

RANDOMIZED GENERATION OF ERROR CONTROL CODES WITH AUTOMATA AND TRANSDUCERS[☆]

STAVROS KONSTANTINIDIS^{1,*}, NELMA MOREIRA² AND ROGÉRIO REIS²

Abstract. We introduce the concept of an f -maximal error-detecting block code, for some parameter f in $(0,1)$, in order to formalize the situation where a block code is close to maximal with respect to being error-detecting. Our motivation for this is that it is computationally hard to decide whether an error-detecting block code is maximal. We present an output-polynomial time randomized algorithm that takes as input two positive integers N, ℓ and a specification of the errors permitted in some application, and generates an error-detecting, or error-correcting, block code of length ℓ that is 99%-maximal, or contains N words with a high likelihood. We model error specifications as (nondeterministic) transducers, which allow one to represent any rational combination of substitution and synchronization errors. We also present some elements of our implementation of various error-detecting properties and their associated methods. Then, we show several tests of the implemented randomized algorithm on various error specifications. A methodological contribution is the presentation of how various desirable error combinations can be expressed formally and processed algorithmically.

Mathematics Subject Classification. 68Q45, 68W20, 94A40, 94B25.

Accepted November 21, 2018.

1. INTRODUCTION

We consider block codes C , that is, sets of words of the same length ℓ , for some integer $\ell > 0$. The elements of C are called *codewords* or C -*words*. We use A to denote the alphabet used for making words and

$$A^\ell = \text{the set of all words of length } \ell.$$

Our typical alphabet will be the binary one $\{0, 1\}$. We shall use the variables u, v, w, x, y, z to denote words over A (not necessarily in C). The *empty word* is denoted by ε . We also consider *error specifications* \mathbf{er} for specifying the error situations permitted in a *channel*—this could be any communication or storage medium. An error specification \mathbf{er} specifies, for each allowed input word x , the set $\mathbf{er}(x)$ of all possible output words resulting by applying specified errors on x . We assume that error-free communication is always possible, so $x \in \mathbf{er}(x)$ for every input word x . On the other hand, if $y \in \mathbf{er}(x)$ and $y \neq x$ then the channel introduces errors into x .

[☆]Research supported by NSERC and FCT project UID/MAT/00144/2013.

Keywords and phrases: Randomized algorithm, output polynomial time algorithm, error control codes, maximal codes, synchronization errors, combinatorial channels.

¹ Department of Mathematics and Computing Science, Saint Mary's University, Halifax, Nova Scotia, Canada.

² CMUP & DCC, Faculdade de Ciências da Universidade do Porto, Rua do Campo Alegre, 4169-007 Porto, Portugal.

* Corresponding author: s.konstantinidis@smu.ca

A typical problem in coding theory is the construction of block codes that are capable of detecting or correcting a certain number of errors. An important requirement is that these codes contain as many words as possible—in some cases this requirement is formalized via the concept of maximal code, or even the concept of a largest cardinality code. Most of classical coding theory deals with various types of substitution errors, that is errors where a symbol of the input word is replaced by another symbol. In this case, many block codes have a vector space structure (they are linear codes) [17]. On the other hand, there is also research on error control codes for synchronization errors, that is errors where a symbol of the input word is deleted and/or a new symbol is inserted in the input word [20]. In this case, there is no known algebraic structure on the set of codewords and code constructions are complex and specific to the particular error combinations, often requiring several subtle assumptions on how errors are permitted or not permitted to occur in input words. Few computational approaches for aiding the construction problem are available, mostly for specific substitution error types [3, 12].

Our approach of error specifications allows one to express various error combinations that includes many of the existing ones as well as any “rational” error combinations (to be made precise further below). Informally, a block code C is **er**-detecting if the channel specified by **er** cannot turn a given C -word into a *different* C -word. It is **er**-correcting if the channel cannot turn two *different* C -words into the same word. We model an error specification as a (nondeterministic) transducer. Unlike automata which “accept” regular languages, transducers “realize” rational relations, called channels in this paper. The channel realized by a transducer **er** is the set of word pairs (x, y) such that $y \in \mathbf{er}(x)$.

In Section 2, we make the above concepts mathematically precise, and show how known examples of various channels can be defined formally with transducers (error specifications) so that they can be processed by algorithms. In Section 3, we present our main randomized algorithm: given an error specification **er**, an **er**-detecting block code $C \subseteq A^\ell$ (which could be empty), and an integer $N > 0$, the algorithm attempts to add N new words of length ℓ to C resulting into a new **er**-detecting code. The new code is 99%-maximal, or it contains N new word with a high likelihood. Our motivation for considering a randomized algorithm is that embedding a given **er**-detecting block code C into a maximal **er**-detecting block code is a computationally hard problem—this is shown in Section 4. In Section 5, we discuss with examples some capabilities of the new module `codes.py` in the open source software package FAdo [1, 5, 10]. In Section 6, we discuss a few more points on channel modelling and present some tests of the randomized algorithm on various error specifications. In Section 7, we conclude with directions for future research.

2. ERROR SPECIFICATIONS AND ERROR CONTROL CODES

We need a mathematical model for error specifications that is useful for answering *algorithmic* questions pertaining to error control codes. We believe the appropriate model for our purposes is that of a transducer. This is because (a) there are many efficient algorithms working on these objects, and (b) to our knowledge most, if not all, examples of combinatorial error control codes in the coding theory literature refer to combinations of substitution and/or synchronization errors whose effects can be expressed as transducers—this, however, is not the case for certain DNA types of errors, which we do not consider here. We note that transducers have been defined as early as in [26], and are a powerful computational tool for processing sets of words—see [2] and pages 41–110 of [25].

A *transducer* is a 5-tuple¹ $\mathbf{t} = (S, A, I, T, F)$ such that A is the alphabet, S is the finite set of states, $I \subseteq S$ is the set of initial states, $F \subseteq S$ is the set of final states, and T is the finite set of transitions. Each transition is a 4-tuple $(s_i, x_i/y_i, t_i)$, where $s_i, t_i \in S$ and x_i, y_i are words over A . The word x_i is the *input label* and the word y_i is the *output label* of the transition. For two words x, y we write $y \in \mathbf{t}(x)$ to mean that y is a possible output of \mathbf{t} when x is used as input. More precisely, there is a sequence

$$(s_0, x_1/y_1, s_1), (s_1, x_2/y_2, s_2), \dots, (s_{n-1}, x_n/y_n, s_n)$$

¹The general definition of transducer allows two alphabets: the input and the output alphabet. Here, however, we assume that both alphabets are the same.

of transitions such that $s_0 \in I$, $s_n \in F$, $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$. The relation $R(\mathbf{t})$ realized by \mathbf{t} is the set of word pairs (x, y) such that $y \in \mathbf{t}(x)$. A relation $\rho \subseteq A^* \times A^*$ is called *rational* if it is realized by a transducer. If every input and every output label of \mathbf{t} is in $A \cup \{\varepsilon\}$, then we say that \mathbf{t} is in *standard form*. We note that every transducer can be converted (in linear time) to one in standard form realizing the same relation. The *domain* of the transducer \mathbf{t} is the set of words x such that $\mathbf{t}(x) \neq \emptyset$. The transducer is called *input-preserving* if $x \in \mathbf{t}(x)$, for all words x in the domain of \mathbf{t} . The inverse of \mathbf{t} , denoted by \mathbf{t}^{-1} , is the transducer that is simply obtained by making a copy of \mathbf{t} and changing each transition $(s, x/y, t)$ to $(s, y/x, t)$. Then

$$x \in \mathbf{t}^{-1}(y) \text{ if and only if } y \in \mathbf{t}(x).$$

A set of words is called a *language*, with a block code being a particular example of a language. We are interested in languages accepted by automata [25]. A (finite) *automaton* \mathbf{a} is a 5-tuple (S, A, I, T, F) as in the case of a transducer, but each transition has only one input label, that is, it is of the form (s, x, t) with x being one alphabet symbol or the empty word ε . For a transition (s, x, t) , we say that it *goes out of state* s . The *language accepted by* \mathbf{a} is denoted by $L(\mathbf{a})$ and consists of all words formed by concatenating the labels in any path from an initial to a final state. The automaton is called *deterministic*, or *DFA* for short, if I consists of a single state, there are no transitions with label ε , and there are no two distinct transitions with same labels going out of the same state. Special cases of automata are constraint systems in which normally all states are final ([23], pp. 1635–1764), and trellises. A *trellis* is an automaton accepting a block code, and has one initial and one final state ([23], pp. 1989–2117). In the case of a trellis \mathbf{a} we refer to $L(\mathbf{a})$ also as the *code represented by* \mathbf{a} . A piece of notation that is useful in the next section is the following, where W is any language,

$$\mathbf{t}(W) = \bigcup_{w \in W} \mathbf{t}(w) \tag{2.1}$$

Thus, $\mathbf{t}(W)$ is the set of all possible outputs of \mathbf{t} when the input is any word from W .

