CODEGEN: The Generation and Testing of DNA Code Words

David E. Kephart
Department of Mathematics
University of South Florida
Tampa, FL 33620
Email: dkephart@mail.usf.edu

Jeff LeFevre
Department of Computer Science
and Engineering
University of South Florida
Tampa, FL 33620
Email: jlefevre@ieee.org

***Abstract*— With this paper we present algorithms to generate and test DNA code words that avoid unwanted cross hybridizations. Methods from the theory of codes based on formal languages are employed. These algorithms are implemented in user-friendly software, CODEGEN, which contains a collection of language-theoretic objects adaptable to various related tasks. Lists of code words may be stored, viewed, altered and retested. Implemented in Visual Basic 6.0, its interface allows for lists of code words to be assembled at varying levels of acceptability from a single main window.**

I. INTRODUCTION

A. Motivation

DNA offers us the prospect of massive parallelism in computing. A few basic obstacles to this, however, must be overcome. The simple property of Watson-Crick complementarity, which allows the precise chemical matching of a denatured, single-stranded DNA molecule, or *oligonucleotide* with an oppositely-oriented complementary strand allows the transfer of information it contains. By assembling particular strands and encoding a problem in these strands, it has been shown that certain NP-complete problems may be solved.

This same property can compromise the results of a DNA program if the encoding is selected carelessly. The information conveyed in ill-chosen strands will be lost if they bond (*hybridize*) with code words. Information will be lost if code words of varied length are used and multi-word strands have more than one possible interpretation.

Further, not all DNA nucleotides are born chemically equal. The higher the percentage of cytosine and guanine in a strand, the higher its melting temperature. Leaving them out means reducing the overall number of useable distinct code words of a given length.

The search for potential unwanted coding ambiguities between DNA words demands exponentially increasing amounts of computation time.

This paper presents algorithms which use a theorem from coding theory and language-theoretic properties to address this dilemma for the DNA programmer. These have been implemented in software in a program – CODEGEN – which performs the various necessary tests in polynomial time. It generates or verifies these properties in a language from a user-friendly interface and enables the storage of languages. It is the extension of software presented in [13].

While the program selects DNA settings by default, the interface and object-oriented implementation lend themselves to more general applications.

In section II-A we define necessary terms from coding theory and present the theorem that drives the main algorithm of CODEGEN. In section III we present the algorithms which vary the level of "goodness" demanded in tested code words, followed by pseudo-code for the main algorithm. In section IV-C, we discuss the object-oriented library created to implement the algorithm, and pseudo-code for a typical algorithm made possible by this approach. In section V we show the details of the user interface. We conclude with several remarks about present limitations and future development of this project.

B. Past and present work on this problem

Our starting point in this is an elaboration of the framework set up by Kari, Hussini, and Konstantinides [17]. The use of computers to assist in the synthetic self-assembly of DNA strands dates from the first laboratory construction of such molecules. The program presented in [22], starts from the topological characteristics of the molecules required in computation and proceeds to the comparison of subsequences. Others include those proposed by Baum [2], Li [16], the DNASequenceGenerator of Felkamp, et. al.[8] and [9], the template-based methods of Arita and Kobayashi[1], and Garzon's amplification of seed sequences via the tensor products of collections of words[12].

As mentioned in [13] these approaches employ Hamming distance and therefore standard binary coding theory methods. Most produce code word sequences of fixed length. They avoid inter-molecular hybridizations between the DNA strands and between each DNA sequence and the complements of other strands.

Variable length code words are necessary in certain DNA computations ([4], [5], [23], for example). CODEGEN uses methods from algebraic (and hence n-ary) coding theory to produce words that not only avoid intermolecular crosshybridizations, but may be of variable length. With the proper set-up Seeman's program also can give variable length code

0-7803-8515-2/04/$20.00 ©2004 IEEE

words. CODEGENś interface allows the user direct, intuitive access to this and all of its capabilities.

In addition, CODEGEN avoids both *intra*-molecular crosshybridizations and can produce sets of code words with properties closed under the Kleene* operation. These could generate infinite sets of "good" code words. We therefore begin with a formal definition of what we mean by a "good" set of DNA code words.

