Finite State Models in the Study of Comma-Free Codes

John A. Llewellyn
Department of Computer Studies, University of Lancaster, Bailrigg, Lancaster, UK

The definition of comma-free codes is extended to be more comprehensive and include codes with variable length codewords and codes with an unlimited number of codewords. Finite state recognizers are used to represent comma-free codes in a compact form, making available the well established methods of manipulating and analysing finite state models. An efficient algorithm is presented to test the comma-freeness of any given code represented by a finite state recognizer. It is well known that any maximal comma-free code with codewords of odd fixed length can be achieved. Although upper limits have been given for even fixed length it has not been established which comma-free codes of this nature can achieve the upper limit. Compact representations of comma-free binary codes are given for variable length codewords with maximum even fixed lengths (6, 8 and 10) and in one case (length not greater than 6) it is seen that the corresponding upper limit for fixed lengths can be exceeded.

1. INTRODUCTION

1.1 Synchronizing codes

A synchronizing code consists of a directory of codewords, from a fixed size alphabet, such that, on receipt of a sequence of symbols from the alphabet, it will be possible after some delay to synchronize and identify the first symbol of a valid codeword. Thereafter the receiver will remain in synchronization and all subsequent codewords can be identified without delay, assuming that the message is error free and only valid codewords are received.

The delay before synchronization is achieved may have a finite maximum (Example 1.1.1) or it may be unlimited (Example 1.1.2). There are codes which are not synchronizing. For the code directory of Example 1.1.3 it can be shown that, for any sequence of symbols received there are at least two alternative interpretations and that synchronization is not possible without external intervention.

1.1.1 Example of a code with finite synchronizing delay

$$S = (xyyxy, yxy\}$$

Consider the sequence $\ldots x\,y\,x\,y\,y\,x\ldots$ which could be interpreted as $\ldots x\,/\,y\,x\,y\,/\,y\,x\ldots$ or $\ldots x\,y\,/\,x\,y\,y\,x\ldots$. The next symbol received will enable the receiver to resolve the ambiguity. It can be shown that this is the worst case and that synchronization can always be achieved within 7 symbols for this example.

1.1.2 Example of a code with unlimited synchronization delay

$$S = (xxyx, xyx, yxy\}$$

Consider the sequence $\ldots x\,y\,x\,(y\,x)^{3n}\,x\ldots$ for $n > 0$, which could be interpreted as $\ldots (x\,y\,x\,/\,y\,x\,y)^n\,/\,x\,y\,x\,/\,x\ldots$ or $\ldots x\,y\,/\,(x\,y\,x\,/\,y\,x\,y)^n\,/\,x\,x\ldots$ The next symbols must be xyx or yx, after which the start of the next codeword will be received. Synchronization is achieved after $6n + 7$ symbols, where n is a positive unlimited integer.

1.1.3 Example of a code which cannot be synchronized.

Consider the set of codewords from the alphabet $(x, y\}$, which contain any number of occurrences of x and exactly two y symbols one of which must be the last of the codeword.

$$S = (yy, xyy, yxy, xxyy, xyxy, yxxy, \ldots\ )$$

or

$$S = (x^n y x^n y \,|\, n > -1)$$

It can be shown that for any possible sequence of valid codewords there are at least two possible alternative points of synchronization.

1.2 Comma codes

A comma code is a directory of codewords in which one (or more) symbols of the alphabet is designated as a codeword delimiter or separator and cannot occur within a codeword. A generalization of a comma code can be designed in which one (or two) groups of symbols, the prefix, acts as separator between codewords and the set of codewords is designed such that the sequence of symbols corresponding to the prefix(es) cannot occur within a codeword or in the concatenation of a prefix and a codeword as in the example:

$$S = (xxx, yxy \,|\, \text{prefix} = xxy).$$

Guibas and Odlyzko[1] discuss the choices of prefixes to maximise the number of codewords in a fixed length prefix code.

1.3 Comma-free codes

A comma-free code is a directory of codewords such that, for any sequence of symbols, synchronization can be achieved within at most (2 × longest codeword − 1) symbols.

CCC-0010-4620/83/0026-0303$04.50

Expressed alternatively: as a code in which a complete codeword can be identified as soon as its last symbols is received. To achieve this, the set of codewords must satisfy the condition that a set of symbols corresponding to a valid codeword cannot occur within another codeword nor within the concatenation of two valid codewords.

The formal definition, for fixed length codewords by Golomb *et al.*[2] has been used as the basic definition by most authors in the field:

> A set D of k-letter codewords is called a comma-free dictionary if whenever $(a_1, a_2, \ldots a_k)$ and $(b_1, b_2, \ldots b_k)$ are in D, the overlaps $(a_2, a_3, \ldots a_k, b_1)$, $(a_3, \ldots a_k, b_1, b_2)$, $\ldots$ $(a_k, b_1, \ldots b_{k-1})$ are not in D.