Definition 2.1. An *error specification* \mathbf{er} is an input-preserving transducer. The (*combinatorial*) *channel* specified by \mathbf{er} is $R(\mathbf{er})$, that is, the relation realized by \mathbf{er} .

Figure 1 refers to examples of channels that have been defined informally in past research when designing error control codes. Here these channels are shown as transducers, which can be used as inputs to algorithms for generating error control codes. For example, for $\mathbf{er} = \mathbf{sub}_2$, we have $00101 \in \mathbf{sub}_2(00000)$ because on input 00000, the transducer \mathbf{sub}_2 can read the first two input 0's at state s and output 0, 0; then, still at state s , read the 3rd 0 and output 1 and go to state t_1 ; etc. If we modify \mathbf{id}_2 in Figure 1 by removing state t_2 , then we get the error specification \mathbf{id}_1 representing the channel that allows up to 1 symbol to be deleted or inserted in the input word; then

$$\mathbf{id}_1(\{00, 11\}) = \{00, 0, 000, 100, 010, 001, 11, 1, 011, 101, 110, 111\}.$$

Notation for transducer figures: A short arrow with no label points to an initial state (*e.g.*, state s in Fig. 1), and a double circle indicates a final state (*e.g.*, state t). An arrow with label a/a represents multiple transitions, each with label a/a , for $a \in A$; and similarly for an arrow with label a/ε —recall, $\varepsilon =$ empty word. Two or more labels on one arrow from some state p to some state q represent multiple transitions between p and q having these labels.

2.1. Operations on automata and transducers

Here, we define the operations $\vee, \triangleright, \circ$ on transducers and automata, which are needed in subsequent sections. For computational complexity considerations, the *size* $|\mathbf{m}|$ of a finite state machine (automaton or transducer) \mathbf{m} is the number of states plus the sum of the sizes of the transitions. The size of a transition is 1 plus the

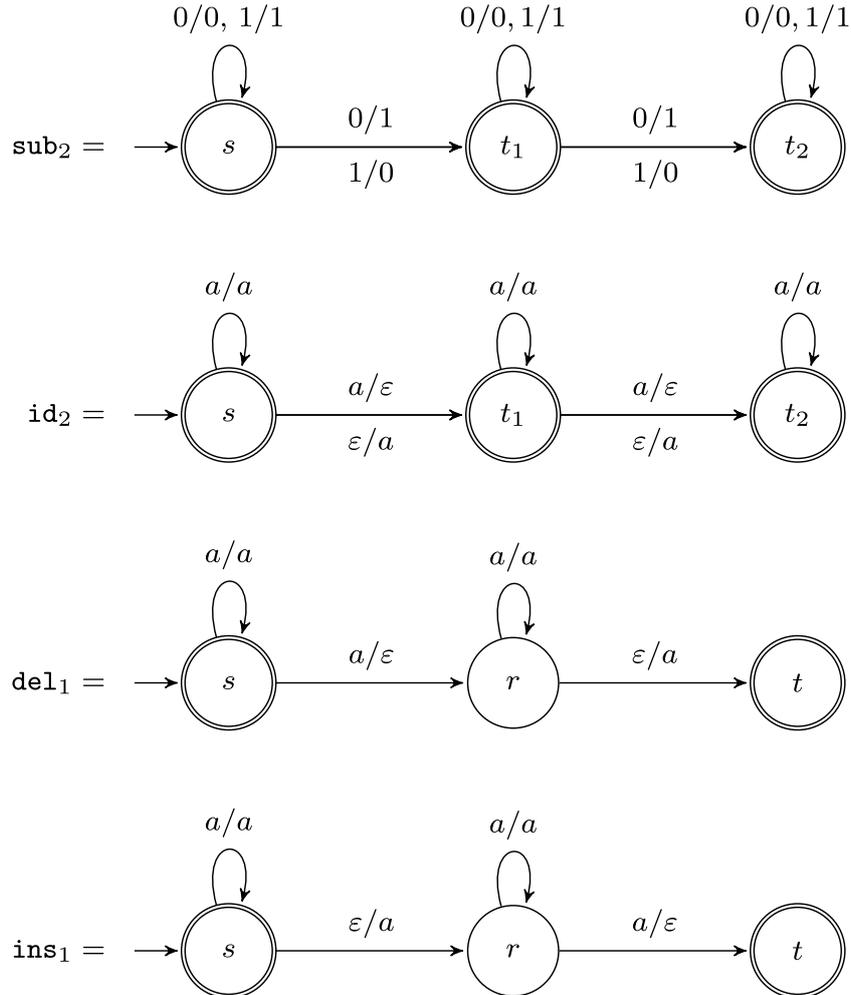


FIGURE 1. Examples of error specifications (input-preserving transducers). sub_2 : uses the binary alphabet $\{0, 1\}$. On input x , sub_2 outputs x , or any word that results by performing one or two substitutions in x . The latter case is when sub_2 takes the transition $(s, 0/1, t_1)$ or $(s, 1/0, t_1)$, corresponding to one error, and then possibly $(t_1, 0/1, t_2)$ or $(t_1, 1/0, t_2)$, corresponding to a second error. id_2 : On input x , id_2 outputs a word that results by inserting and/or deleting at most 2 symbols in x . $\text{del}_1, \text{ins}_1$: considered in [22]. On input x , del_1 (resp., ins_1) outputs either x , or any word that results by deleting (resp., inserting) exactly one symbol in x , and then inserting a symbol at the end of x (resp., deleting the last symbol of x).

length of the label(s) on the transition. We assume that the alphabet A is small so we do not include its size in our estimates.

The operation ‘ \vee ’ between any two transducers \mathbf{t} and \mathbf{s} results into a new transducer which is obtained by simply taking the union of their five corresponding components (states, alphabet, initial states, transitions, final states) after a renaming, if necessary, of the states such that the two channels have no states in common. Then,

$$(\mathbf{t} \vee \mathbf{s})(x) = \mathbf{t}(x) \cup \mathbf{s}(x).$$

An important operation between an automaton \mathbf{a} and a transducer \mathbf{t} , here denoted by ‘ \triangleright ’, returns an automaton ($\mathbf{a} \triangleright \mathbf{t}$) that accepts the set of all possible outputs of \mathbf{t} when the input is any word from $L(\mathbf{a})$, that is,

$$L(\mathbf{a} \triangleright \mathbf{t}) = \mathbf{t}(L(\mathbf{a})).$$

Remark 2.2. We recall here the construction of ($\mathbf{a} \triangleright \mathbf{t}$) from given $\mathbf{a} = (S_1, A, I_1, T_1, F_1)$ and $\mathbf{t} = (S_2, A, I_2, T_2, F_2)$, where we assume that \mathbf{a} contains no transition with label ε . First, if necessary, we convert \mathbf{t} to standard form. Second, if \mathbf{t} contains any transition whose input label is ε , then we add into T_1 transitions (q, ε, q) , for all states $q \in S_1$. Let T_1 denote now the updated set of transitions. Then, we construct the automaton

$$\mathbf{b} = (S_1 \times S_2, A, I_1 \times I_2, T, F_1 \times F_2)$$

such that $((p_1, p_2), y, (q_1, q_2)) \in T$, exactly when there are transitions $(p_1, x, q_1) \in T_1$ and $(p_2, x/y, q_2) \in T_2$. The above construction can be done in time $O(|\mathbf{a}||\mathbf{t}|)$ and the size of \mathbf{b} is $O(|\mathbf{a}||\mathbf{t}|)$. The required automaton ($\mathbf{a} \triangleright \mathbf{t}$) is the trim version of \mathbf{b} , which can be computed in time $O(|\mathbf{b}|)$. (The trim version of an automaton \mathbf{m} is the automaton resulting when we remove any states of \mathbf{m} that do not occur in some path from an initial to a final state of \mathbf{m} .)

