GENERATING ERROR CONTROL CODES WITH AUTOMATA AND TRANSDUCERS

Stavros Konstantinidis[(A)] Nelma Moreira[(B)]
Rogério Reis[(B)]

[(A)]Department of Mathematics and Computing Science
Saint Mary's University
Halifax, Nova Scotia, Canada
Email: s.konstantinidis@smu.ca

[(B)]CMUP & DCC
Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 4169007 Porto Portugal
Email: {nam,rvr}@dcc.fc.up.pt

Abstract
We introduce the concept of an f-maximal error-detecting block code, for some parameter f between 0 and 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 constructing a maximal error-detecting code is a computationally hard problem. We present a randomized algorithm that takes as input two positive integers N, ℓ, a probability value f, and a specification of the errors permitted in some application, and generates an error-detecting, or error-correcting, block code having up to N codewords of length ℓ. If the algorithm finds less than N codewords, then those codewords constitute a code that is f-maximal with high probability. The error specification is modelled as a (nondeterministic) transducer, which allows one to model 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.

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

[(A)]Research supported by NSERC.

[(B)]Research supported by FCT project UID/MAT/00144/2013.

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* **er** for specifying the error situations permitted in a *channel*—this could be any communication or storage medium. An error specification **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.

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 substitutions 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) [13]. 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 [16]. 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 almost exclusively for specific substitution error types [3, 10].

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 two randomized algorithms: the first one decides (up to a certain degree of confidence) whether a given block code C is maximal **er**-detecting for a given error specification **er**. The second algorithm is given an error specification **er**, an **er**-detecting block code $C \subseteq A^\ell$ (which could be omitted), and an integer $N > 0$, and attempts to add to C N new words of length ℓ resulting into a new **er**-detecting code. If less than N words get added then either the new code is 95%-maximal or the chance that a randomly chosen word can be added is less than 5%. 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, 7]. In Section 6., we discuss a few more points on channel modelling and present some tests of the randomized algorithms 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 [21], and are a powerful computational tool for processing sets of words—see [2] and pg 41–110 of [20].

A *transducer* is a 5-tuple[1] $\mathbf{t} = (S, A, I, T, F)$ 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

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

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 [20]. 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

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

L(**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 (pg 1635–1764 of [19]), and trellises. A *trellis* is an automaton accepting a block code, and has one initial and one final state (pg 1989–2117 of [19]). In the case of a trellis **a** we refer to L(**a**) also as the *code represented by* **a**. A piece of notation that is useful in the next section is the following, where W is any language,

$$\mathbf{t}(W) \;=\; \bigcup_{w\in W} \mathbf{t}(w) \tag{1}$$

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

Fig. 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 $\mathbf{er} = \mathtt{sub}_2$, we have $00101 \in \mathtt{sub}_2(00000)$ because on input 00000, the transducer $\mathtt{sub}_2$ 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 $\mathtt{id}_2$ in Fig. 1 by removing state t_2, then we get the error specification $\mathtt{id}_1$ representing the channel that allows up to 1 symbol to be deleted or inserted in the input word; then

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

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, ε = 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.

The concepts of error-detection and -correction mentioned in the introduction are phrased more rigorously in the next definition. This definition is adapted from [9], 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 2.2 *Let C be a block code of length ℓ and let* **er** *be an error specification. We say that C is* **er**-detecting *if*

