begins */
main()
{

    int functor[2][MAX_VALS];
                                /*  Value sequence for each function in
                                ** the complete set. */

    int count = TOTAL - FREE;
                                /*  The number of functions left to be
                                ** generated.  When count reaches zero, the
                                ** program is finished.
                                */

                                /*  Subroutine declarations */

    void compose(int p1, int p2, int functor[][MAX_VALS], int *counter);

    void compute(int col1, int col2, int functor[], int *counter)

    void input(int in[], int j);

    int expon(int base, int exp);
```

```
float stat(int *largest, int size);


                        /*  Local variable declarations */
int i,j,
int max;                /*  Largest radius of all functions */
fioat avg;

prn_ptr = fopen("PRN", "wt");
                        /*  Request storage for data structures */

values = malloc ( sizeof (int) * TOTAL );
if (values = = NULL)
    {
        fprintf(stderr, "Unable to allocate memory for values.\n");
        return 1;
    }

                        /*  Initialize value array with free functions:
                        **                  x   y   zero one  two
                        **                  0   0   0   1   2
                        **                  0   1   0   1   2
                        **                  0   2   0   1   2
                        **                  1   0   0   1   2
                        **                  1   1   0   1   2
                        **                  1   2   0   1   2
                        **                  2   0   0   1   2
                        **                  2   1   0   1   2
                        **                  2   2   0   1   2
                        */

values[0] = 377;        /*  <0 0 0 1 1 1 2 2 2> = 377 in
                        ** base 3, etc. */
values[1] = 3785;
values[2] = 0;
values[3] = 9841;
values[4] = 19682;

tally = malloc(sizeof(char) * TOTAL);
if (tally = = NULL)
  {
  fprintf(stderr, "Unable to allocate memory for tally.\n");
  return 1;
  }
```

```
                        /*  Initialize tally array with zero radius
                        **  functions marked with 0; all others
                        **  marked as ungenerated with -1.
                        */
for ( i = 0; i < TOTAL; i++)
    tally[i] = -1;

                        /*  The constants zero, one and two marked as
                        **  zero radius functions.
                        */

tally[0] = tally[9841] = tally[19682] = 0;

                        /*  x = <0 0 0 1 1 1 2 2 2>, index of 377,
                        **  and y = <0 1 2 0 1 2 0 1 2>, index of
                        **  3785, also marked as zero radius
                        **  functions.
                        */
tally[377] = tally[3785] = 0;

                        /*  Initialize pointer array */

*pointer = 0;           /*  Zero radius functions begin in column
                        **  zero of values array. Since there are
                        **  5 free functions, radius one functions
                        **  begin in column FREE = 5.
                        */
*(pointer + 1) = FREE;

                        /*  Query for the number of functions in the
                        **  complete set.  A maximum of two are
                        **  allowed.
                        */
printf("\nEnter the number of basis functions: ");
scanf("%d", &num_functions);

for (j = 0; j < num_functions; j++)
    {
        input(*(functor + j), j+1);
                        /*  Use subroutine to get value
                        **  sequence */
        printf("\n");
    }
```

```
                        /*  Start compositions.  Stop when all
                        **  functions have been generated or
                        **  maximum number of compositions
                        **  have been attempted.
                        */
while ( (count > 0) && (level < MAX_LVLS) )
    {
        i = 0;
        j = level - 1;

                        /*  Form all functions of radius = "level" */

        while (i < = j)
            compose(i + +, j--, functor, &count);

        level + +;      /*  Increment level and get ready for next
                        **  round of compositions. */
        *(pointer + level) = next_col;

                        /*  Print out number of functions yet to be
                        **  generated. */

        fprintf(prn_ptr, "level = %d, count = %d\n", level - 1, count);
    }

                        /*  Calculations complete - print out data */

if (level > = MAX_LVLS) /*  Couldn't generate all functions in
                        **  MAX_LVLS ops. */
    if ( num_functions = = 1 )
        {
            fprintf(prn_ptr,"\nfunction  failed in %d operations\n",
                        MAX_LVLS - 1);
            fprintf(prn_ptr,"\ncount = %d", count);
            fprintf(prn_ptr, "\nfunctions not generated yet:\n\n");
            for (i = 0; i < MAX_COL; i + +)
                if (tally[i] = = -1)
                    fprintf(prn_ptr, "\n function #%d", i);