This can be extended to include code directories containing other than fixed length codewords:

> A set of codewords, from a finite alphabet $\{a_i | i < n + 1\}$ is a comma-free dictionary if:
> (i) whenever $(a_1, a_2, \ldots a_k)$ is a codeword then the sequences $(a_1, a_2, \ldots a_p | p < k)$, $(a_p, a_{p+1}, \ldots, a_k | 1 < p)$, $(a_p, a_{p+1}, \ldots, a_q | 1 < p, q < k)$ are not valid codewords:
> (ii) whenever $(a_1, a_2, \ldots, a_k)$ and $(b_1, b_2, \ldots, b_m)$ are codewords then the sequences $(a_p, \ldots, a_k, b_1, \ldots, b_q | 0 < p < k, 1 < q \leq m)$ are not valid codewords.

1.3.1 Example of a fixed word length comma-free code. The set of codewords consisting of n binary symbols such that a codeword contains at least one of each symbol and all 0 symbols precede all 1 symbols. E.g. for $n = 5$ the 5 bit codewords are defined by

$$S = (00001, 00011, 00111, 01111)$$

1.3.2 Example of a variable length comma-free code. The set of codewords consisting of a starting symbol a, ending with symbol c and one or more b symbols between the a and c symbols.

$$S = (abc, abbc, abbbc, \text{etc.})$$

1.3.3 Example of a code which is not comma-free.

$$S = (010101, 100111, 110011, 000001, 011001)$$

The example illustrates three types of situation which violate the requirements for comma-freeness.

(a) A codeword is contained within the string obtained by concatenating two codewords.

011001 is contained in 000001 100111

(b) The special case in which a codeword is contained within the string obtained by concatenating two copies of a codeword; this corresponds to a cyclic shift.

100111 is contained in 110011 110011

(c) The special case in which a codeword is contained within the string obtained by concatenating two copies of itself; this corresponds to a repeated subsequence of symbols.

010101 is contained in 010101 010101

2. PREVIOUS WORK

The concept of comma-free coding was first discussed in the literature by Crick[3] in connection with the coding of amino acid chains in genetic coding. Golomb, Welch and Delbruck[4] develop the ideas further and lay a foundation for the formal development of comma-free codes and Golomb *et al.*[2] establishes that the maximum number of codewords in a fixed length comma-free directory has an upper limit defined by:

$$L \leq \frac{1}{k} \sum \{U(d) \times n^{k/d}\}$$

where the summation is over all divisors, d of k and $U(d)$ is the Mobius function defined by:

$$U(d) = \begin{cases} 1 \text{ if } d = 1 \\ 0 \text{ if } d \text{ has a square factor} \\ (-1) \text{ if } d = p_1 p_2 \ldots p_r \text{ where } p_1, p_2, \ldots, p_r \text{ are distinct primes.} \end{cases}$$

Upper limits for the sizes of binary comma-free codes with fixed length codewords are shown in the following table:

word length	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
directory size	1	2	3	6	9	18	30	56	99	186	335	630	1161	2192	4080

Jiggs[4] reports a theorem by Jewett to show that the upper bound cannot be achieved for certain classes of codes with even fixed codeword lengths and Eastmann[6] demonstrates that the upper limit can be achieved for codes with odd fixed codeword lengths, presenting an algorithm to generate such codes. Taylor and Green[7] gives restrictions on the selection of codes from non-degenerate equivalence classes in order to achieve maximal comma-free codes. To date, whether the upper limit can be achieved for comma-free codes of even fixed length is unsolved, although it has been shown that in certain cases the upper limit cannot be achieved.[5] Yoji[8] demonstrates that the upper limit can be achieved for a binary comma-free code with codewords of length 10, but abandoned the investigation of length 12 as too exhausting.

Examples will be given later in this paper in which the upper limit is achieved by also including codewords of length less than the 'fixed codeword length'; and in fact the upper limit is exceeded in one case. Scholtz[9] presents algorithms to generate codes with variable word lengths, but allows the possibility of shorter codewords occurring within the longer codewords of a set. In this paper, this will not be allowable and the definition of comma-freeness will be much stricter, as defined in the previous section. Stiffler,[10] in a comprehensive treatment of synchronous communication theory, presents a summary of comma-free codes and further development, including error correction properties.

Finite state machines, regular grammars, regular algebra and the interrelation between them have been extensively studied. There are many texts which introduce and develop the subject.[11–14] The purpose of this paper is to show how comma-free codes can be represented economically by the use of finite state models and how well established techniques can be used in the analysis of those models. In particular an efficient algorithm is presented which will establish if a code is

comma-free or not, and it is demonstrated that the model can be used as the basis for the design of a decoding system. Variable length binary codes, which it is believed have not previously been published, are presented as illustrations of the use of finite state models.