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

We say that C is **er**-correcting *if*

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

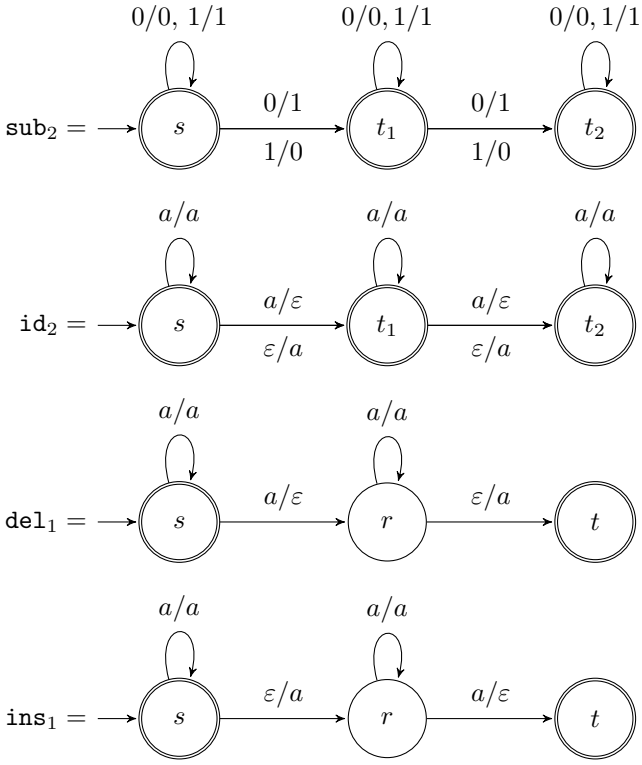

Figure 1: Examples of error specifications (input-preserving transducers). $\underline{\texttt{sub}_2}$: uses the binary alphabet $\{0,1\}$. On input x, $\texttt{sub}_2$ outputs x, or any word that results by performing one or two substitutions in x. The latter case is when $\texttt{sub}_2$ takes the 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. In the following, a is any symbol of an alphabet A. $\underline{\texttt{id}_2}$: On input x, $\texttt{id}_2$ outputs a word that results by inserting and/or deleting at most 2 symbols in x. $\underline{\texttt{del}_1, \texttt{ins}_1}$: considered in [18]. On input x, $\texttt{del}_1$ outputs either x, or any word that results by deleting exactly one symbol in x and then inserting a symbol at the end of x.

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

Referring to Fig. 1, we have that a block code C is $\texttt{sub}_2$-detecting iff the min. Hamming distance of C is > 2. A block code C is $\texttt{id}_2$-detecting iff the min. Levenshtein distance of C is > 2 [11]. 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 min. Hamming (resp. Levenshtein) distance of C is the min. Hamming (resp. Levenshtein) distance of any two different C-words.

From a logical point of view (see Lemma 2.3 below) error-detection subsumes the concept of error-correction. This connection is stated already in [9] 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.

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

Lemma 2.3 *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 [9]. 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. □

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,

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

Let $\mathbf{er}$ be an error specification, let $C \subseteq A^\ell$ be an $\mathbf{er}$-detecting block code, and let $w \in A^\ell \setminus C$. In [4], the authors show that

$$C \cup \{w\} \text{ is } \mathbf{er}\text{-detecting if and only if } w \notin (\mathbf{er} \vee \mathbf{er}^{-1})(C). \tag{2}$$

Definition 2.4 *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) above implies that

$$C \text{ is maximal } \mathbf{er}\text{-detecting if and only if } A^\ell \setminus (\mathbf{er} \vee \mathbf{er}^{-1})(C) = \emptyset. \tag{3}$$

Definition 2.5 *The* maximality index *of a block code $C \subseteq A^\ell$ w. r. t. a channel* $\mathbf{er}$ *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 2.6 *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* **er***-detecting, that is,*

$$\mathrm{maxind}(C, \mathbf{er}) \;=\; \mathtt{Pr}\Big[w \textit{ cannot be added into } C\Big].$$

Proof. The first statement follows from Definition 2.5 and condition (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 $w \in A^\ell \cap (\mathbf{er} \vee \mathbf{er}^{-1})(C)$. □

3. Generating Error Control Codes

We turn now our attention to algorithms processing error specifications and sets of words. Our main goal is to compute, for any given error specification **er**, integer $N > 0$, and deterministic trellis **a** representing some **er**-detecting code C, a deterministic trellis accepting a superset of C and containing either N new words, or fewer than N new words such that it is close to maximal **er**-detecting with high probability. Solving this problem can be used to also solve the problem of generating an **er**-detecting block code of up to N words of length ℓ, for given **er** and integers $N, \ell > 0$, such that, if the generated code has fewer than N words, then it is close to maximal **er**-detecting with high probability. 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.