        }
    else
        {
            fprintf(prn_ptr, "\nfunctions failed in %d operations\n",
                        MAX_LVLS - 1);
```

```
                fprintf(prn_ptr, "count = %d", count);
            }
    else
        {                       /* Print out statistics */
            avg = stat(&max, TOTAL);
            fprintf(prn_ptr, "\n\n largest number of operations = %d\n",
max);
            fprintf(prn_ptr, "\n average number of operations = %f\n", avg);
        }
  free((void *) tally);
  fclose(prn_ptr);
  return 0;
}

/*  Query user for input of the decimal equivalent of the function's value
**  sequence.
**
**  ARGUMENTS
**    in    - an array holding the value sequence of the function
**    j     - number of the function from the complete set.  First function is
**            number 0, second is number 1
**
**  RETURNS
**    nothing
*/

void input (int in[], int j)
{
    int bin;
    int denom;
    int i;

    printf("\nEnter the decimal equivalent of ");
    printf("function %d's value sequence:  ", j);
    printf("\n(MSB is the 0-0 entry):  ");
    scanf ("%d", &bin);
    fprintf(prn_ptr, "function # %d\n", bin);

                            /* Convert decimal to ternary, and fill
                            ** array. */

   for (i = 0; i < MAX_VALS; i++)
   {
    denom = expon(BASE, MAX_VALS - 1 - i);
```

```
    in[i] = bin/denom;
    bin = bin % denom;
   }
}
/*  Exponential
**
**  ARGUMENTS
**    base - base of the exponential
**    exp  - exponent
**
**  RETURNS
**    prod (integer)
*/

int expon(int base, int exp)
{
    int i, prod;

    prod = 1;
    for (i = 0; i < exp; i++)
        prod = prod * base;
    return (prod);
}

/*  Specifies which of the previously generated functions should be used in
**  the next composition.
**
**  ARGUMENTS
**    p1          -   radius of the first function used as an argument in the
**                    composition
**    p2          -   radius of the second function used as an argument in
**                    the composition
**    functor     -   an array holding the value sequence of each of the
**                    functions in the complete set.  At most two functions.
**    *counter    -   pointer to "count" of functions yet to be generated
**
**  RETURNS
**     nothing

void compose(int p1, int p2, int functor[][MAX_VALS], int *counter)

{
    int i, j, k;
    void compute (int col1, int col2, int functor[], int *counter);
```

```
    for (k = 0; k < num_functions; k++)
    {                           /* Form every combination of a function
                                ** of radius p1 with a function of radius
                                ** p2.
                                */

        for (i = *(pointer + p1); i < *(pointer + p1 + 1); i++)
            for (j = *(pointer + p2); j < *(pointer + p2 + 1); j++)
                compute(i, j, *(functor + k), counter);
    }
}
/* Denoting "functor" by F, computes the value sequences of the functions
** obtained by the compositions Fxy and Fyx, where x is the value
** sequence at location v1 in the values array and y is the function at
** location v2 in the values array. If the functions generated by the
** compositions have not been generated previously, the radius of each is
** entered in the appropriate location in the values array and the value of
** "count" is decremented.
**
** ARGUMENTS
**   v1          - the location in the values array of the value sequence of
**                 the first function to be used as an argument in the
**                 composition
**   v2          - the location of the second value sequence to be used as
**                 an argument
**   functor     - the function being used
**   counter     - the address of the "count" of functions generated so far
**
** RETURNS
**   nothing
*/

void compute(int v1, int v2, int functor[], int *counter)
/* calculate compositions functor(value[v1], value[v2]) and
** functor(value[v2],value[v1]) */

{

    static int three[] = {1, 3, 9, 27, 81, 243, 729, 2187, 6561};
    int a, b, c, d,i;
    int index = 0;              /* Index into the tally array of the
                                ** function formed by the first
                                ** composition */
```

```
int index1 = 0;          /* Index for second composition */
int temp;
int a_mod3, b_mod3;