The operation ‘ \circ ’ between two transducers \mathbf{t} and \mathbf{s} is called *composition* and returns a new transducer $\mathbf{s} \circ \mathbf{t}$ such that

$$z \in (\mathbf{s} \circ \mathbf{t})(x) \text{ if and only if } y \in \mathbf{t}(x) \text{ and } z \in \mathbf{s}(y), \text{ for some } y.$$

Unlike the construction of $\mathbf{a} \triangleright \mathbf{t}$, in the present paper we make no use of the construction of $\mathbf{s} \circ \mathbf{t}$ other than its meaning displayed above.

2.2. Error-detection, error-correction, maximality

The concepts of error-detection and -correction mentioned in the introduction are phrased more rigorously in the next definition. This definition is adapted from [11], where the concepts are meaningful more generally for any channel that is simply an input-preserving binary relation (not necessarily one realized by a transducer). As explained in the introductory paragraph of this section, however, here we are interested in transducer-based channels.

Definition 2.3. Let C be a block code of length ℓ and let \mathbf{er} be an error specification. We say that C is *\mathbf{er} -detecting* if

$$v \in C, w \in C \text{ and } w \in \mathbf{er}(v) \text{ imply } v = w.$$

We say that C is *\mathbf{er} -correcting* if

$$v \in C, w \in C \text{ and } \mathbf{er}(v) \cap \mathbf{er}(w) \neq \emptyset \text{ imply } v = w.$$

An \mathbf{er} -detecting block code C is called *maximal \mathbf{er} -detecting* if $C \cup \{w\}$ is not \mathbf{er} -detecting for any word w of length ℓ that is not in C . The concept of a *maximal \mathbf{er} -correcting* code is similar.

Referring to Figure 1, we have that a block code C is \mathbf{sub}_2 -detecting iff the minimum Hamming distance of C is > 2 . A block code C is \mathbf{id}_2 -detecting iff the minimum Levenshtein distance of C is > 2 [13]. The *Hamming distance* $H(x, y)$ of two equal-length words x, y is the number of corresponding positions on which they differ (e.g., $H(0000, 0101) = 2$). The *Levenshtein distance* $V(x, y)$ of any two words x, y is the smallest number of

insertions, deletions required to turn x to y (e.g., $V(00, 0110) = 2$). The minimum Hamming (resp. Levenshtein) distance of C is the minimum Hamming (resp. Levenshtein) distance of any two different C -words.

From a logical point of view (see Lem. 2.4) error-detection subsumes the concept of error-correction. This connection is stated already in [11] but without making use of it there. Here we add the fact that maximal error-detection subsumes maximal error-correction. Due to this observation, in this paper we focus only on error-detecting codes.

Lemma 2.4. *Let $C \subseteq A^\ell$ be a block code and \mathbf{er} be an error specification. Then C is \mathbf{er} -correcting if and only if it is $(\mathbf{er}^{-1} \circ \mathbf{er})$ -detecting. Moreover, C is maximal \mathbf{er} -correcting if and only if it is maximal $(\mathbf{er}^{-1} \circ \mathbf{er})$ -detecting.*

Proof. The first statement is already in [11]. For the second statement, first assume that C is maximal \mathbf{er} -correcting and consider any word $w \in A^\ell \setminus C$. If $C \cup \{w\}$ were $(\mathbf{er}^{-1} \circ \mathbf{er})$ -detecting then $C \cup \{w\}$ would also be \mathbf{er} -correcting and, hence, C would be non-maximal; a contradiction. Thus, C must be maximal $(\mathbf{er}^{-1} \circ \mathbf{er})$ -detecting. The converse can be shown analogously. \square

The next lemma is based on concepts considered in [4].

Lemma 2.5. *Let \mathbf{er} be an error specification, let $C \subseteq A^\ell$ be an \mathbf{er} -detecting block code, and let $w \in A^\ell \setminus C$. We have that*

$$C \cup \{w\} \text{ is } \mathbf{er}\text{-detecting if and only if } w \notin (\mathbf{er} \vee \mathbf{er}^{-1})(C). \quad (2.2)$$

Proof. For the ‘only-if part’, assume that $w \notin C$ and $C \cup \{w\}$ is \mathbf{er} -detecting. We show that $w \notin \mathbf{er}^{-1}(C)$: if $w \in \mathbf{er}^{-1}(u)$, for some $u \in C$, then $u \in \mathbf{er}(w)$. This implies that $u = w$, which contradicts $w \notin C$. Similarly one shows that $w \in \mathbf{er}(C)$ leads to a contradiction. For the ‘if’ part, assume that $w \notin (\mathbf{er} \vee \mathbf{er}^{-1})(C)$. Consider any words $u, v \in C \cup \{w\}$ such that $v \in \mathbf{er}(u)$. We need to show that $u = v$. If $u, v \neq w$ then indeed $u = v$, as C is \mathbf{er} -detecting. If $u = w$ or $v = w$ then $v \in \mathbf{er}(u)$ contradicts the assumption $w \notin (\mathbf{er} \vee \mathbf{er}^{-1})(C)$. \square

Definition 2.6. Let $C \subseteq A^\ell$ be an \mathbf{er} -detecting block code. We say that a word w can be added into C if $w \notin (\mathbf{er} \vee \mathbf{er}^{-1})(C)$.

Statement (2.2) above implies that

$$C \text{ is maximal } \mathbf{er}\text{-detecting if and only if } A^\ell \setminus (\mathbf{er} \vee \mathbf{er}^{-1})(C) = \emptyset. \quad (2.3)$$

Definition 2.7. The *maximality index* of a block code $C \subseteq A^\ell$ w.r.t. an error specification \mathbf{er} is the quantity

$$\text{maxind}(C, \mathbf{er}) = \frac{|A^\ell \cap (\mathbf{er} \vee \mathbf{er}^{-1})(C)|}{|A^\ell|}.$$

Let f be a real number in $[0, 1]$. An \mathbf{er} -detecting block code C is called *f -maximal \mathbf{er} -detecting* if $\text{maxind}(C, \mathbf{er}) \geq f$.

The maximality index of C is the proportion of the ‘used up’ words of length ℓ over all words of length ℓ . We have the following useful lemma.

Lemma 2.8. *Let \mathbf{er} be an error specification and let $C \subseteq A^\ell$ be an \mathbf{er} -detecting block code.*

1. $\text{maxind}(C, \mathbf{er}) = 1$ if and only if C is maximal \mathbf{er} -detecting.
2. Assuming that words are chosen uniformly at random from A^ℓ , the maximality index is the probability that a randomly chosen word w of length ℓ cannot be added into C preserving its being \mathbf{er} -detecting, that is,

$$\text{maxind}(C, \mathbf{er}) = \Pr \left[w \text{ cannot be added into } C \right].$$

Proof. The first statement follows from Definition 2.7 and condition (2.3). The second statement follows when we note that the event that a randomly chosen word w from A^ℓ cannot be added into C is the same as the event that $w \in A^\ell \cap (\mathbf{er} \vee \mathbf{er}^{-1})(C)$. \square

3. GENERATING ERROR CONTROL CODES

We now turn our attention to randomized algorithms processing error specifications and sets of words. Our main goal is to compute, for any given error specification \mathbf{er} , integer $N > 0$, and deterministic trellis \mathbf{a} representing some \mathbf{er} -detecting code C , a deterministic trellis accepting an \mathbf{er} -detecting superset of C that is close to maximal or contains N new words with a high likelihood. Solving this problem can be used to also solve the problem of generating an \mathbf{er} -detecting block code of up to N words of length ℓ , for given \mathbf{er} and integers $N, \ell > 0$, such that the generated code is close to maximal or contains N new words with a high likelihood. We note that our focus on trellises is because most error control codes in the literature are block codes, which are naturally accepted by trellises. Next we present our randomized algorithms—we use [21] as reference for basic concepts. We assume that we have available to use in our algorithms an ideal method `pickFrom(A, ℓ)` that chooses uniformly at random a word in A^ℓ . A randomized algorithm $R(\dots)$ with specific values for its parameters can be viewed as a random variable whose value is whatever value is returned by executing R on the specific values.