For computational complexity considerations, the *size* $|\mathbf{m}|$ of a finite state machine (automaton or transducer) **m** is the number of states plus the sum of the sizes of the transitions. The size of a transition is 1 plus the 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.

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

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

Remark 3.1 *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* **a** *contains no transition with label ε. First, if necessary, we convert* **t** *to standard form. Second, if* **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 **b** *is $O(|\mathbf{a}||\mathbf{t}|)$. The required automaton $(\mathbf{a} \triangleright \mathbf{t})$ is the trim version of* **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 $\mathbf{m}$.)

```
nonMax (er, a, f, ε)
    b := (a ▷ (er ∨ er^{-1}));
    n := 1 + ⌊ 1/(4ε(1 − f)^2) ⌋;
    ℓ := the length of the words in L(a);
    tr := 1;
    while (tr ≤ n):
        w := pickFrom(A, ℓ);
        if (w not in L(b)) return w;
        tr := tr+1;
    return None;
```

Figure 2: Algorithm `nonMax`—see Theorem 3.2.

Next we present our randomized algorithms—we use [17] as reference for basic concepts. We assume that we have available to use in our algorithms an ideal method $\texttt{pickFrom}(A, \ell)$ that chooses uniformly at random a word in A^ℓ. A randomized algorithm $R(\cdots)$ 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.

Theorem 3.2 *Consider the algorithm* `nonMax` *in Fig. 2, which takes as input an error specification* $\mathbf{er}$*, a trellis* $\mathbf{a}$ *accepting an* $\mathbf{er}$*-detecting code, and two numbers* $f, \varepsilon \in [0, 1]$*.*

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

$$\texttt{Pr}\Big[\texttt{nonMax } \textit{returns } \texttt{None}\Big] < \varepsilon.$$

3. *The time complexity of* `nonMax` *is* $O\Big(\ell|\mathbf{a}||\mathbf{er}|\Big/(\varepsilon(1-f)^2)\Big)$.

Proof. The first statement follows from statement (2) in the previous section, as any w returned by the algorithm is not in $(\mathbf{er} \vee \mathbf{er}^{-1})(\mathrm{L}(\mathbf{a}))$. For the second statement, suppose that the code $\mathrm{L}(\mathbf{a})$ is not f-maximal $\mathbf{er}$-detecting. Let `Cnt` be the random variable whose value is the value of tr $-$ 1 at the end of execution of the randomized algorithm `nonMax`. Then, `Cnt` counts the number of words that are in $\mathrm{L}(\mathbf{a})$ out of n randomly chosen words w. Thus `Cnt` is binomial: the number of successes (words w in $\mathrm{L}(\mathbf{b})$) in n trials. So $E(\texttt{Cnt}) = np$, where $p = \texttt{Pr}\Big[w \in \mathrm{L}(\mathbf{b})\Big]$. By the definition of n in `nonMax`, we get $1/(4n(1-f)^2) < \varepsilon$. Now consider the Chebyshev inequality, $\texttt{Pr}\Big[|X - E(X)| \geq a\Big] \leq \sigma^2/a^2$, where $a > 0$ is arbitrary and σ^2 is the variance of some random variable X. For $X = \texttt{Cnt}$ the variance is $np(1-p)$, and we get

$$\texttt{Pr}\Big[\ |\texttt{Cnt}/n - p| \geq 1 - f\ \Big] < \varepsilon,$$

where we used $a = n(1-f)$ and the fact that $p(1-p) \leq 1/4$.