3. FINITE STATE RECOGNIZERS

3.1 Representation of code directories

To define a code directory, referred to simply as a code in what follows, it is required to specify the alphabet of symbols and the set of combinations of the symbols which make up the codewords which are valid for the code under consideration. A code can be specified in many ways. Examples 1.3.1 and 1.3.2 are used as illustrations in the following.

(i) As a list of valid codewords, possibly with some method of generalization for directories which are large or of unlimited size.

$D1 = \{00001, 00011, 00111, 01111\}$ or $(0^{5-k}1^k \mid 0 < k < 5\}$

$D2 = (abc, abbc, abbbc, \ldots, \text{etc})$ or $(ab^kc \mid 0 < k\}$

(ii) As a set of rules which define the structure of the code; a grammar which in the codes considered will be a regular grammar (classified as Chomsky type 3).

$G1 = S \to 0A, \quad A \to 0B, \quad A \to 1C$
$B \to 0D, \quad B \to 1E, \quad C \to 1E$
$D \to 0F, \quad D \to 1F, \quad E \to 1F, \quad F \to 1$

$G2 = S \to aA, \quad A \to bB, \quad B \to bB, \quad B \to c$

(iii) As an algebraic formula which defines the strings of the 'language' of the code; in this case using regular algebra.

$R1 = 00001 + 00011 + 00111 + 01111$
$= 0(00(0 + 1) + (0 + 1)11)1$

$R2 = abb^*c = ab^*bc$ (* is the closure operator).

3.2 Concept of states, alphabets and recognizers

A finite state machine (Moore type) is an automata model defined by

(i) a finite state alphabet
(ii) a finite alphabet of input symbols
(iii) a finite alphabet of output symbols
(iv) a function defining the next state for each input/state pair
(v) a function defining the output symbol for each state.

An alternative representation of a finite state machine (Mealy type) differs only in that the output function is defined for each input/state pair. It can be shown that the two types of model are equivalent and there are standard techniques for converting between the two types of model.

A finite state recognizer is a Moore type model in which one (or more) state is designated as an 'accept' state, in which the output is implicit. Given a 'start' state, any sequence of symbols which causes transition from the start state to an accept state is said to be accepted by the machine. The next-state function, which defines the set of strings from the input alphabet which are accepted by the machine in question, can be represented in the following ways:

(a) in formal terms as a set of 'triples'
(b) in tabular form as a rectangular finite state table
(c) in diagrammatic form as a finite state graph.

The set of strings or codewords making up a code directory can be specified by means of a finite state recognizer.

A finite state model to control a sequence of actions on an environment, which is external to the finite state model itself, can be represented by either a Mealy or a Moore type machine, in which the outputs are signals to initiate the appropriate action, and any such sequences of initiated actions are assumed not to overlap in time. In the design of a decoding device, the external environment could be the required decoded message and the called actions the arithmetic or logical operations on that message in the course of its reconstruction. An example is shown in sections 5.1 and 5.5.

3.3 Minimal forms of finite state recognizers

There is a well established body of knowledge on the analysis and minimal representations of finite state machines. Any completely defined finite state machine can be shown to be equivalent to a unique (except for state labelling) form with a minimum number of states. There are well established efficient algorithms for achieving the unique canonical form. Finite state machines which are not completely defined may correspond to models which have 'don't care' situations, or there may be sequences of inputs which cannot possibly occur. A unique minimal form may not always be possible, but there are techniques available for identifying possible redundant states and producing compact 'compatible' representations.[11,12]

3.4 Deterministic and non-deterministic recognizers

In a deterministic finite state machine, for every state/input combination there is a unique successor state. There is also a unique output signal, associated with every state/input combination for a Mealy machine and associated with every state for a Moore machine. In a non-deterministic machine, there may be more than one possible successor state: as in a recognizer, when the model is not prescribing a required set of behaviours, but rather specifying a set of possible behaviour patterns which are acceptable. In the case of a Moore machine there are standard techniques available for converting into an equivalent deterministic finite state machine, except that in some cases there may be some non-deterministic output inherent to the problem itself. For example in constructing a recognizer for the two sequences $\{abc, abcb\}$, when the sequence abc has been received, it will not yet be clear whether to output an 'accept' signal or not. A technique to eliminate non-determinism (except for output situations alluded to) is

based on the concept of a composite state: 'the model is in state1 or state2 . . .'.

Subsequent states are defined in a similar manner, and it can be shown that the number of possible composite states is strictly limited, resulting in a finite state machine.

When constructing the finite state machine representing the reverse of a (deterministic) recognizer, for every state and input combination it is required to define the 'predecessor state'. In general there will be more than one predecessor state resulting in a non-deterministic recognizer for the reverse machine. The model can be converted to deterministic form using standard algorithms.

3.5 Code directories as finite state recognizers

A code can be represented by a finite state recognizer, together with an identification of the 'start' and 'accept' states.