Lemma 3.1. *Consider the algorithm `nonMax` in Figure 2, which takes as input an error specification \mathbf{er} , a trellis \mathbf{a} accepting an \mathbf{er} -detecting code of some length ℓ , and a positive integer $n > 0$.*

1. *The algorithm either returns a word $w \in A^\ell \setminus L(\mathbf{a})$ such that the code $L(\mathbf{a}) \cup \{w\}$ is \mathbf{er} -detecting, or it returns `None`.*
2. *Let $f \in (0, 1)$. If $L(\mathbf{a})$ is not f -maximal \mathbf{er} -detecting, then*

$$\Pr \left[\text{nonMax returns None} \right] < f^n.$$

3. *The time complexity of `nonMax` is $O(n\ell|\mathbf{a}||\mathbf{er}|)$.*

Proof. The first statement follows from statement (2.2) in the previous section, as any w returned by the algorithm is not in $(\mathbf{er} \vee \mathbf{er}^{-1})(L(\mathbf{a}))$. For the second statement, suppose that the code $L(\mathbf{a})$ is not f -maximal \mathbf{er} -detecting. Then, using Lemma 2.8, we have that $p < f$, where $p = \Pr [w \in L(\mathbf{b})]$. As the algorithm returns `None` when all n picked words w are in $L(\mathbf{b})$, we have

$$\Pr \left[\text{nonMax returns None} \right] = p^n < f^n,$$

```

nonMax ( $\mathbf{er}, \mathbf{a}, n$ )
   $\mathbf{b} := (\mathbf{a} \triangleright (\mathbf{er} \vee \mathbf{er}^{-1}))$ ;
   $\ell :=$  the length of the words in  $L(\mathbf{a})$ ;
   $\text{tr} := 1$ ;
  while ( $\text{tr} \leq n$ ):
     $w := \text{pickFrom}(A, \ell)$ ;
    if ( $w$  not in  $L(\mathbf{b})$ ) return  $w$ ;
     $\text{tr} := \text{tr}+1$ ;
  return None;

```

FIGURE 2. Algorithm `nonMax`—see Lemma 3.1.

```

makeCode (er, a, N)
  f := 0.99           // f and  $\varepsilon$  are named constants
   $\varepsilon$  := 0.0001
  n := 1 +  $\lfloor \log \frac{N}{\varepsilon} / \log \frac{1}{f} \rfloor$ 
  W := empty list;
  c := a
  i := 1;
  while (i ≤ N)
    w := nonMax (er, c, n);
    if (w is None) return c, W;
    else { add w to c and to W;
          i := i+1;}
  return c, W;

```

FIGURE 3. Algorithm `makeCode`—see Theorem 3.4. The trellis \mathbf{a} can be omitted so that the algorithm would start with an empty set of codewords. In this case, however, the algorithm would require as extra input the codeword length ℓ and the desired alphabet A . We used the fixed values $f = 0.99$ and $\varepsilon = 10^{-4}$, as they seem to work well in practical testing.

as required. For the third statement, we use standard results from automaton theory [25] and Remark 2.2. In particular, computing \mathbf{b} can be done in time $O(|\mathbf{a}| \cdot |\mathbf{er}|)$ such that $|\mathbf{b}| = O(|\mathbf{a}| \cdot |\mathbf{er}|)$. Testing whether $w \in L(\mathbf{b})$ can be done in time $O(|w||\mathbf{b}|) = O(\ell|\mathbf{b}|)$. Thus, the algorithm works in time $O(n\ell|\mathbf{a}||\mathbf{er}|)$. \square

Remark 3.2. We mention the important observation that one can modify the algorithm `nonMax` by removing the construction of \mathbf{b} and replacing the ‘if’ line in the loop with

if ($w \notin L(\mathbf{a})$ and $L(\mathbf{a}) \cup \{w\}$ is \mathbf{er} -detecting) return w ;

While with this change the output would still be correct, the time complexity of the algorithm would increase to $O(n|\mathbf{a}|^2|\mathbf{er}|)$. This is because testing whether $L(\mathbf{v})$ is \mathbf{er} -detecting, for any given automaton \mathbf{v} and error specification \mathbf{er} , can be done in time $O(|\mathbf{v}|^2|\mathbf{er}|)$, and in practice $|\mathbf{v}|$ is much larger than ℓ .

In Figure 3, we present the main algorithm for adding new words into a given deterministic trellis \mathbf{a} .

Remark 3.3. In some sense, algorithm `makeCode` generalizes to arbitrary error specifications the idea used in the proof of the well-known Gilbert-Varshamov bound [19] for the largest possible block code $M \subseteq A^\ell$ that is \mathbf{sub}_k -correcting, for some number k of substitution errors. In that proof, a word can be added into the code M if the word is outside of the union of the ‘balls’ $\mathbf{sub}_{2k}(u)$, for all $u \in M$. In that case, we have that $\mathbf{sub}_k^{-1} = \mathbf{sub}_k$ and $(\mathbf{sub}_k^{-1} \circ \mathbf{sub}_k) = \mathbf{sub}_{2k}(u)$. The present algorithm adds new words w to the constructed trellis \mathbf{c} such that each new word w is outside of the ‘union-ball’ $(\mathbf{er} \vee \mathbf{er}^{-1})(L(\mathbf{c}))$.

Theorem 3.4. *Algorithm `makeCode` in Figure 3 takes as input an error specification \mathbf{er} , a deterministic trellis \mathbf{a} of some length ℓ , and an integer $N > 0$ such that the code $L(\mathbf{a})$ is \mathbf{er} -detecting, and returns a deterministic trellis \mathbf{c} and a set W of words of length ℓ not in $L(\mathbf{a})$ such that the following statements hold true:*

1. $L(\mathbf{c}) = L(\mathbf{a}) \cup W$ and $L(\mathbf{c})$ is \mathbf{er} -detecting.
2. $L(\mathbf{c})$ is f -maximal, or $|W| = N$ with probability $> 1 - \varepsilon$; where f and ε are the named constants in the algorithm.
3. The algorithm runs in time $O(\ell N \log N |\mathbf{er}| |\mathbf{a}| + \ell^2 N^2 \log N |\mathbf{er}|)$.

Proof. Let \mathbf{c}_i be the value of the trellis \mathbf{c} at the end of the i th iteration of the while loop. The first statement follows from Lemma 3.1: any word w returned by `nonMax` is such that $L(\mathbf{c}_i) \cup \{w\}$ is \mathbf{er} -detecting. For the

second statement, assume that, at the end of execution, $L(\mathbf{c})$ is not f -maximal. Then as $L(\mathbf{c}_i) \subseteq L(\mathbf{c}_{i+1})$, we have that each $L(\mathbf{c}_i)$ is not f -maximal. Thus, at step i , the probability p_i that `nonMax` returns `None` is $< f^n$. Then we have

$$\begin{aligned} \Pr \left[|W| = N \right] &= \Pr \left[\text{nonMax returns } \neq \text{None in all } N \text{ iterations} \right] \\ &> (1 - f^n)^N \\ &\geq 1 - N f^n \\ &> 1 - \varepsilon, \end{aligned}$$

where the second last inequality follows from Bernoulli's inequality, and the last inequality follows from the definition of n in the algorithm.

For the third statement, as the loop in the algorithm `nonMax` performs n iterations, we have that the cost of each `nonMax` call is $O(n\ell|\mathbf{c}_i||\mathbf{er}|)$. The cost of adding a new word w of length ℓ to \mathbf{c}_{i-1} is² $O(\ell)$ and increases its size by $O(\ell)$, so each \mathbf{c}_i is of size $O(|\mathbf{a}| + i\ell)$. Thus, the cost of the i th iteration of the while loop in `makeCode` is $O(n\ell|\mathbf{er}|(|\mathbf{a}| + i\ell))$. As there are up to N iterations the total cost is

$$\sum_{i=1}^N O\left(n\ell|\mathbf{er}| \cdot (|\mathbf{a}| + i\ell)\right) = O\left(n\ell N|\mathbf{er}||\mathbf{a}| + n\ell^2 N^2|\mathbf{er}|\right).$$

Finally by the definition of n in the algorithm, we have that $n = O(\log N)$. □

Remark 3.5. The second statement of the above theorem implies that the algorithm fails with probability $< \varepsilon$. Indeed, we consider that the algorithm succeeds when the code $L(\mathbf{a}) \cup W$ is f -maximal or W contains N new words. Thus, it fails exactly when $L(\mathbf{a}) \cup W$ is not f -maximal and W contains fewer than N words. We say that the *failure tolerance* of the algorithm is $< \varepsilon$.