Using Lemma 2.6 and the assumption that $\mathrm{L}(\mathbf{a})$ is not f-maximal, we have that $\mathrm{maxind}(\mathrm{L}(\mathbf{a}), \mathbf{er}) < f$, which implies $\mathtt{Pr}\Big[w \in \mathrm{L}(\mathbf{b})\Big] < f$; hence, $p < f$. Then

$$\begin{aligned}
&\mathtt{Pr}\Big[\ \mathtt{nonMax}\ \text{returns}\ \mathtt{None}\ \Big] \\
&= \mathtt{Pr}\Big[\mathtt{Cnt} = n\Big] \\
&= \mathtt{Pr}\Big[\mathtt{Cnt}/n = 1\Big] \\
&= \mathtt{Pr}\Big[\mathtt{Cnt}/n - p = 1 - p\Big] \\
&\leq \mathtt{Pr}\Big[|\mathtt{Cnt}/n - p| \geq 1 - p\Big] \\
&\leq \mathtt{Pr}\Big[|\mathtt{Cnt}/n - p| \geq 1 - f\Big] \ < \varepsilon,
\end{aligned}$$

as required.

For the third statement, we use standard results from automaton theory, [20], and Remark 3.1. 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 $w \in \mathrm{L}(\mathbf{b})$ can be done in time $O(|w||\mathbf{b}|) = O(\ell|\mathbf{b}|)$. Thus, the algorithm works in time $O(\ell|\mathbf{a}||\mathbf{er}|\,/(\varepsilon(1-f)^2))$. □

Remark 3.3 *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 ($\mathrm{L}(\mathbf{a}) \cup \{w\}$ *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 $O\Big(|\mathbf{a}|^2|\mathbf{er}|\Big/\big(\varepsilon(1-f)^2\big)\Big)$*. This is because testing whether* $\mathrm{L}(\mathbf{v})$ *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 Fig. 3, we present the main algorithm for adding new words into a given deterministic trellis $\mathbf{a}$.

Remark 3.4 *In some sense, algorithm* `makeCode` *generalizes to arbitrary error specifications the idea used in the proof of the well-known Gilbert-Varshamov bound [15] for the largest possible block code* $M \subseteq A^\ell$ *that is* $\mathtt{sub}_k$*-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"* $\mathtt{sub}_{2k}(u)$*, for all* $u \in M$*. In that case, we have that* $\mathtt{sub}_k^{-1} = \mathtt{sub}_k$ *and* $(\mathtt{sub}_k^{-1} \circ \mathtt{sub}_k) = \mathtt{sub}_{2k}(u)$*. 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"* $(\mathbf{er} \vee \mathbf{er}^{-1})(\mathrm{L}(\mathbf{c}))$*.*

Theorem 3.5 *Algorithm* `makeCode` *in Fig. 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* $\mathrm{L}(\mathbf{a})$ *is* $\mathbf{er}$*-detecting,*

```
makeCode (er, a, N)
    W := empty list;   c:= a
    cnt := 0;    more := True;
    while (cnt < N and more)
        w := nonMax (er, c, 0.95, 0.05);
        if (w is None) more := False;
        else {add w to c and to W;   cnt := cnt+1;}
    return c, W;
```

Figure 3: Algorithm `makeCode`—see Theorem 3.5. 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 0.95 and 0.05, as they seem to work well in practical testing.

and returns a deterministic trellis $\mathbf{c}$ *and a list* W *of words such that the following statements hold true:*

1. $\mathrm{L}(\mathbf{c}) = \mathrm{L}(\mathbf{a}) \cup W$ *and* $\mathrm{L}(\mathbf{c})$ *is* $\mathbf{er}$*-detecting,*
2. *If* W *has less than* N *words, then either* $\mathrm{maxind}(\mathrm{L}(\mathbf{c}), \mathbf{er}) \geq 0.95$ *or the probability that a randomly chosen word from* A^ℓ *can be added in* $\mathrm{L}(\mathbf{c})$ *is* < 0.05*.*
3. *The algorithm runs in time* $O\Big(\ell N|\mathbf{er}||\mathbf{a}| + \ell^2 N^2|\mathbf{er}|\Big)$*.*