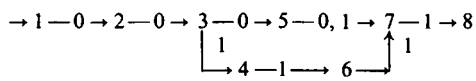
3.5.1. The code presented in the example of Section 1.3.1 and represented by $D1$, $G1$ and $R1$ above can be expressed precisely and concisely as follows: The start state is 1 and the accept state is 8.

(a) Set of triples: (1,0,2) (2,0,3) (2,1,4) (3,0,5) (3,1,6) (4,1,6) (5,0,7) (5,1,7) (6,1,7) (7,1,8)

(b) Finite state table:

inputs	states						
	1	2	3	4	5	6	7
0	2	3	5	–	7	–	–
1	–	4	6	6	7	7	8

(c) Finite state graph:



(d) Finite state model for the reversed code: The start state is 8 and the accept state is 1.

inputs	states						
	8	7	5	5/6	3/4	3	2
0	–	5	3	3	2	2	1
1	7	5/6	–	3/4	2	–	–

replace 5/6 by 6, 3/4 by 4

inputs	states						
	8	7	6	5	4	3	2
0	–	5	3	3	2	2	1
1	1	6	4	–	2	–	–

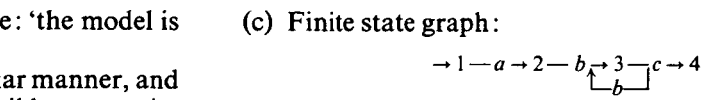
3.5.2. The code presented in the example of section 1.3.2 and represented by $D2$, $G2$ and $R2$ above can be expressed precisely and concisely as follows: The start state is 1 and the accept state is 4.

(a) Set of triples: (1, a, 2) (2, b, 3) (3, b, 3) (3, c, 4)

(b) Finite state table:

inputs	states		
	1	2	3
a	2	–	–
b	–	3	3
c	–	–	4

(c) Finite state graph:



(d) Finite state model for the reversed code: The start state is 4 and the accept state is 1.

inputs	states			
	4	3	2	2/3
a	–	–	1	1
b	–	2/3	–	2/3
c	3	–	–	–

replace 2/3 by 5

inputs	states		
	4	3	5
a	–	–	1
b	–	5	5
c	3	–	–

3.6 Codeword sequences

When a sequence of codewords is represented as a continuous sequence of symbols, the start state will also be the accept state. The set of codewords making up a code can be represented as a deterministic finite state recognizer, except when the code is not instantaneous—i.e. a valid codeword is the prefix of another valid codeword, in which case the model will be inherently non-deterministic.

A finite state recognizer can be used as the basis of a 'table driven' parser. Once synchronization has been achieved the model will be in the start/accept state at the beginning of a codeword. Successive states are selected from the table according to the symbols input. A sequence of symbols is recognized as a valid codeword, i.e. a comma can be inserted into the symbol sequence, whenever the start/accept state is entered.

4. COMMA-FREE CODES

4.1 Conditions and tests for comma free codes

4.1.1. On receiving a sequence of symbols the recognizer will be in start state, S, at the beginning of a valid codeword. It will pass through a sequence of states, including states T and U (say) and then return to accept state S after the last symbol of the n symbol codeword.

$$S—a_1 \ldots a_j \rightarrow T—a_{j+1} \ldots a_{j+k} \rightarrow U—a_{j+k+1} \ldots a_n \rightarrow S$$

If there is a k symbol codeword in the code which is contained within the original codeword, then there will be a sequence of symbols, say $a_{j+1} \ldots a_{j+k}$, which will result in transition from the start state S to the accept state S:

$$S—a_{j+1} \ldots a_{j+k} \rightarrow S$$

In this case the code does not satisfy the criteria for comma-freeness and there exists a sequence $a_{j+1} \ldots a_{j+k}$ which gives rise to two corresponding sets of transitions, with initial states S and T, represented by:

$$\frac{S}{T}—a_{j+1} \ldots a_{j+k} \rightarrow \frac{S}{U}$$

The case $j = 0$ corresponds to a non-instantaneous code in which a valid codeword is the prefix of another valid codeword. The case $j = n - k$ corresponds to a code in which a valid codeword occurs as the suffix of another valid codeword.