II. THE PROBLEM: WHAT IS A "GOOD" DNA CODE WORD?

A. Definitions

A finite alphabet, Σ, is a finite collection of symbols. A language is a subset of the set of all possible concatenations of symbols from, or *over* an alphabet. We denote by Σ^* the set of all possible words over Σ. This set includes the empty word. The set of words over Σ of positive length we denote by Σ^+, and the set of all possible words over Σ of length n we denote by Σ^n. A language over Σ is then $X \subset \Sigma^*$. We denote by X^n $(n \geq 1)$ the set of all possible concatenations of n members of X. We denote by X^* all possible concatenations of words in X.

If a function $\theta : \Sigma \to \Sigma$ has the property that $\theta(\theta(a)) = \theta^2(a) = a$ for each symbol a in Σ, then θ is an *involution on* Σ with two possible obvious extensions to Σ^*. If, for every $x, y \in \Sigma^*$, $\theta(xy) = \theta(x)\theta(y)$, then θ is a *morphism*, while if $\theta(xy) = \theta(y)\theta(x)$, then θ is an *antimorphism*.

Definition 1: Let Δ represent the *DNA alphabet*, i.e., the set {adenine, cytosine, guanine, thymine}, or more succinctly, $\{A, C, G, T\}$.

The nucleotides of a DNA strand bind at one end at the $5'$ location on a carboxyl ring and at the other end at the $3'$ location into an oligonucleotide. This orients the strand. Taking the $5'$ end by convention as the beginning position, the sequence of nucleotides in a DNA strand form a word over Δ. Watson-Crick complementarity refers to the hydrogen bonds which form between oppositely-oriented A and T nucleotides and between oppositely-oriented C and G nucleotides. This extends to entire strands, so we have the following.

Definition 2: Let $\rho : \Delta^* \to \Delta^*$ be the antimorphic extension to Δ^* of the mapping such that $\rho : A \mapsto T$, $\rho : T \mapsto A$, $\rho : C \mapsto G$, and $\rho : G \mapsto C$. Let σ map a DNA word to its reverse (e.g., $\sigma(AC) = CA$). Then $\theta = \sigma \circ \rho$ is an antimorphic involution on Δ^* representing Watson-Crick complementarity.

We may now define formally the coding property and unwanted cross-hybridizations between or within members of a language over Δ^* ($X \subset \Delta^*$) – the "good" set of code words we are after. These definitions are therefore placed in a more general setting.

Let Σ be an alphabet, let X be a subset of Σ^*, and let θ be a morphic or antimorphic involution on Σ^*.

1) We require that X be a code, in the sense that if $x = x_1x_2x_3 \cdots x_r = y_1y_2y_3 \cdots y_s$, $x_1, x_2, \ldots, x_r, y_1, y_2, \ldots, y_s \in X$, then $r = s$ and $x_i = y_i$, $1 \leq i \leq r$.
2) The involution of an element of X should not be a proper subword of an element of X. Stated formally, $\Sigma^+\theta(X)\Sigma^* \cap X = \emptyset$ and $\Sigma^*\theta(X)\Sigma^+ = \emptyset$. Following [17], we call such a code *θ-compliant*.
3) The involution of an element of X should not be the proper subword of the concatenation of two elements of X. $\Sigma^+\theta(X)\Sigma^* \cap \Sigma^2 = \emptyset$ and $\Sigma^*\theta(X)\Sigma^+ \cap \Sigma^2 = \emptyset$. We then call X *θ-free*, also per [17].
4) Consider each element w of X, and the k-length prefix p and k-length suffix s of w. Then $\theta(p)$ and $\theta(s)$ should either not be a subwords of w or, if $px\theta(p)$ is a prefix of w, or $\theta(s)xs$ is a subword of w, then $|x|$ should be less than m_1 or more than m_2. Then we say X is *$\theta(k, m_1, m_2)$-subword compliant*, as first defined in [13]: for $m_1 \leq i \leq m_2$ and $w \in \Sigma^k$, $w\Sigma^i\theta(w)\Sigma^+ \cap X = \emptyset$, and $\Sigma^+\theta(w)\Sigma^i w \cap X = \emptyset$.
5) Suppose that, for some $k \geq 1$ and for all $w \in \Sigma^k$, if for some $a \in \Sigma^+$, $b \in \Sigma^*$ $awb \in X$ or $bwa \in X$, then $\Sigma^*\theta(w)\Sigma^+ \cap X = \Sigma^+\theta(w)\Sigma^* \cap X = \emptyset$. Then we say X is *θ-k*, as first defined first in [14]. No k-length subword of any element of X is the involution of any subword of any element of X.
6) If Σ^+ is replaced by Σ^* in 2, 3, 4, or 5, then X is said to be *strictly* θ-compliant, θ-free, $\theta(k, m_1, m_2)$-subword compliant, or θ-k.