                         /* Convert the indices, "value[v1]" and
                         ** "value[v2]", of the two functions
                         ** used as arguments in the
                         ** composition into value sequences by
                         ** changing the decimal index into a
                         ** ternary number. The conversion is
                         ** done one digit at a time, starting
                         ** with the least significant digit, and as
                         ** each digit is found the corresponding
                         ** digit of the composition of the two
                         ** functions and the composition of the
                         ** transpose are calculated.
                         */

a = values[v1];
b = values[v2];

for (i = 0; i < MAX_VALS; i++)
    {
        c = a/3;         /* Quotients */
        d = b/3;
        a_mod3 = a - 3 * c; /* Next digit in each argument value
                            ** sequence. */
        b_mod3 = b - 3 * d;
                            /* Next digit in composition value
                            ** sequence. */

        temp = functor[BASE * a_mod3 + b_mod3];
        index = index + three[i] * temp;

                            /* Next digit in composition of
                            ** transpose. */

        temp = functor[BASE * b_mod3 + a_mod3];
        index1 = index1 + three[i] * temp;

        a = c;
        b = d;

    }
```

```
                                /*  Index now holds the decimal
                                ** equivalent of the composition value
                                ** sequence; index1 holds the decimal
                                ** equivalent of the transpose.
                                */

    if ( tally [index] = = -1 ) /*  Function not previously generated  */
        {
            values[next_col] = index;/*  Place function on list of functions
                                     ** generated-so-far. */

            tally[index] = level;      /*  Enter radius of newly-generated
                                       ** function. */

            (*counter)--;              /*  Decrement count of functions yet to
                                       ** be generated. */

            next_col + +;              /*  Point to next open column in values
                                       ** array. */
        }
                                /*  If different from the function formed
                                ** by the first composition, check if the
                                ** function formed by the transpose is
                                ** new.  If so, update as above.
                                */
    if(index1 ! = index)
        if (tally[index1] = = -1) /*  Function not previously generated.  */
            {
                values[next_col] = index1;
                tally[index1] = level;
                (*counter)--;
                next_col + +;
            }

}
/*  Compute the radius of the complete set of functions by finding the
** largest number of compositions required to generate a function in E(3).
** Also return the average number of compositions required to generate all
** functions.
**
**  ARGUMENTS
**     *largest    - a pointer to variable "max", allowing it to be updated by
**                   side effects.
**     size        - the total number of functions in the space
```

```
**
** RETURNS
**    the average number of compositions required to generate all functions
*/
float stat (int *largest, int size)
{
  int i;
  long int total = 0L;