4.1.2. Consider a code in which there are codewords as follows, for which S is the start/accept state and T, U and V are intermediate states:

(a) A codeword $a_1 \ldots a_j a_{j+1} \ldots a_n$ for which corresponding state transitions are:

$$S — a_1 \ldots a_j \rightarrow T — a_{j+1} \ldots a_n \rightarrow S$$

(b) A codeword with suffix $a_1 \ldots a_j$ for which corresponding state transitions are:

$$U — a_1 \ldots a_j \rightarrow S$$

(c) A codeword with prefix $a_{j+1} \ldots a_n$ for which corresponding state transitions are:

$$S — a_{j+1} \ldots a_n \rightarrow V$$

In this case the code does not satisfy the criteria for comma-freeness and there exists a sequence $a_1 \ldots a_n$ which gives rise to two corresponding sets of transitions, with initial states S and U,

$$\frac{S}{U} — a_1, a_2, \ldots, a_j \rightarrow \frac{T}{S} — a_{j+1}, a_{j+2}, \ldots, a_n \rightarrow \frac{S}{V}$$

4.1.3. In each of the cases considered in 4.1.1 and 4.1.2 it is seen that if the code is not comma-free, then there is a sequence of symbols representing a codeword giving rise to two sequences of transitions:

$$\frac{\text{Start state}}{\text{State 1}} — \langle\text{symbol sequence}\rangle \rightarrow \frac{\text{Start/accept state}}{\text{State 2}}$$

where State 1 is an initial state, other than the start state.

Conversely, for a finite state recognizer which represents a comma-free code, there is no such sequence of transitions, for any possible initial state other than the start state.

4.2 Algorithm to determine if a code is comma-free

The previous section gives the basis of a test to determine if a finite state recognizer represents a comma-free code, or alternatively, a proof of comma-freeness for a given code.

4.2.1. Define:

$(S_0, S_1, \ldots, S_n)$: states of the recognizer
S_0: start state
A: input symbol string
H: successor state to S_0 with respect to A
G: set of successor states to $S_1, \ldots, S_n$ with respect to A.

It is required to determine, in order to establish comma-freeness, that there is no sequence of input symbols which satisfies the above condition for all possible start states $S_1, S_2, \ldots, S_n$, i.e.

$$\frac{H = S_0}{G = (S_1, S_2, \ldots, S_n)} — A \rightarrow \frac{H = S_0}{G''}$$

where G'' is not the empty set.

This can be expressed as a recursively called procedure which is initially called *CFtest* (S_0, (S_1, S_2, . . ., S_n), *NULLSYMBOL*), and will set the global variable *Commafree* (initially set true) to false on establishing a symbol string A which violates the condition required of commafree codes.

Procedure *CFtest* (H, G, A)
{H: successor state to S_0 w.r.t. symbol string A.}
{G: set of successor states to $S_1, S_2, \ldots, S_n$ w.r.t. A.}
{Initially $H = S_0$ and $G = (S_1, S_2, \ldots, S_n)$.}
if {($H = S_0$) *and* (A *not NULLSTRING*)}
then if {G *not NULLSET*}
  then set *COMMAFREE* **to** *false*
else for X **inrange** {*code alphabet*} **repeat**
  replace H **by** {*successor state of* H *w.r.t. symbol* X}
  if {H *not NULLSET*}
  then replace G **by** {*set of successor states w.r.t.*
    symbol X, *of states of set* G}
    if {G *not NULLSET*}
    then set A **to** {*concatenation of* A *and* X}
      CFtest (H, G, A).
until {*last of code alphabet*}

4.2.2. Suppose that it is not required to identify the precise input string which violates the condition for comma-freeness. Then G can be replaced by the union of the successor sets G for input strings leading to the corresponding successor of the start state, H. This gives rise to a more efficient algorithm, eliminating needless duplication of computation. The finite state recognizer is effectively extended by including an extra parameter with each state, the set of states, G, which is the set of successor states to $S_1, S_2, \ldots, S_n$ for all input strings leading to that state from the initial state. This is illustrated in tabular form in Example 4.2.3 together with the corresponding tree diagram.

4.2.3 Example to show that the code (*baaa, baab, bacc, bbac*) is not a comma-free code. The corresponding finite state recognizer with state set (S, 1, 2, 3, 4, 5), where S is the start/accept state, is represented:

inputs	states					
	S	1	2	3	4	5
a	–	2	4	5	S	–
b	1	3	–	–	S	–
c	–	–	5	–	–	S

The sequence of H/G when calling *CFtest* (S, [1, 2, 3, 4, 5], *NULL*) is:

$$\frac{S}{1,2,3,4,5} — b \rightarrow \frac{1}{3,S} \begin{cases} a \rightarrow \frac{2}{5} \begin{cases} a \rightarrow \frac{4}{-} \\ c \rightarrow \frac{5}{S} — c \rightarrow \frac{S}{-} \end{cases} \\ b \rightarrow \frac{3}{1} — a \rightarrow \frac{5}{2} — c \rightarrow \frac{S}{5} \end{cases}$$

The sequence *bbac* results from suffix *b* and prefix *bac*.

In tabular form, with an extra column for the set of successor states G:

H-states	Successor states			G-sets
S	–	1	–	1, 2, 3, 4, 5
1	2	3	–	3, S
2	4	–	5	5
3	5	–	–	1
4	S	S	–	–
5	–	–	S	S, 2
S	as above			5
inputs	a	b	c	

The non-null set {5}, in the row corresponding to the accept state S, shows that there is a sequence:

$$\frac{S}{G} \rightarrow string \rightarrow \frac{S}{5}$$

and consequently the code is not comma-free.