In a set X of DNA code words without property 1 there are sequences of words which cannot be uniquely decoded into sequences from X. Some identifiable code word sequences will "factor" in more than one way.

In a code X of DNA words without property 2 some code word will hybridize with the interior portion of another sequence in the code and neither one will be identifiable in further computation. This is shown in Figure 1(a).

Fig. 1. Intermolecular Cross-Hybridizations. *A violation of θ-compliance is shown in* **(a)**, *while* **(b)** *illustrates a violation of θ-freedom. After hybridization* **(a)** *both* u *and* v *are useless in further DNA computation. After hybridization* **(b)** u, v, *and* w *are all rendered useless.*

In a set of DNA code words without property 3, some code word will "glue" two other code words together and destroy all three for purposes of computation. This is shown in Figure 1(b).

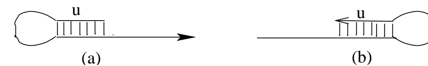

Fig. 2. Intra-Molecular Cross-Hybridizations. *A word is equal to $uv\theta(u)x$, as show in* **(a)**. *If $|u| = k$ and $m_1 < |v| < m_2$, then the word is not $\theta(k, m_1, m_2)$-subword compliant. The similar situation where* u *is the suffix of the word is shown in* **(b)**.

In a DNA code without property 4, with values k, m_1, and m_2 dependent on the specific conditions of the computation,

some sequences can form "hairpin" structures as shown in Figures 2(a) and (b). These strands are then rendered useless.

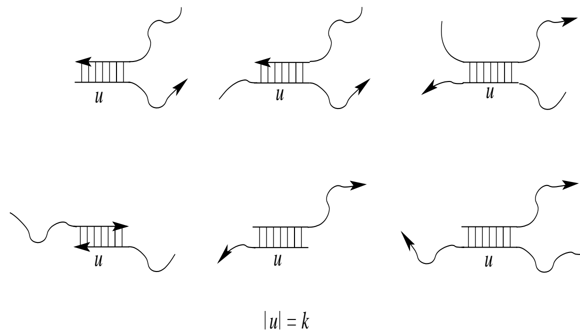

Fig. 3. Cross-Hybridizations Avoided by θ-k Codes

Finally, with property 5, a DNA code avoids the above and various other cross-hybridizations, as shown in Figure 3. In particular, consider the language $Y = X^2$, where X is θ-k. As proven in [14], if Y is θ-k as well, and there are no words in X with fewer than k symbols, then *no* cross-hybridizations in X or X^* are possible.

We now require some definitions from coding theory.

Definition 3 (Flower): The *flower automaton* of a language X over a finite alphabet Σ is the finite state automaton $\mathcal{F}(Q, I, T, E)$ where $I = T = \{0\} \subset Q$ and $E \subset Q \times \Sigma \times Q$ which *recognizes* X in the following sense. If w is a word in X of length s, there is a *path* $\pi = e_1 e_2 \cdots e_s$, $e_1, \ldots, e_s \subset E$ in $\mathcal{F}$ such that $e_i = (q_i, w_i, q_{i+1})$, w_i is the ith symbol of w, and $q_1 = q_{s+1} = 0$, but $q_i \neq 0$ if $1 < i < s + 1$.

We say that w_i is the *label* of e_i, and w is the label of π. As a labeled graph, $\mathcal{F}$ resembles a flower with a petal for each word starting and ending at the same central state.