Computational complexity. Algorithm `makeCode` is not a decision algorithm; it produces up to N words, where N is one of the input parameters and, therefore its time complexity cannot be lower than $O(N)$. Following [9], we say that an algorithm is *output-polynomial time* if it runs within polynomial time with respect to the sum of the sizes of the input and the output.³ Thus, `makeCode` is an output-polynomial time algorithm. When only the input size is considered, then the algorithm can be called pseudo-polynomial as the complexity involves the term $N^2 \log N$ which is of exponential magnitude if N is given in binary, or polynomial magnitude if N is given in unary. On the other hand, using the algorithm to add only one word into $L(\mathbf{a})$ (case of $N = 1$), requires time $O(\ell|\mathbf{er}||\mathbf{a}| + \ell^2|\mathbf{er}|)$, which is of polynomial magnitude with respect to the size of the input.

Remark 3.6. In the version of the algorithm `makeCode` where the initial trellis \mathbf{a} is omitted, the time complexity is $O(\ell^2 N^2 \log N |\mathbf{er}|)$. We also note that the algorithm would work with the same time complexity if the given trellis \mathbf{a} is not deterministic. In this case, however, the resulting trellis would not be (in general) deterministic either.

4. WHY NOT USE A DETERMINISTIC ALGORITHM

In the first place we wanted to solve efficiently the following problem, which we call MAKECODE:

²This can be done as follows, using the fact that \mathbf{c}_{i-1} is deterministic: (i) access the start state s_0 of \mathbf{c}_{i-1} ; (ii) for each symbol $\sigma_1, \sigma_2, \dots$ of the word w follow transitions of \mathbf{c}_{i-1} :

$$(s_0, \sigma_1, s_1), \dots, (s_{j-1}, \sigma_j, s_j)$$

until there is no transition from s_j with label σ_{j+1} ; (iii) add in \mathbf{c}_{i-1} new states $t_{j+1}, \dots, t_{\ell-1}$ and transitions $(s_j, \sigma_{j+1}, t_{j+1}), \dots, (t_{\ell-1}, \sigma_\ell, f)$, where f is the existing final state of \mathbf{c}_{i-1} .

³The concept of output-polynomial time for enumeration algorithms is already signified in [7], where the terms 'polynomial total time' and 'polynomial delay' are used for such algorithms.

Instance error specification \mathbf{er} ; deterministic trellis \mathbf{d} of some word length ℓ such that $L(\mathbf{d})$ is \mathbf{er} -detecting; positive integer N .

Answer set W of words of length ℓ not in $L(\mathbf{d})$ such that $L(\mathbf{d}) \cup W$ is \mathbf{er} -detecting, and $L(\mathbf{d}) \cup W$ is maximal \mathbf{er} -detecting or $|W| = N$.

Solving the above problem also solves the problem of generating a set W of up to N words of length ℓ which is \mathbf{er} -detecting and such that W is maximal \mathbf{er} -detecting or $|W| = N$, where the input consists of \mathbf{er} , ℓ and N .⁴ It turns out, however, that one can reduce to MAKECODE the following coNP-hard problem (see Thm. 4.2), which we call MAXED:

Instance error specification \mathbf{er} ; deterministic trellis \mathbf{d} such that $L(\mathbf{d})$ is \mathbf{er} -detecting.

Answer whether $L(\mathbf{d})$ is maximal \mathbf{er} -detecting.

Thus as MAKECODE is hard, we have made in this work a practically similar problem that can be solved in output-polynomial time using a randomized algorithm with a less than ε failure tolerance (see Rem. 3.5).

In the next remark, we explain why MAXED is reducible to MAKECODE, and then we proceed to showing that MAXED is coNP-hard.

Remark 4.1. To show that MAXED is polynomially reducible to MAKECODE, we consider the problem EMBED:

Instance error specification \mathbf{er} ; deterministic trellis \mathbf{d} of some word length ℓ such that $L(\mathbf{d})$ is \mathbf{er} -detecting.

Answer set W of words of length ℓ not in $L(\mathbf{d})$ such that $L(\mathbf{d}) \cup W$ is maximal \mathbf{er} -detecting

First we note that, in the above problem, W cannot have more than 2^ℓ words. Then it follows that EMBED is polynomially reducible to MAKECODE by mapping any instance $(\mathbf{er}, \mathbf{d})$ of EMBED to the instance $(\mathbf{er}, \mathbf{d}, 1 + 2^\ell)$ of MAKECODE, where $1 + 2^\ell$ is computed in binary. Next, we have that MAXED is polynomially reducible to EMBED. Indeed, this follows when we note that $L(\mathbf{d})$ is maximal \mathbf{er} -detecting if and only if any solution W to EMBED is such that $W = \emptyset$.

Theorem 4.2. MAXED is coNP-hard.

Proof. Let FULLBLOCK be the problem of deciding whether a given (nondeterministic) trellis over the alphabet $A_2 = \{0, 1\}$ with no ε -labeled transitions accepts A_2^ℓ , for some ℓ . The statement is a logical consequence of the following claims.

Claim 1: FULLBLOCK is coNP-complete.

Claim 2: FULLBLOCK is polynomially reducible to MAXED.

The first claim follows from the proof of the following fact on page 329 of [16]: Deciding whether two given star-free regular expressions over A_2 are inequivalent is an NP-complete problem. Indeed, in that proof the first regular expression can be arbitrary, but the second regular expression represents the language A^ℓ , for some positive integer ℓ . Moreover, converting a star-free regular expression to an acyclic automaton with no ε -labeled transitions is a polynomial time problem.

For the second claim, consider any trellis $\mathbf{a} = (S, A_2, s, T, F)$ with no ε -labeled transitions in T . We need to construct in polynomial time an instance $(\mathbf{d}, \mathbf{er})$ of MAXED such that \mathbf{a} accepts A_2^ℓ if and only if $L(\mathbf{d})$ is a maximal \mathbf{er} -detecting block code of length ℓ . The rest of the proof consists of 5 parts: construction of deterministic trellis \mathbf{d} accepting words of length ℓ , construction of \mathbf{er} , facts about \mathbf{d} and \mathbf{er} , proving that $L(\mathbf{d})$ is \mathbf{er} -detecting, proving that \mathbf{a} accepts A_2^ℓ if and only if $L(\mathbf{d})$ is maximal \mathbf{er} -detecting.

Construction of \mathbf{d} : Let A be the alphabet $A_2 \cup T$, where T is the set of transitions of \mathbf{a} . Thus, $A \setminus A_2 = T$. The required deterministic trellis \mathbf{d} is any deterministic trellis accepting $A^\ell \setminus A_2^\ell$, that is,

$$L(\mathbf{d}) = A^\ell \setminus A_2^\ell.$$

⁴Enumerating the words of optimal codes using tools specific to some error situation is a relevant problem [28].

This can be constructed, for instance, by making deterministic trellises \mathbf{d}_1 and \mathbf{d}_2 accepting, respectively, A^ℓ and A_2^ℓ , and then intersecting \mathbf{d}_1 with the complement of \mathbf{d}_2 . Note that any word in $L(\mathbf{d})$ contains at least one symbol in T .

Construction of \mathbf{er} : This is of the form $(\mathbf{er}_1 \vee \mathbf{er}_2)$ as follows. The transducer \mathbf{er}_2 has only one state s and transitions $(s, \alpha/\alpha, s)$, for all $\alpha \in A$, and realizes the identity relation $\{(x, x) \mid x \in A^*\}$. Thus, we have that $\mathbf{er}_2(x) = \{x\}$, for all words $x \in A^*$. The transducer $\mathbf{er}_1 = (S, A, s, T'', F)$ is such that T'' consists of exactly the transitions $(p, (p, a, q)/a, q)$ for which (p, a, q) is a transition of \mathbf{a} . Note that \mathbf{er}_1 rejects every string that contains symbols not from T .

Facts about \mathbf{d} and \mathbf{er} : The following facts are helpful in the rest of the proof. Some of these facts refer to the deterministic trellis $\mathbf{d}_1 = (S, T, s, T_1, F)$ resulting by omitting the output parts of the transition labels of \mathbf{er}_1 , that is, $(p, (p, a, q), q) \in T_1$ exactly when $(p, (p, a, q)/a, q) \in T''$. Then, $L(\mathbf{d}_1) \subseteq T^\ell \subseteq L(\mathbf{d})$.