5. EXAMPLES OF COMMA-FREE CODE DIRECTORIES

The first three examples illustrate binary comma-free codes in which some codeword lengths are shorter than the 'fixed length' assumed in calculating the upper limit to the size of the code. They have been designed using finite state recognizer generating methods based on the construction methods of Scholtz.[9] All are assumed to have start/accept state 1. The G-sets, demonstrating comma-freeness, are given alongside the finite state recognizer.

5.1 Example of a comma-free code directory of size 10

Although 9 is the upper bound for the size of a binary comma-free code with codewords of length 6,[5] by considering variable length codewords with maximum length 6, a code can be designed of size 10 (Table 1).

Table 1.

state	0	1	G-sets
1	2	–	2 3 4 5 6 7 8
2	–	3	1 4 6 8
3	4	5	1
4	6	–	2
5	6	7	–
6	8	1	–
7	8	8	–
8	1	1	–
1	2	–	–

The null G-set associated with accept state 1 shows that the recognizer represents a comma-free code. The set of codewords is: {01001, 01101, 010000, 010001, 011000, 011001, 011100, 011101, 011110, 011111}.

This example also illustrates how the finite state model can be used to recognize codewords and convert them into decimal values. The transducer shown in Table 2 is obtained directly from the finite recognizer representing the comma-free code. It takes a binary sequence as input and when a valid code is recognized evaluates the corresponding decimal value in the external variable D. If an error occurs, the transducer will output an error signal and synchronize in at most one wordlength.

Table 2.

inputs	1	2	3	4	5	6	7	8	States
0	2	2	4	6	6	8	8	1	Next state
1	1	3	5	3	7	1	8	1	function
0	–	pE	–	p0	p1	p2	p2	p5	Output
1	pE	–	–	pE	p3	p4	–	p6	function

Procedures acting on external variable D:

–	No action	pE	Error, not yet synchronized
p0	Set D to 0	p1	Set D to 1
p2	Replace D by $D + 1$	p3	Set D to 3
p4	Output value of D	p5	Output value of $2D$
p6	Output value of $2D + 1$		

5.2 Finite state recognizer for a binary comma-free code with codewords not longer than 8 binary symbols

There are 24 codewords of length 8 and 6 codewords of length 7 (Table 3).

Table 3.

	0	1	G-sets
1	2	–	2 13
2	–	3	1 4 6 8 11 12 13
3	4	5	1 9 12
4	6	–	1 2 13
5	6	7	1
6	8	9	2
7	8	10	
8	11	12	
9	13	–	
10	11	11	
11	12	12	
12	1	1	
13	–	12	4
1	2	–	–

The set of codewords is represented by the regular expression:

$$01\,[(00 + 1(0 + 1))(01 + 00(0 + 1)) + ((0 + 1)0101 + 111(0 + 1)(0 + 1))](0 + 1)$$

5.3 Finite state recognizer for a binary comma-free code with codewords not longer than 10 binary symbols

There are 78 codewords of length 10 and 20 codewords of length 9 (Table 4).

Table 4.

	0	1	G-sets
1	2	–	2 23
2	–	3	1 4 6 8 11 13 15 17 19 20 21 22
3	4	5	1 9 12 16 20 22 23
4	6	–	1 2 13 17 21 22
5	6	7	1 18 22
6	8	9	1 2 19
7	8	10	1
8	11	12	2
9	13	–	3 23
10	11	14	
11	15	16	
12	17	18	3
13	19	16	4
14	15	15	
15	20	20	
16	21	22	
17	19	20	4
18	19	–	5
19	–	23	6
20	22	22	
21	1	22	
22	1	1	
23	–	1	9
1	2	–	–

This is to be compared with the maximal comma-free code, with 99 binary codewords of length 10, presented by Yoji.[8] The finite state recognizer in Table 5 represents the reversal of the code presented by Yoji.

Table 5.