  *largest = 0;
                                /* Search "tally" for largest radius */
  for (i = 0; i < size; i++)
  {
     total += tally[i];         /* Accumulate total */
     if (tally[i] > *largest)
     *largest = tally[i];
  }
  return (double) total/size;
}
```

APPENDIX B

The radii of Sheffer functions for Muzio's Type I and II
(* = symmetric functions)

value sequence	decimal index	type	radius
100022210	6798	II	7
101020210	7473	II	7
101021210	7500	II	7
101022210	7527	II	7
101122010	7752	II	7
101122210	7770	II	7
111021210	9687	II	7
112120210	10632	II	7
120001200	10980	I	7
120001210	10983	I	7
120002200	11007	I	7
120002210	11010	I	7
120101210	11226	I	7
120102010	11235	I	7
120102210	11253	I	7
120201020	11454	I	7
120201100	11457	I	7
120201110	11460	I	7
120201200	11466	I	7
120201210	11469	I	7
120201220	11472	I	7
120202210	11496	I	7
121001210	11712	I	7
121002010	11721	I	7
121002110	11730	I	7
121002210	11739	I	7
121020210	11847	II	7
121022210	11901	II	7
121102010	11964	I	7
121102110	11973	I	7
121102210	11982	I	7
121201120	12192	I	7
121201210	12198	I	7

value sequence	decimal index	type	radius
121202210	12225	I	7
122001210	12441	I	7
122002210	12468	I	7
122101210	12684	I	7
122102210	12711	I	7
122201220	12930	I	7
100021200	6768	II	8
100021210	6771	II	8
100021220	6774	II	8
100022220	6801	II	8
100122210	7041	II	8
101021200	7497	II	8
101022010	7509	II	8
101022200	7524	II	8
101120010	7698	II	8
110020200	8928	II	8
110020210	8931	II	8
110120210	9174	II	8
111021100	9675	II	8
111021200	9684	II	8
111022210	9714	II	8
112020100	10377	II	8
112020200	10386	II	8
112020210	10389	II	8
112120010	10614	II	8
112120020	10617	II	8
112220100	10863	II	8
112220210	10875	II	8
120000210	10956	I	8
120001010	10965	I	8
120001020	10968	I	8
120001110	10974	I	8
120001220	10986	I	8
120002010	10992	I	8
120002220	11013	I	8
120100010	11181	I	8
120100210	11199	I	8
120101010	11208	I	8
120101200	11223	I	8
120102000	11232	I	8
120102020	11238	I	8

value sequence	decimal index	type	radius
120102110	11244	I	8
120102200	11250	I	8
120102220	11256	I	8
120200010	11424	I	8
120200210	11442	I	8
120202110	11487	I	8
121000210	11685	I	8
121001010	11694	I	8
121001200	11709	I	8
121002100	11727	I	8
121002200	11736	I	8
121002220	11742	I	8
121022200	11898	II	8
121100010	11910	I	8
121100210	11928	I	8
121101210	11955	I	8
121102020	11967	I	8
121102120	11976	I	8
121102200	11979	I	8
121102220	11985	I	8
121200110	12162	I	8
121200210	12171	I	8
121201220	12201	I	8
122000210	12414	I	8
122001010	12423	I	8
122001020	12426	I	8
122001110	12432	I	8
122001120	12435	I	8
122001200	12438	I	8
122001220	12444	I	8
122002120	12462	I	8
122100210	12657	I	8
122101010	12666	I	8
122101110	12675	I	8
122101220	12687	I	8
122102010	12693	I	8
122102110	12702	I	8
122200210	12900	I	8
100120010	6969	II	9
100122010	7023	II	9
110120200	9171	II	9

value sequence	decimal index	type	radius
111020200	9657	II	9
111022200	9711	II	9
111120200	9900	II	9
111120210	9903	II	9
112020120	10383	II	9
112120220	10635	II	9
112220010	10857	II	9
112220020	10860	II	9
112220120	10869	II	9
112220200	10872	II	9
120001000	10962	I	9
120001100	10971	I	9
120001120	10977	I	9
120002000	10989	I	9
120002020	10995	I	9
120002100	10998	I	9
120002110	11001	I	9
120101020	11211	I	9
120101220	11229	I	9
120102100	11241	I	9
120200110	11433	I	9
120201010	11451	I*	9
121001110	11703	I	9
121001120	11706	I	9
121001220	11715	I	9
121002120	11733	I	9
121020200	11844	II	9
121101010	11937	I	9
121102000	11961	I	9
121102100	11970	I	9
121201110	12189	I*	9
121201200	12195	I	9
121202220	12228	I	9
122001100	12429	I	9
122002010	12450	I	9
122002110	12459	I	9
122100010	12639	I	9
122100020	12642	I	9
122101020	12669	I	9
122101120	12678	I	9