Definition 4 (Square): The direct product of a flower automaton with itself gives a new automaton $\mathcal{P}(Q', I', T', E')$ with $Q' = Q \times Q$, and $E' \subset Q' \times \Sigma \times Q'$ such that $((q_1, q_2), a, (q_3, q_4)) \in E'$ if and only if $(q_1, a, q_3) \in E$ and $(q_2, a, q_4) \in E$. Then $\mathcal{P}$ is the *square automaton of* X.

The square automaton is *ambiguous* if there exists is a path in $\mathcal{P}$, $(p_0, q_0) = (0, 0) \rightarrow (p_1, q_1) \rightarrow \cdots \rightarrow (p_{n-1}, q_{n-1}) \rightarrow (p_n, q_n) = (0, 0)$, such that $p_i \neq q_i$ for some $1 \leq i < n$. CODEGEN uses the following theorem from coding theory [3] as the basis for its main algorithm.

Theorem 1: A language over a finite alphabet is a code if and only if its square automaton is unambiguous.

Ambiguity in the square automaton of a language X means that X is not a code, for two distinct paths with the same label may then be traced in the flower automaton of X from 0 to 0, so there is a word in X^* equal to two distinct concatenations of words in X, contrary to the definition in item 1.

The following example illustrates this. Suppose $\Sigma = \{a, b\}$ and $X = \{a, ba, aba\} \subset \Sigma^*$. Then in 4(a) we picture the flower of X, $\mathcal{P} = (\{0, 1, 2, 3\}, \{0\}, \{0\}, \{(0, a, 0), (0, b, 1), (1, a, 0), (0, a, 2), (2, b, 3), (3, a, 0)\})$. Figure 4(b) shows the subgraph of the square automaton $\mathcal{P}$ containing the ambiguous path. X is not a code because $a\ ba = aba$.

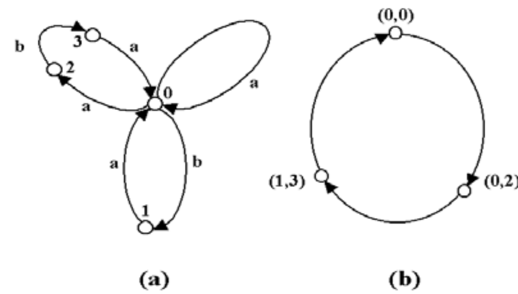

Fig. 4. Theorem 1 Applied. *The language $X = \{a, ba, aba\}$ is not a code.* **(a)** *shows the flower automaton of X, with one possible numbering of the states.* **(b)** *shows the ambiguous path in the square automaton of X.*

B. Combinatorial Facts

CODEGEN implements an algorithm based on Theorem 1. The program attempts to take advantage of combinatorial coding facts to speed up the code word verification and generation processes. For instance, if the request is for a language of words all of the same length or if the language to be tested consists of same-length code words, then it is a code and no check for this property is made.

If Σ has n symbols, and if the involution θ on Σ is such that $\theta(a) \neq a$ for any $a \in X$ (which implies that n is even), then there are limits on how many words are in X if X is θ-k. There are a maximum of $n^k/2$ distinct subwords of length k which can be used in X. Suppose we desire that all k-length subwords be unique in a θ-k code consisting of words of from k_1 to k_2 symbols, where $0 < k_1 < k \leq k_2$. We can form at most

$$\sum_{i=k_1}^{k-1} n^i + \frac{n^k}{2}$$

such words and at most

$$\frac{n^{k_3}}{2(k_3 - k + 1)}$$

words of individual length k_3, where $k_3 \geq k$.

These bounds are sharp if k is composite and θ is antimorphic, as some words are their own complements.

The user has minimum and maximum length, symbol-repetition limits, and symbol content (the percentage of Cs and Gs, that is) in mind depending on the experiment when generating DNA code words. If these add up to a request that is combinatorially impossible to satisfy, CODEGEN will inform the user and will not attempt to answer it.