F0: $L(\mathbf{d}_1 \triangleright \mathbf{er}_1) = L(\mathbf{a})$.

F1: The domain of \mathbf{er}_1 is $L(\mathbf{d}_1)$, a subset of T^ℓ .

F2: If $v \in \mathbf{er}_1(u)$ then $v \in A_2^\ell$ and $v \neq u$.

F3: $\mathbf{er}_1(L(\mathbf{d})) = L(\mathbf{a})$.

F4: $\mathbf{er}_1^{-1}(L(\mathbf{d})) = \emptyset$.

For fact F0, note that the product construction described in Remark 2.2 produces in $(\mathbf{d}_1 \triangleright \mathbf{er}_1)$ exactly the transitions $((p, p), a, (q, q))$, where (p, a, q) is a transition in \mathbf{a} , by matching any transition $(p, (p, a, q), q)$ of \mathbf{d}_1 only with the transition $(p, (p, a, q)/a, q)$ of \mathbf{er}_1 . Fact F1 follows by the construction of \mathbf{er}_1 and the definition of \mathbf{d}_1 : in any accepting computation of \mathbf{er}_1 , the input labels appear in an accepting computation of \mathbf{d}_1 that uses the same sequence of states. F3 is shown as follows: As the domain of \mathbf{er}_1 is $L(\mathbf{d}_1)$ and $L(\mathbf{d}_1) \subseteq L(\mathbf{d})$, we have that $\mathbf{er}_1(L(\mathbf{d})) = \mathbf{er}_1(L(\mathbf{d}_1))$, which is $L(\mathbf{a})$ by F0. Fact F4 follows by noting that the domain of \mathbf{er}_1^{-1} is a subset of A_2^ℓ but $L(\mathbf{d})$ contains no words in A_2^ℓ .

$L(\mathbf{d})$ is \mathbf{er} -detecting: Let $u, v \in L(\mathbf{d})$ such that $v \in \mathbf{er}(u) = \mathbf{er}_1(u) \cup \{u\}$. We need to show that $v = u$, that is, to show that $v \notin \mathbf{er}_1(u)$. Indeed, if $v \in \mathbf{er}_1(u)$ then $v \in A_2^\ell$, which contradicts $v \in L(\mathbf{d}) = A^\ell \setminus A_2^\ell$.

\mathbf{a} accepts A_2^ℓ if and only if $L(\mathbf{d})$ is maximal \mathbf{er} -detecting: By statement (2.3) we have that $L(\mathbf{d})$ is maximal \mathbf{er} -detecting, if and only if $(\mathbf{er} \vee \mathbf{er}^{-1})(L(\mathbf{d})) = A^\ell$. We have:

$$\begin{aligned} (\mathbf{er} \vee \mathbf{er}^{-1})(L(\mathbf{d})) &= L(\mathbf{d}) \cup \mathbf{er}_1(L(\mathbf{d})) \cup \mathbf{er}_1^{-1}(L(\mathbf{d})) \\ &= (A^\ell \setminus A_2^\ell) \cup L(\mathbf{a}) \cup \emptyset = (A^\ell \setminus A_2^\ell) \cup L(\mathbf{a}). \end{aligned}$$

Thus, $L(\mathbf{d})$ is maximal \mathbf{er} -detecting, if and only if $L(\mathbf{a}) = A_2^\ell$, as required. \square

5. IMPLEMENTATION AND USE

All main algorithmic tools have been implemented over the years in the Python package FAdo [1, 5, 10]. Many aspects of the new module `FAdo.codes` are presented in [10]. Here we present methods of that module pertaining to generating codes.

Assume that the string `d1` contains a description of the transducer `del1` in FAdo format. In particular, `d1` begins with the type of FAdo object being described, the final states, and the initial states (after the character `*`). Then, `d1` contains the list of transitions, with each one of the form “`s x y t\n`”, where ‘`\n`’ is the new-line character. This is shown in the following Python script.

```
import FAdo.codes as codes
d1 = '@Transducer 0 2 * 0\n'
      '0 0 0 0\n0 1 1 0\n0 0 @epsilon 1\n0 1 @epsilon 1\n'
      '1 0 0 1\n1 1 1 1\n1 @epsilon 0 2\n1 @epsilon 1 2\n'
pd1 = codes.buildErrorDetectPropS(d1)
a = pd1.makeCode(100, 8, 2)
```

```

print pd1.notSatisfiesW(a)
print pd1.nonMaximalW(a, m)
s2 = ...string for transducer sub_2
ps2 = codes.buildErrorDetectPropS(s2)
pd1s2 = pd1 & ps2
b = pd1s2.makeCode(100, 8, 2)

```

The above script uses the string `d1` to create the object `pd1` representing⁵ the del_1 -detection property over the alphabet $\{0,1\}$. Then, it constructs an automaton `a` representing a del_1 -detecting block code of length 8 with up to 100 words over the 2-symbol alphabet $\{0,1\}$. The method `notSatisfiesW(a)` tests whether the code $L(\mathbf{a})$ is del_1 -detecting and returns a witness of non-error-detection (= pair of codewords u, v with $v \in \text{del}_1(u)$), or `(None, None)`—of course, in the above example it would return `(None, None)`. The method `nonMaximalW(a, m)` tests whether the code $L(\mathbf{a})$ is maximal del_1 -detecting and returns either a word $v \in L(\mathbf{m}) \setminus L(\mathbf{a})$ such that $L(\mathbf{a}) \cup \{v\}$ is del_1 -detecting, or `None` if $L(\mathbf{a})$ is already maximal. The object `m` is any automaton—here it is the trellis representing A^ℓ . This method is used only for small codes, as in general the maximality problem is algorithmically hard (recall Thm. 4.2), which motivated us to consider the randomized algorithm `nonMax` in this paper. For any error specification `er` and trellis `a`, the method `notSatisfiesW(a)` can be made to work in time $O(|\mathbf{er}| |a|^2)$, which is of polynomial complexity. The operation ‘&’ combines error-detection properties. Thus, the second call to `makeCode` constructs a code that is del_1 -detecting and `sub_2`-detecting (= `sub_1`-correcting).

6. MORE ON CHANNEL MODELLING, TESTING

In this section, we consider further examples of error specifications and show how operations on error specifications can result in new ones. We also show the results of testing our codes generation algorithm for several different error specifications.

Remark 6.1. We note that the definition of error-detecting (or error-correcting) block code C is trivially extended to any language L , that is, one replaces in Definition 2.3 ‘block code C ’ with ‘language L ’. Let `er`, `er1`, `er2` be error specifications. By Definition 2.3 and using standard logical arguments, it follows that

1. L is `er1`-detecting and `er2`-detecting, if and only if L is `(er1 ∨ er2)`-detecting;
2. L is `er-1`-detecting, if and only if it is `er`-detecting, if and only if it is `(er-1 ∨ er)`-detecting.

The inverse of del_1 is ins_1 and is shown in Figure 1, where recall it results by simply exchanging the order of the two words in all the labels in del_1 . By statement 2 of the above remark, the del_1 -detecting codes are the same as the ins_1 -detecting ones, and the same as the $(\text{del}_1 \vee \text{ins}_1)$ -detecting ones—this is shown in [22] as well. The method of using transducers to model channels is quite general and one can give many more examples of past channels as transducers, as well as channels not studied before. Some further examples are shown in Figures 4–6.

One can go beyond the classical error control properties and define certain synchronization properties via transducers. Let `OF` be the set of all overlap-free words, that is, all words w such that a proper and nonempty prefix of w cannot be a suffix of w . A block code $C \subseteq \text{OF}$ is a *solid code* if any proper and nonempty prefix of a C -word cannot be a suffix of a C -word. For example, $\{0100, 1001\}$ is not a solid code, as `01` is a prefix and a suffix of some codewords and `01` is nonempty and a proper prefix (shorter than the codewords). Solid codes can also be non-block codes by extending appropriately the above definition [27] (they are also called codes without overlaps in [14]). The transducer `ov` in Figure 6 is such that any block code $C \subseteq \text{OF}$ is a solid code, if and only if C is an ‘`ov`-detecting’ block code. We note that solid codes have instantaneous synchronization capability (in particular all solid codes are comma-free codes) as well as synchronization in the presence of noise [8].

⁵In [10], the authors implement the Python class `ErrDetectProp`. An object of this class is initialized using an error specification `er`, like `del1`, and ‘represents’ all languages that are `er`-detecting. Each such object has methods to test whether the language of a given automaton satisfies the `er`-detection property or whether that language is maximal `er`-detecting.