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 Theorem 3.2: any word w returned by `nonMax` is such that $\mathrm{L}(\mathbf{c}_i) \cup \{w\}$ is $\mathbf{er}$-detecting. For the second statement, assume that, at the end of execution, W has $< N$ words and $\mathrm{L}(\mathbf{c})$ is not 95%-maximal. By the previous theorem, this means that the random process $\texttt{nonMax}(\mathbf{er}, \mathbf{c}, 0.95, 0.05)$ returns `None` with probability < 0.05, as required. For the third statement, as the loop in the algorithm `nonMax` performs a fixed number of iterations (=2 000), we have that the cost of `nonMax` is $O(\ell|\mathbf{c}_i||\mathbf{er}|)$. The cost of adding a new word w of length ℓ to $\mathbf{c}_{i-1}$ is $O(\ell)$ 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 $O(\ell|\mathbf{er}|(|\mathbf{a}| + i\ell))$. As there are up to N iterations the total cost is

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

□

Remark 3.6 *In the algorithm* `makeCode`*, attempting to add only one word into* $\mathrm{L}(\mathbf{a})$ *(case of* $N = 1$*), requires time* $O(\ell|\mathbf{er}||\mathbf{a}| + \ell^2|\mathbf{er}|)$*, which is of polynomial magnitude. This case is equivalent to testing whether* $\mathrm{L}(\mathbf{a})$ *is maximal* $\mathbf{er}$*-detecting, which is shown to be a hard decision problem in Theorem 4.1.*

Remark 3.7 *In the version of the algorithm* `makeCode` *where the initial trellis* $\mathbf{a}$ *is omitted, the time complexity is* $O(\ell^2 N^2|\mathbf{er}|)$*. 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

Our motivation for considering randomized algorithms is that the *embedding problem* is computationally hard: given a deterministic trellis **d** and an error specification **er**, compute (using a deterministic algorithm) a trellis that represents a maximal **er**-detecting code containing L(**d**). By computationally hard, we mean that a decision version of the embedding problem is coNP-hard. This is stated next. The proof can be found in [8].

Theorem 4.1 *The following decision problem is coNP-hard.*

Instance: *deterministic trellis* **d** *and error specification* **er**.
Answer: *whether* L(**d**) *is maximal* **er**-*detecting.*

5. Implementation and Use

All main algorithmic tools have been implemented over the years in the Python package FAdo [1, 5, 7]. Many aspects of the new module FAdo.codes are presented in [7]. Here we present methods of that module pertaining to generating codes.

Assume that the string `d1` contains a description of the transducer $\texttt{del}_1$ 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 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` representing the $\texttt{del}_1$-detection property over the alphabet {0,1}. Then, it constructs an automaton `a` representing a $\texttt{del}_1$-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(`a`) is $\texttt{del}_1$-detecting and returns a witness of non-error-detection (= pair of codewords u, v with $v \in \texttt{del}_1(u)$), or (`None`, `None`)—of course, in the above example it would return (`None`, `None`). The method `nonMaximalW(a, m)` tests

whether the code L(a) is maximal $\mathtt{del}_1$-detecting and returns either a word $v \in \mathrm{L}(\mathtt{m}) \setminus \mathrm{L}(\mathtt{a})$ such that $\mathrm{L}(\mathtt{a}) \cup \{v\}$ is $\mathtt{del}_1$-detecting, or `None` if L(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 Theorem 4.1), which motivated us to consider the randomized version `nonMax` in this paper. For any error specification **er** and trellis `a`, the method `notSatisfiesW(a)` can be made to work in time $O(|\mathbf{er}||\mathtt{a}|^2)$, which is of polynomial complexity. The operation '`&`' combines error-detection properties. Thus, the second call to `makeCode` constructs a code that is $\mathtt{del}_1$-detecting and $\mathtt{sub}_2$-detecting (=$\mathtt{sub}_1$-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 2.2 'block code C' with 'language L'. Let $\mathbf{er}, \mathbf{er}_1, \mathbf{er}_2$ be error specifications. By Definition 2.2 and using standard logical arguments, it follows that*