	0	1	*G*-sets
1	–	2	2 64
2	3	4	1 4 6 8 10 13 15 17 20 22 24 26 33 35 40 41 43 44 47 49 50 51 52 53 55 57 58 59 61 62 63 64
3	5	6	1 7 11 14 18 23 25 28 31 34 37 39 46 48 53 54 56 59 60 62 63 64
4	7	8	1 2 8 15 24 26 35 43 49 53 55 59 61 62 64
5	9	10	1 12 19 25 29 36 38 45 53 62 64
6	11	–	1 2 13 17 20 41 47 50 52 58 59 62 63 64
7	12	13	1 3 14 25 37 39 48 54 60 62 63 64
8	14	15	1 2 4 15 26 49 53 59 62 64
9	16	17	1 21 30 38 42 62
10	18	–	1 2 22 43 51 57 62
11	19	20	1 3 23 28 31 59 60 62 64
12	21	22	1 5 25 38
13	23	24	1 2 6 17 50 52 59 62 64
14	25	17	1 3 7 25 39 60 62 64
15	25	26	1 2 4 8 26 59 62 64
16	27	–	1 32 41 54
17	28	–	1 2 6 13 33 44 51 52 55 64
18	29	–	1 3 34 54 64
19	30	–	1 5 36 45 64
20	31	–	1 2 6 41 64
21	32	33	9 38
22	34	35	2 10 51
23	36	–	1 3 11 28 60 64
24	37	–	1 2 4 55 64
25	38	–	1 3 5 7 12 14 38 39 64
26	39	–	1 2 4 8 15 64
27	40	40	–
28	–	41	3 11 23 46 56 63
29	42	43	5
30	41	44	9
31	45	41	3 11
32	–	40	16
33	46	43	17 51
34	–	47	3 18
35	48	49	4 61
36	–	43	5 19
37	–	50	3 7 63
38	–	51	5 9 12 21 25
39	–	52	3 7 14 25
40	53	53	–
41	–	53	6 16 20 58 63
42	54	55	9
43	54	53	10 61
44	56	53	17
45	–	57	5 19
46	53	58	28
47	59	53	6
48	–	59	7 63
49	60	59	8
50	60	55	6 13
51	–	61	10 17 22 33
52	–	55	6 13 17
53	62	62	–
54	–	62	16 18 63
55	63	62	17 24
56	62	63	28
57	64	62	10
58	62	–	41
59	64	64	11 13 15
60	–	64	11 14 23
61	63	–	35 43
62	1	1	–
63	1	–	28 37 41 48 54
64	–	1	17 18 19 20 23 24 25 26
1	–	2	–

Table 6.

H-states		0	1	*G*-sets
	S	1	–	$1, 2, (3j, 3j+1, 3j+2 \mid 0 < j < k+1\}$
	1	–	2	$(3j, 3j+1 \quad \mid 0 < j < k+1\}$
	2	3	3	$(3j+2 \quad \mid 0 < j < k+1\}$
$0 < i < k$	$3i$	$3i+1$	$3i+2$	$(3j \quad \mid i < j < k+1\}, S$
	$3i+1$	$3i+3$	–	$(3j+1 \mid i < j < k+1\}, 1$
	$3i+2$	$3i+3$	$3i+3$	$(3j+2 \mid i < j < k+1\}$
	$3k$	$3k+1$	$3k+2$	S
	$3k+1$	S	–	1
	$3k+2$	S	S	–
	S	1	–	–

5.4 A class of fixed length binary comma-free codes of odd length which approach the maximum for smaller codeword lengths.

The virtue of this class of codes is the conciseness with which they can be represented.

As a regular expression: xy (x or y) (xx or xy or yy)^k. For example, with $k = 2$, there are 18 codewords of length 7:

$\{xyxxxxx, xyxxyxx, xyxyyxx, xyyxxxx, xyyxyxx, xyyyyxx\}$
$\{xyxxxxy, xyxxyxy, xyxyyxy, xyyxxxy, xyyxyxy, xyyyyxy\}$
$\{xyxxxyy, xyxxyyy, xyxyyyy, xyyxxyy, xyyxyyy, xyyyyyy\}$

As a finite state recognizer, where S is the start/accept state (see Table 6).

The proof of comma-freeness can be illustrated graphically as follows:

$$\frac{S}{1, 2, \{3j, 3j+1, 3j+2 \mid 0 < j < k+1\}}$$
$$\xrightarrow{x} \frac{1}{\{3j, 3j+1 \mid 0 < j < k+1\}, S}$$
$$\xrightarrow{y} \frac{2}{\{3j+2 \mid 0 < j < k+1\}}$$
$$\xrightarrow{x, y} \frac{(3i \mid i = 1\}}{S, \{3j \mid i < j < k+1\}}$$

In general:

$$\text{for } i < k \quad \frac{3i}{\{3j \mid i < j < k+1\}, S} \rightarrow$$
$$\rightarrow \begin{cases} \xrightarrow{x} \dfrac{3i+1}{\{3j+1 \mid i < j < k+1\}, 1} \xrightarrow{x} \dfrac{3i+3}{\{3j \mid i+1 < j < k+1\}} \\ \xrightarrow{y} \dfrac{3i+2}{\{3j+2 \mid i < j < k+1\}} \end{cases}$$
$$\xrightarrow{x, y} \frac{3i+3}{\{3j+3 \mid i < j < k+1\}, S}$$

$$\text{for } i = k \quad \frac{3k}{S} \rightarrow \begin{cases} \xrightarrow{x} \dfrac{3k+1}{1} \xrightarrow{x} \dfrac{S}{-} \\ \xrightarrow{y} \dfrac{3k+2}{-} \end{cases}$$

Since there is no *string* such that S/H —*string*→ S/G, i.e. the G-set corresponding to the accept state S is the null set, the code, for any k is comma-free.