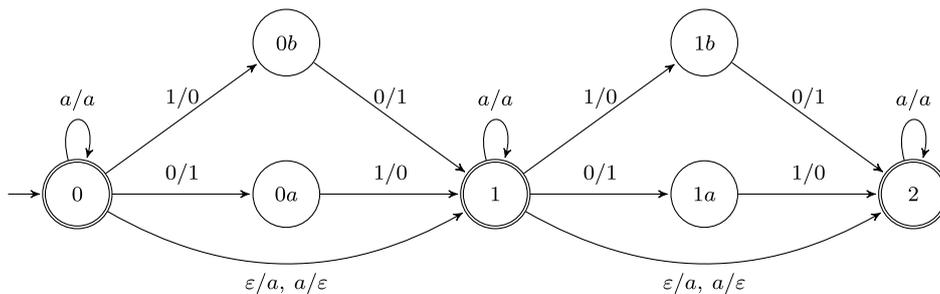


FIGURE 4. The channel specified by bsid_2 allows up to two errors in the input word. Each of these errors can be a deletion, an insertion, or a bit shift: a 10 becomes 01, or a 01 becomes 10. The alphabet is $\{0, 1\}$.

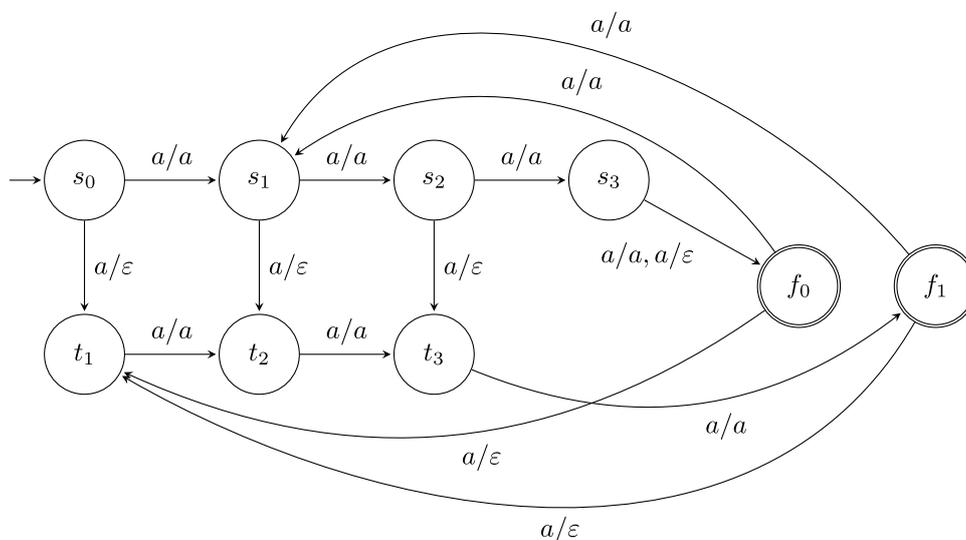


FIGURE 5. Transducer segd_4 for the segmented deletion channel of [18] with parameter $b = 4$. In each of the length b consecutive segments of the input word, at most one deletion error occurs. The length of the input word is a multiple of b . By Lemma 2.4, segd_4 -correction is equivalent to $(\text{segd}_4^{-1} \circ \text{segd}_4)$ -detection.

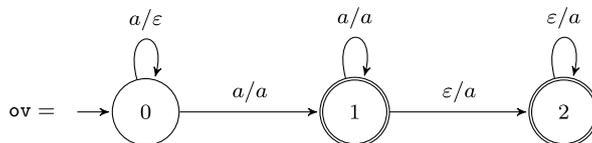


FIGURE 6. This input-preserving transducer deletes a prefix of the input word (a possibly empty prefix) and then inserts a possibly empty suffix at the end of the input word.

We performed several executions of the algorithm `makeCode` on various error specifications using $f = 0.99$, $N \leq 2^{15}$, no initial trellis, and alphabet $A = \{0, 1\}$. In all executions, we used in `nonMax` the fixed value $n = 2000$, which is higher than the one defined in `nonMax` for $N \leq 2^{15}$ and $\varepsilon = 10^{-4}$ (for example $n = 1951$ when $N = 2^{15}$). This choice implies a better failure tolerance for `makeCode` (smaller than 10^{-4}) and an opportunity to see how the size of the error specification affects execution time. It turned out that executions took a long time for $N > 2^{12}$,

TABLE 1. Code Generation I.

N	ℓ	id_2				id_5			
		<i>time</i>	<i>avg</i>	<i>max</i>	<i>min</i>	<i>time</i>	<i>avg</i>	<i>max</i>	<i>min</i>
100	8	0.42	20.0	23	17	1.17	4.9	6	4
500	12	9.47	181.8	192	171	11.80	22.8	26	20
2^{11}	16	2063.99	1858.0	1882	1849	237.07	130.4	140	123
2^{12}	18	78678.58	4096	4096	4096	2086.91	315.2	331	304

TABLE 2. Code Generation II.

N	ℓ	sub_2			
		<i>time</i>	<i>avg</i>	<i>max</i>	<i>min</i>
100	8	0.14	16.4	18	15
500	12	4.57	153.4	159	145
2^{11}	16	1367.32	1589.4	1621	1597
2^{12}	18	79074.42	4096	4096	4096

so those execution times are not reported. All experiments were performed with PyPy, which implements the Python language version 2.7.12 [24], on a computer with an Intel Xeon Quad-Core X5550 at 2.67 GHz processor running a Debian GNU/Linux system.

Tables 1–5 summarize some of the results. In each table, the first two columns give the values of N and ℓ . In Table 4, the third column under the heading *end* indicates the pattern with which all desired codewords end (either 01 or empty)—see discussion on this table further below. For $N = 100$ and $N = 500$, and for each error specification, we executed `makeCode` 50 times and reported the average, largest and smallest sizes of the 50 generated codes. For $N = 2^{11}$ and $N = 2^{12}$, we reported the same figures by executing the algorithm 5 times. The average time of the generation of each code (in seconds) is also presented. Some of the larger codes for $N = 2^{12}$ took about one day to be generated.

In Table 1, the entries for [$N = 100, \ell = 8; \mathbf{er} = \text{id}_2$] correspond to 2-synchronization error-detection which is equivalent to 1-synchronization error-correction. Our largest randomly generated code has 23 codewords. The Levenshtein code [13] of length 8 has 30 codewords. In the same table, the entries for [$N = 500, \ell = 12; \mathbf{er} = \text{id}_5$] correspond to 5-synchronization error-detection which implies 2-synchronization error-correction. Our largest randomly generated code has 26 codewords. As it is id_5 -detecting it is also id_4 -detecting. Using a computer search specific to id_4 , the authors of [28] found a id_4 -detecting code having 29 codewords.

In Table 2, the entries for [$N = 100, \ell = 8; \mathbf{er} = \text{sub}_2$] correspond to 2-substitution error-detection, which is equivalent to 1-substitution error-correction. Our largest randomly generated code has 12 codewords. The Hamming code of length 8 has 16 codewords—in fact the original Hamming code is of length 7, but one can append a 0 to every codeword so that the codeword length is 8.

In Table 3, the entries for [$N = 2^{11}, \ell = 16; \mathbf{er} = \text{sub}_7$] correspond to 7-substitution error-detection. Our largest randomly generated code has 9 codewords. Also, the entries for [$N = 2^{12}, \ell = 18; \mathbf{er} = \text{sub}_5$] correspond to 5-substitution error-detection. Our largest randomly generated code has 130 codewords. In [6], the author uses theory of linear block codes (that is, codes whose elements constitute a vector space) and constructs a 7-substitution error-detecting code of length 16 having 32 words, and a 5-substitution error-detecting code of length 18 having 512 words.

In Table 4, the entries for [$N = 100, \ell = 8, \text{end} = 01; \mathbf{er} = \text{del}_1$] correspond to the systematic code of [22], that is, the code consisting of all codewords ending with 01—any of the 64 possible 6-bit words can be used in the positions 1–6 of these codewords. In the same table, the entries for [$N = 500, \ell = 12, \text{end} = \varepsilon; \mathbf{er} = \text{ov}$] correspond to block solid codes of length 12. Our largest randomly generated code has 77 codewords. When we required that all words end with 01, then the largest generated code has 55 words. In [15], the author considers

TABLE 3. Code Generation III.