1. *L is $\mathbf{er}_1$-detecting and $\mathbf{er}_2$-detecting, if and only if L is $(\mathbf{er}_1 \vee \mathbf{er}_2)$-detecting;*
2. *L is $\mathbf{er}^{-1}$-detecting, if and only if it is $\mathbf{er}$-detecting, if and only if it is $(\mathbf{er}^{-1} \vee \mathbf{er})$-detecting.*

The inverse of $\mathtt{del}_1$ is $\mathtt{ins}_1$ and is shown in Fig. 4, where recall it results by simply exchanging the order of the two words in all the labels in $\mathtt{del}_1$. By statement 2 of the above remark, the $\mathtt{del}_1$-detecting codes are the same as the $\mathtt{ins}_1$-detecting ones, and the same as the $(\mathtt{del}_1 \vee \mathtt{ins}_1)$-detecting ones—this is shown in [18] 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 the next figures, Fig. 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 $C \subseteq$ 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 block 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 [22] (they are also called codes without overlaps in [12]). The transducer `ov` in Fig. 6 is such that any block code $C \subseteq$ 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 [6].

For $\varepsilon = 0.05$ and $f = 0.95$, the value of n in `nonMax` is 2 000. We performed several executions

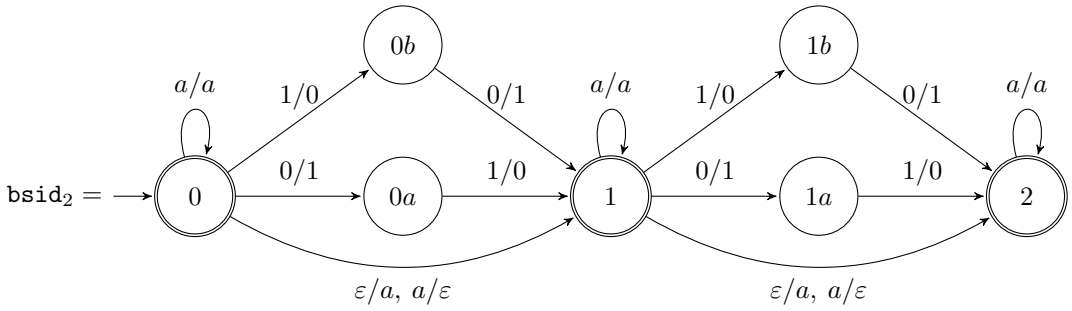

Figure 4: The channel specified by $\texttt{bsid}_2$ allows up to two errors in the input word. Each of these errors can be a deletion, an insertion, or a bit shift: a 10 becomes 01, or a 01 becomes 10. The alphabet is $\{0, 1\}$.