122101200	12681	I	9
122102020	12696	I	9

value sequence	decimal index	type	radius
122102120	12705	I	9
122201210	12927	I*	9
100122020	7026	II	10
101020200	7470	II	10
101021010	7482	II	10
101022000	7506	II	10
110020220	8934	II	10
110120010	9156	II	10
111020210	9660	II	10
111021000	9666	II	10
111021010	9669	II	10
111021110	9678	II	10
111120010	9885	II	10
112020000	10368	II	10
112020110	10380	II	10
112120110	10623	II	10
112120120	10626	II	10
112120200	10629	II*	10
112220000	10854	II	10
112220110	10866	II	10
120000010	10938	I	10
120000200	10953	I	10
120000220	10959	I	10
120002120	11004	I	10
120100020	11184	I	10
120101000	11205	I	10
120101100	11214	I	10
120101110	11217	I	10
120102120	11247	I	10
120200020	11427	I	10
120201120	11463	I	10
120202100	11484	I	10
120202120	11490	I	10
120202200	11493	I	10
120202220	11499	I	10
121002020	11724	I	10
121020110	11838	II	10
121100220	11931	I	10
121101110	11946	I	10
121101200	11952	I	10
121200120	12165	I	10

value sequence	decimal index	type	radius
121202200	12222	I	10
122000010	12396	I	10
122002020	12453	I	10
122002100	12456	I	10
100022200	6795	II	11
100120020	6972	II	11
100121020	6999	II	11
101020010	7455	II	11
110020100	8919	II	11
111020100	9648	II	11
111022010	9696	II	11
112020010	10371	II	11
112020220	10392	II	11
120100200	11196	I	11
120100220	11202	I	11
120200100	11430	I	11
120200120	11436	I	11
121000010	11667	I	11
121000110	11676	I	11
121001020	11697	I	11
121001100	11700	I	11
121002000	11718	I	11
121020100	11835	II	11
121100020	11913	I	11
121101120	11949	I	11
122000110	12405	I	11
122002200	12465	I	11
122002220	12471	I	11
122100220	12660	I	11
122102000	12690	I	11
100021010	6753	II*	12
100021020	6756	II	12
101020110	7464	II	12
101021000	7479	II	12
101220000	7938	II	12
101222200	8010	II	12
110020120	8925	II	12
110120220	9177	II	12
110220010	9399	II	12
110220200	9414	II	12

value sequence	decimal index	type	radius
110220210	9417	II	12
111220010	10128	II	12
111220100	10134	II	12
111220200	10143	II	12
112020020	10374	II	12
120101120	11220	I	12
120200200	11439	I	12
120200220	11445	I	12
120202020	11481	I*	12
121000120	11679	I	12
121000220	11688	I	12
121100110	11919	I	12
121100120	11922	I	12
121101220	11958	I	12
121200220	12174	I	12
121202120	12219	I*	12
122000120	12408	I	12
122100110	12648	I	12
122101000	12663	I	12
122102100	12699	I	12
122102200	12708	I	12
122102220	12714	I	12
100020010	6726	II	13
102200100	8514	I	13
110120020	9159	II	13
110220020	9402	II	13
110220100	9405	II	13
110220120	9411	II	13
111220210	10146	II	13
120000020	10941	I	13
120000110	10947	I	13
120100000	11178	I	13
122000220	12417	I	13
122001000	12420	I	13
122100120	12651	I	13
122101100	12672	I	13
100020020	6729	II	14
101021110	7491	II*	14
110020010	8913	II	14
110020020	8916	II	14

value sequence	decimal index	type	radius
111120110	9894	II	14
120000100	10944	I	14
120100110	11190	I	14
121001000	11691	I	14
121100000	11907	I	14
100022020	6783	II*	15
110020000	8910	II	15
111020110	9651	II	15
111022000	9693	II	15
111220000	10125	II	15
112220220	10878	II	15
120100100	11187	I	15
120200000	11421	I*	15
121101020	11940	I	15
121101100	11943	I	15
122002000	12447	I	15
122100000	12636	I	15
122200220	12903	I	15
122202220	12957	I*	15
101200100	7785	I	16
101200200	7794	I	16
101222000	7992	II	16
110120100	9162	II	16
110120110	9165	II	16
111020000	9639	II	16
120000120	10950	I	16
121000020	11670	I	16
121101000	11934	I	16
122000020	12399	I	16
101020000	7452	II	17
101020100	7461	II*	17
111020010	9642	II	17
111220110	10137	II	17
120100120	11193	I	17
110020110	8922	II	18
110220110	9408	II	18
111120100	9891	II*	18
120000000	10935	I	18

value sequence	decimal index	type	radius
100020000	6723	II*	19
110220000	9396	II	19