The sizes of the codes are:

k	1	2	3	4	5	6	7	8
length, $2k+3$	5	7	9	11	13	15	17	19
size	6	18	54	162	486	1458	4374	13122
max size	6	18	56	186	630	2182	7710	27594

5.5

The variable length code represented by the regular expression:

$$0\,1(0 + 11^*)\,00^*1$$

is comma-free and represents an infinite set of binary codewords, with length distribution: k codewords of length $(k + 3)$ for all $k > 1$. The finite state recognizer, with the G-set justification of comma-freeness is:

	0	1	*G*-sets
1	2	–	2 ··· 6
2	–	3	4 6
3	4	5	1
4	6	–	–
5	6	5	–
6	6	1	–
1	2	–	null

The finite state recognizer can be readily converted into a transducer which reads a binary sequence of comma-free codewords and outputs the corresponding ordinal integer x whenever a codeword is received. The parameters x, y are external to the finite state model which calls appropriate procedures to operate on x, y by means of output signals (Table 7).

Table 7.

inputs	1	2	3	4	5	6	States
0	2	2	4	6	6	6	Next state
1	1	3	5	3	5	1	function
0	–	*p*E	*p*0	*p*2	*p*2	*p*2	Output
1	*p*E	*py*	*p*1	*p*E	*p*4	*p*3	function

– No action
*p*0 Set x to 0
py Set y to 0
*p*3 Output results x
*p*E Error, not yet synchronized
*p*1 Set x to 1
*p*2 Replace y by $y + 1$, x by $x + y$
*p*4 Replace y by $y + 1$, x by $x + y + 1$

6. CONCLUSIONS

It has been demonstrated that comma-free codes can be represented concisely and compactly as finite state recognizers, providing, not only a conceptual alternative to the more frequently used sets of symbol strings, but also a sound theoretical base and a set of well developed manipulation techniques. In particular, an efficient algorithm to test for comma-freeness has been presented and its use demonstrated for sets of binary comma-free codes. These include codes with variable codeword lengths, one of which has a greater number of codewords than the theoretical maximum for the corresponding fixed length code, and a code with an infinite number of permissible codewords. The maximal comma-free code with binary codewords of length 10, presented by Yoji, has been represented as a finite state recognizer, the reverse of the code being significantly more compact than that originally given. By using finite state manipulation techniques an alternative code has been developed which has one less codeword (98 codewords compared with the maximal 99) but which can be represented considerably more compactly, requiring 23 states compared with the original 64 states.

REFERENCES

1. L. J. Guibas and A. M. Odlyzko, Maximal prefix-synchronised codes. *SIAM J. Appl. Math.* **35** (2), 401–418 (1978).
2. S. W. Golomb, B. Gordon and L. R. Welch, Comma-free codes. *Canad. J. Math.* **10**, 202–209 (1958).
3. H. C. Crick, J. S. Griffith and L. E. Orgel, Codes without commas. *Proc. Nat. Acad. Sci.* **43**, 416–421 (1957).
4. S. W. Golomb, L. R. Welch and M. Delbruck, Construction and properties of comma-free codes. *Biol. Med. Danske. Vid. Selsk.* **23** (9), 1–34 (1958).
5. B. H. Jiggs, Recent results in comma-free codes. *Canad. J. Math.* **15**, 178–187 (1963).
6. W. L. Eastman, On the construction of comma-free codes. *IEEE Trans. Information Theory* **IT-11**, 263–267 (1965).
7. I. S. Taylor and D. H. Green, Restrictions on nonbinary comma-free codes. *Electron. Lett.* **9** (3), 60–61 (1973).
8. Yoji Niho, On maximal comma-free codes. *IEEE Trans. Information Theory* **IT-19**, 580–581 (1973).
9. R. A. Scholtz, Maximal and variable-length comma-free codes. *IEEE Trans. Information Theory* **IT-15**, 300–306 (1969).
10. J. J. Stiffler, *Theory of Synchronous Communications*, pp. 351–356 *et al.*, Prentice Hall (1971).
11. A. Gill, *Introduction to the Theory of Finite State Machines*, McGraw-Hill (1962).
12. Z. Kohavi, *Switching and Finite Automata Theory*, pp. 280–313, 444–488, McGraw-Hill (1970).
13. R. Y. Kain, *Automata Theory: Machines and Languages*, pp. 28–82, McGraw-Hill (1972).
14. P. J. Denning, J. B. Dennis and J. E. Qualitz, *Machines, Languages and Computation*, pp. 88–225, Prentice Hall (1978).

Received August 1982