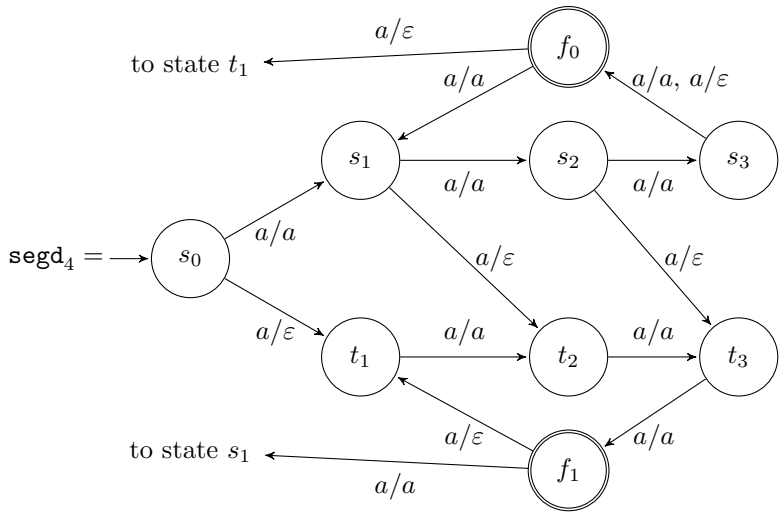

Figure 5: Transducer for the segmented deletion channel of [14] with parameter $b = 4$. In each of the length b consecutive segments of the input word, at most one deletion error occurs. The length of the input word is a multiple of b. By Lemma 2.3, $\texttt{segd}_4$-correction is equivalent to $(\texttt{segd}_4^{-1} \circ \texttt{segd}_4)$-detection.

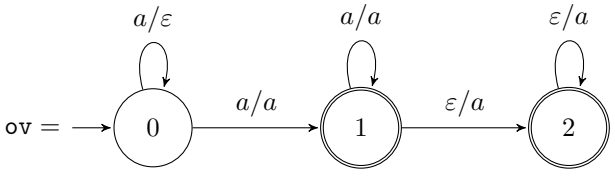

Figure 6: This input-preserving transducer deletes a prefix of the input word (a possibly empty prefix) and then inserts a possibly empty suffix at the end of the input word.

of the algorithm `makeCode` on various error specifications using $n = 2\,000$, no initial trellis, and

alphabet $A = \{0, 1\}$.

N=, ℓ=, end=	$\mathtt{id}_2$	$\mathtt{del}_1$	$\mathtt{sub}_2$	$\mathtt{bsid}_2$	ov
100, 8,	18, 20, 23	37, 42, 51	15, 16, 18	17, 19, 21	01, 07, 08
100, 7,	10, 12, 13	20, 23, 28	09, 10, 13	11, 11, 13	03, 04, 05
100, 8, 1	11, 13, 14	39, 50, 64	09, 10, 11	09, 12, 13	01, 05, 06
100, 8, 01	06, 07, 08	64, 64, 64	04, 06, 08	06, 07, 09	01, 04, 05
500, 12,	177, 182, 188	500, 500, 500	148, 157, 162	169, 173, 178	51, 59, 63
500, 13,	318, 327, 334		272, 273, 278	302, 303, 309	43, 111, 120

In the above table, the first column gives the values of N and ℓ, and if present and nonempty, the pattern that all codewords should end with (1 or 01). For each entry in an '$N = 100$' row, we executed `makeCode` 21 times and reported smallest, median, and largest sizes of the 21 generated codes. For $N = 500$, we reported the same figures by executing the algorithm 5 times. For example, the entry 37,42,51 corresponds to executing `makeCode` 21 times for **er** = $\mathtt{del}_1$, $\ell = 8$, end = ε. The entry 64,64,64 corresponds to the systematic code of [18] whose codewords end with 01, and any of the 64 6-bit words can be used in positions 1–6. The entry for '$\ell = 7$, end = ε, **er** = $\mathtt{sub}_2$' corresponds to 2-substitution error-detection which is equivalent to 1-substitution error-correction. Here the Hamming code of length 7 with 16 codewords has a maximum number of codewords for this length. Similarly, the entry for '$\ell = 7$, **er** = $\mathtt{id}_2$' corresponds to 2-synchronization error-detection which is equivalent to 1-synchronization error-correction. Here the Levenshtein code [11] of length 8 has 30 codewords. We recall that a maximal code is not necessarily maximum, that is, having the largest possible number of codewords, for given **er** and ℓ. It seems maximum codes are rare, but there are many random maximal ones having lower rates. The $\mathtt{del}_1$-detecting code of [18] has higher rate than all the random ones generated here.