110120120	9168	II	20
110120000	9153	II*	21
102200200	8523	I	24
110220220	9420	II	25

BIBLIOGRAPHY

Bochenski, J. M. *A Precis of Mathematical Logic*. Dordrecht, Holland: D. Reidel. 1959, p. 17.

Foxley, E. "The determination of all Sheffer functions in 3-valued logic, using a logical computer." *Notre Dame J. of Formal Logic*, vol. 3, no. 1, January 1962, pp. 41-50.

Kalicki, J. "A test for the existence of tautologies according to many-valued truth tables." *J. Symbolic Logic*, vol. 15, no. 3, September 1950, pp. 182-184.

Martin, N. M. "Some analogues of the Sheffer stroke function in *n*-valued logic." *Indagationes Mathematicae*, vol. 12, 1950, pp. 393-400.

_____. "The Sheffer functions of 3-valued logic." *J. Symbolic Logic*, vol. 19, no. 1, March 1954, pp. 45-51.

Muzio, J. C. "A decision process for 3-valued Sheffer functions I." *This Zeitschr.*, vol. 16, 1970, pp. 271-280.

_____. "Binary functions that are complete with constants over {0,1,2}." *Proceedings of The Fifth Manitoba Conference on Numerical Mathematics*, 1975a, pp. 561-570.

_____. "A complete classification of two-place Sheffer functions in 3-valued logic." *The University of Manitoba Scientific Report No. 80*, July 1975b.

Muzio, J. C. and Wesselkamper, T. C. *Multiple-valued switching theory*. Boston: Adam Hilger Ltd. 1986.

Piccard, S. "Sur les fonctions dé finies dans les ensembles finis quelconque." *Fundamenta Mathematicae*, vol. 24, 1935, pp. 298-301.

Post, E. L. "Introduction to a general theory of elementary propositions." *Amer. J. of Mathematics*, vol. 43, 1921, pp. 163-185.

Rose, A. "Review of a paper by Martin." *J. Symbolic Logic*, vol. 16, 1951, pp. 275-276.

Rosenberg, I.G. "La structure des fonctions de plusieurs variables sur un ensemble fin." *Compte Rendu de Acad. Sci Paris*, 1965, pp. 3817-3819.

Rousseau, G. "Completeness in finite algebras with a single operation." *Proc. Amer. Math. Soc.*, vol. 18, 1967, pp. 1009-1013.

Salomaa, A. "On the compositions of functions of several variables ranging over a finite set." *Annales Universitates Turkuensis*, ser. AI, vol. 41, 1960, pp. 7-48.

Schofield, P. "Independent conditions for completeness of finite algebras with a single operator." *J. London Math. Soc.*, vol. 44, 1969, pp. 413-423.

Sheffer, H. M. "A set of five independent postulates for boolean algebras, with application to logical constants." *Trans. Amer. Math. Soc.*, vol. 14, 1913, pp. 481-488.

Słupecki, J. "Kryterium pełności wielowartościowych systemów logiki zdań." *Comptes Rendus Des Séances de la Société des Sciences et des Lottres de Varsovie*, vol. 32, 1939, pp. 102-109. Reprinted in *Studia Logica*, vol. XXX, 1972, pp. 153-157.

Swift, J. D. "Algebraic properties of *n*-valued propositional calculi." *Amer. Math. Monthly*, vol. 59, no. 9, 1952, pp. 612-621.

Webb, D. L. "Generation of any *n*-valued logic by one binary operation." *Proc. National Academy of Science*, vol. 21, 1935, pp. 252-254.

Webb, D. L. "Definition of Post's generalized negative and maximum in terms of one binary operation." *Amer. J. of Math.*, vol. 58, 1936, pp. 193-194.

Wesselkamper, T. C. and Cabrasawan, F. J. "Searching for complete functions over E(3) with small radius." *Proc. The Twenty-Fourth International Symposium on Multiple-Valued Logic*, 1994, pp. 172-176.

Wesselkamper, T. C.; Lesniak, J.; and Cabrasawan, F. J. "Genetic Algorithm techniques for 3-valued transistor design. *Proc. of The First IEEE Conference on Evolutionary Computation*, vol. II, June 1994, pp. 656-660.

Wheeler, R. F. "Complete connectives for the 3-valued propositional calculus." *Proc. London Math. Soc.*, vol. 16, 1966, pp. 167-192.