N	ℓ	sub ₅				sub ₇			
		<i>time</i>	<i>avg</i>	<i>max</i>	<i>min</i>	<i>time</i>	<i>avg</i>	<i>max</i>	<i>min</i>
100	8	0.05	2.0	2	2	0.06	2.0	2	2
500	12	0.34	9.6	12	7	0.12	3.0	4	2
2 ¹¹	16	4.73	43.2	46	40	0.76	9.0	9	9
2 ¹²	18	36.17	110.6	130	94	6.52	19.2	21	17

TABLE 4. Code Generation IV.

N	ℓ	<i>end</i>	ov				del ₁			
			<i>time</i>	<i>avg</i>	<i>max</i>	<i>min</i>	<i>time</i>	<i>avg</i>	<i>max</i>	<i>min</i>
100	8		0.68	6.78	8	1	0.26	43.6	51	39
100	8	01	0.215	4.76	5	1	0.667	64.00	64	64
500	12		1.413	59.40	77	28	30.70	500.0	500	500
500	12	01	0.775	44.24	55	1	39.339	500.0	500	500
2 ¹¹	16		479.27	604.4	663	548	2443.26	2048.0	2048	2048
2 ¹²	18		7479.56	1994.6	2188	1844	22872.65	4096.0	4096	4096

TABLE 5. Code Generation V.

N	ℓ	bsid ₂				segd ₄			
		<i>time</i>	<i>avg</i>	<i>max</i>	<i>min</i>	<i>time</i>	<i>avg</i>	<i>max</i>	<i>min</i>
100	8	0.54	19.4	21	18	0.66	100.0	100	100
500	12	13.17	172.1	182	163	60.05	500.0	500	500
2 ¹¹	16	3136.94	1723.2	1754	1698	7404.81	2048.0	2048	2048

block solid codes consisting of binary Gilbert words with parameter m , where m is a positive integer. These words start with 0 and end with $0^m 1$ and contain no other occurrence of $0^m 1$. They constitute a solid code. For the case of $\ell = 12$ and $m = 3$, this code has 81 words and is conjectured to be a maximum length 12 solid code.

We recall that a maximal block code is not necessarily maximum, that is, having the largest possible number of codewords, for given \mathbf{er} and ℓ . It seems maximum codes are rare and are defined using subtle combinatorial constructions. On the other hand, there are many random maximal codes having lower information rates. Our setup allows one to get a ‘feel’ of the smallest size of a maximum code, for any rational combination of errors.

For the case of block solid codes, we note that the function `pickFrom` in the algorithm `nonMax` has to be modified as the randomly chosen word w should be in OF.

7. CONCLUSIONS

We have presented a unified method for generating error control codes, for any rational combination of errors. The method cannot of course replace innovative code design, but should be helpful in computing various examples of codes. The implementation `codes.py` is available to anyone for download and use [5]. In the implementation for generating codes, we allow one to specify that generated words only come from a certain desirable subset M of A^ℓ , which is represented by a deterministic trellis. This requires changing the function `pickFrom` in `nonMax` so that it chooses randomly words from M .

There are a few directions for future research. One is to work on the efficiency of the implementations, possibly allowing parallel processing, so as to allow generation of block codes having longer block length. Another direction is to somehow find a way to specify that the set of generated codewords is a ‘systematic’ code

so as to allow efficient encoding of information. A third direction is to do a systematic study on how one can map a stochastic channel \mathbf{sc} , like the binary symmetric channel or one with memory, to an error specification \mathbf{er} (representing a combinatorial channel), so as the available algorithms on \mathbf{er} have a useful meaning on \mathbf{sc} as well.

Acknowledgements. The authors are grateful to the referees of this paper. Their excellent comments have helped to improve it in terms of both clarity and correctness.

REFERENCES

- [1] A. Almeida, M. Almeida, J. Alves, N. Moreira and R. Reis, FAdo and GUItar: Tools for automata manipulation and visualization. In *Proc. of CIAA 2009, Sydney, Australia*, edited by S. Maneth. Vol. 5642 of *Lecture Notes in Computer Science*. Springer-Verlag (2009) 65–74.
- [2] J. Berstel, *Transductions and Context-Free Languages*. B.G. Teubner, Stuttgart (1979).
- [3] E.Z. Chen, Computer construction of quasi-twisted two-weight codes. In *Sixth International Workshop on Optimal Codes and Related Topics* (2009) 62–68.
- [4] K. Dudzinski and S. Konstantinidis, Formal descriptions of code properties: decidability, complexity, implementation. *Int. J. Found. Comput. Sci.* **23** (2012) 67–85.
- [5] FAdo, Tools for formal languages manipulation. Accessed in January 2016. Available at: <http://fado.dcc.fc.up.pt/>.
- [6] D.B. Jaffe, Optimal binary linear codes of length ≤ 30 . In *Proc. of the 1998 International Symposium on Information Theory* (1998) 17.
- [7] D.S. Johnson, C.H. Papadimitriou and M. Yannakakis, On generating all maximal independent sets. *Inf. Process. Lett.* **27** (1988) 119–123.
- [8] H. Jürgenenes and S.S. Yu, Solid codes. *Elektron. Informationsverarbeitung. Kybernetik.* **26** (1990) 563–574.
- [9] M.M. Kanté, V. Limouzy, A. Mary and L. Nourine, On the enumeration of minimal dominating sets and related notions. *SIAM J. Discrete Math.* **28** (2014) 1916–1929.
- [10] S. Konstantinidis, C. Meijer, N. Moreira and R. Reis, Implementation of code properties via transducers. In *Proc. of CIAA 2016*, edited by Y.-S. Han and K. Salomaa. Vol. 9705 in *Lecture Notes in Computer Science*. Springer-Verlag (2016) 189–201.
- [11] S. Konstantinidis and P.V. Silva, Maximal error-detecting capabilities of formal languages. *J. Autom. Lang. Comb.* **13** (2008) 55–71.
- [12] C.W.H. Lam, Finding error-correcting codes using computers. In *Information Security, Coding Theory and Related Combinatorics* (2011) 278–284.
- [13] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.* **10** (1966) 707–710.
- [14] V.I. Levenshtein, Maximum number of words in codes without overlaps. *Probl. Inform. Trans.* **6** (1973) 355–357.
- [15] V.I. Levenshtein, Combinatorial problems motivated by comma-free codes. *J. Comb. Des.* **12** (2004) 184–196.
- [16] H. Lewis and C.H. Papadimitriou, *Elements of the Theory of Computation*, 2nd ed. Prentice Hall (1998).
- [17] S. Lin and D.J. Costello Jr, *Error control coding*, 2nd ed. Pearson (2005).
- [18] Z. Liu and M. Mitzenmacher, Codes for deletion and insertion channels with segmented errors. In *Proc. of ISIT, Nice, France, 2007* (2007) 846–849.
- [19] F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error-Correcting Codes*. Amsterdam (1977).
- [20] H. Mercier, V. Bhargava and V. Tarokh, A survey of error-correcting codes for channels with symbol synchronization errors. *IEEE Commun. Surv. Tutor.* **12** (2010) 87–96.
- [21] M. Mitzenmacher and E. Upfal, *Probability and Computing*. Cambridge University Press (2005).
- [22] F. Paluncic, K. Abdel-Ghaffar and H. Ferreira, Insertion/deletion detecting codes and the boundary problem. *IEEE Trans. Inf. Theory* **59** (2013) 5935–5943.
- [23] V.S. Pless and W.C. Huffman, editors. *Handbook of Coding Theory*. Elsevier (1998).
- [24] PyPy, PyPy, a fast, compliant alternative implementation of the Python language. Available at: <http://pypy.org> (2017).
- [25] G. Rozenberg and A. Salomaa, *Handbook of Formal Languages*, Vol. I. Springer-Verlag, Berlin (1997).
- [26] C.E. Shannon and W. Weaver, *The Mathematical Theory of Communication*. University of Illinois Press, Urbana (1949).
- [27] H.J. Shyr, *Free Monoids and Languages*, 2nd ed. Hon Min Book Company, Taichung (1991).
- [28] T.G. Swart and H.C. Ferreira, A note on double insertion/deletion correcting codes. *IEEE Trans. Inf. Theory* **49** (2003) 269–273.