For the case of block solid codes (last column of the table), 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 **sc**, like the binary symmetric channel or one with memory, to an error specification **er** (representing a combinatorial channel), so as the available algorithms on **er** have a useful meaning on **sc** as well.

References

[1] André Almeida, Marco Almeida, José Alves, Nelma Moreira, and Rogério Reis. FAdo and GUItar: Tools for automata manipulation and visualization. In *Proceedings of CIAA 2009, Sydney, Australia*, volume 5642 of *Lecture Notes in Computer Science*, pages 65–74, 2009.

[2] Jean Berstel. *Transductions and Context-Free Languages*. B.G. Teubner, Stuttgart, 1979.

[3] Eric Z. Chen. Computer construction of quasi-twisted two-weight codes. In *Sixth International Workshop on Optimal Codes and Related Topics*, pages 62–68. 2009.

[4] Krystian Dudzinski and Stavros Konstantinidis. Formal descriptions of code properties: decidability, complexity, implementation. *International Journal of Foundations of Computer Science*, 23:1:67–85, 2012.

[5] FAdo. Tools for formal languages manipulation. Accessed in Jan. 2016. URL: `http://fado.dcc.fc.up.pt/`.

[6] Helmut Jürgenesen and S. S. Yu. Solid codes. *Elektron. Informationsverarbeit. Kybernetik.*, 26:563–574, 1990.

[7] Stavros Konstantinidis, Casey Meijer, Nelma Moreira, and Rogério Reis. Implementation of code properties via transducers. In Yo-Sub Han and Kai Salomaa, editors, *Proceedings of CIAA 2016*, number 9705 in Lecture Notes in Computer Science, pages 189–201, 2016. ArXiv version: Symbolic manipulation of code properties. arXiv:1504.04715v1, 2015.

[8] Stavros Konstantinidis, Nelma Moreira, and Rogério Reis. Channels with synchronization/substitution errors and computation of error control codes. *CoRR*, arXiv:1601.06312, 2016. http://arxiv.org/abs/1601.06312v2.

[9] Stavros Konstantinidis and Pedro V. Silva. Maximal error-detecting capabilities of formal languages. *J. Automata, Languages and Combinatorics*, 13(1):55–71, 2008.

[10] Clement W. H. Lam. Finding error-correcting codes using computers. In *Information Security, Coding Theory and Related Combinatorics*, pages 278–284. 2011.

[11] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Dokl.*, 10:707–710, 1966.

[12] Vladimir I. Levenshtein. Maximum number of words in codes without overlaps. *Probl. Inform. Transmission*, 6(4):355–357, 1973.

[13] Shu Lin and Daniel J. Costello Jr. *Error control coding, 2nd Edition*. Pearson, 2005.

[14] Zhenming Liu and Michael Mitzenmacher. Codes for deletion and insertion channels with segmented errors. In *Proceedings of ISIT, Nice, France, 2007*, pages 846–849, 2007.

[15] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. Amsterdam, 1977.

[16] Hugues Mercier, Vijay Bhargava, and Vahid Tarokh. A survey of error-correcting codes for channels with symbol synchronization errors. *IEEE Communic. Surveys & Tutorials*, 12(1):87–96, 2010.

[17] Michael Mitzenmacher and Eli Upfal. *Probability and Computing*. Cambridge Univ. Press, 2005.

[18] Filip Paluncic, Khaled Abdel-Ghaffar, and Hendrik Ferreira. Insertion/deletion detecting codes and the boundary problem. *IEEE Trans. Information Theory*, 59(9):5935–5943, 2013.

[19] V. S. Pless and W. C. Huffman, editors. *Handbook of Coding Theory*. Elsevier, 1998.

[20] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, Vol. I*. Springer-Verlag, Berlin, 1997.

[21] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, 1949.

[22] H. J. Shyr. *Free Monoids and Languages*. Hon Min Book Company, Taichung, second edition, 1991.