III. ALGORITHM FOR FINDING AND DETECTING "GOOD" DNA CODE WORDS

The basic algorithms of CODEGEN answer requests for randomly-generated code words with any number of the θ

properties and tests any list of code words for specified θ properties. The structures it uses are easily accessible, so that the most costly computation involved is a string operation. The testing process is recursive, a methodical check for ambiguous paths in a square automaton, as suggested by Theorem 1.

We first present the data structures used and the algorithms which use them. Then as examples we describe here how CHECK and TRACE determine whether the language X is a code and whether X with involution θ is strictly θ-free or strictly θ-compliant. Finally, we give pseudo-code for these routines.

1) FLOWER: is the data structure representing a flower automaton. It is an array of lists. Node k of list p contains $(i_{p,k}, j_{p,k})$ and $a_{p,k}$, where $(i_{p,k}, a_{p,k}, j_{p,k})$ is edge k of "petal" p of the flower automaton.

2) SQUARE: is the structure representing a square automaton such as $\mathcal{P}$. It is an array of two lists. The first has a node for each edge in the square automaton the source of which is (0,0). The second holds a node for each of the other edges, those with sources other than (0,0). The SQUARE and FLOWER expand and contract when a word is added to or deleted from the language.

3) SOURCES: is a list of states in the SQUARE which are sources of edges in SQUARE. This list is depleted by the main algorithm as it finds states which do not lie along the type of path for which it is looking. It is renewed between calls to TRACE.

4) TRACE(initial-states, final-states, check-flag): This is a call to the main algorithm, TRACE. The call returns `TRUE` if a path is found in a square automaton from an initial state to a final state, if that path violates the conditions indicated by the flag. It returns returns `FALSE` if the if the initial states are exhausted without finding such a path.

TRACE callse the routine CHECK with each state in the list of initial states. If the call to CHECK returns `TRUE` all recursion unwinds, and at the top level TRACE returns `TRUE`. Otherwise, recursion unwinds just one level; TRACE sends the next state in its list to CHECK. When a state is exhausted for possible TRACEs, it is removed from SOURCES.

5) CHECK(current-state, final-states, recursion-level, check-flag): This determines whether a given state violates the flagged conditions or not. If it does, CHECK returns `TRUE` and so does TRACE. Otherwise if the current state is in SOURCES, it assembles a list of all targets of edges in SQUARE with source equal to the current state. It calls TRACE with this as the list of initial states. CHECK returns the result of this call. Otherwise, if the current state is not in SOURCES, CHECK returns `FALSE`.

A. Identifying a Code

To verify that X is a code, CODEGEN TRACEs through SQUARE until an ambiguous path is identified – whereupon it returns `TRUE`– or until it exhausts all possible paths or reaches an excessive recursion level. In either of the latter cases it returns `FALSE`.

Thus, the verification begins with the call TRACE({(0,0)}, {(0,0)}, Must_Be_Distinct) on the structure representing $\mathcal{P}$. This call will be passed along to CHECK with the integer 1 (the current recursion level).

When CHECK recieves a state (i, j) and the flag "Must_Be_Distinct," it checks the recursion level. If this is 1, it recurses as described above. If the level is greater than 1 and $i = j$, CHECK returns `FALSE`. Otherwise $i \neq j$ and CHECK sets the flag to "Is_Distinct", before recursing.

When CHECK recieves state (i, j) and the flag "Is_Distinct," it returns `TRUE` if (i, j) is in *final-states*. Otherwise, if (i, j) is in SOURCES, CHECK calls TRACE recursively. If not, CHECK returns `FALSE`.

This works because, according to Theorem 1, a path from $(0, 0)$ to $(0, 0)$ exists in $\mathcal{P}$ such that $i \neq j$ for some state (i, j) on the path if and only if X is not a code. For states (i, j) and (k, l) in SQUARE, $(i, j) \to (k, l)$ in the square automaton if and only if (i) $i = j = 0$, (ii) $k = i + 1$ and $l = j + 1$, or (iii) $k = l = 0$. A check that $i \neq j$ on the first recursion level is sufficient to reject redundant searches, i.e., where the ambiguous path is shorter than the one we have set out upon, for otherwise TRACE is checking a path which reflects a single petal in the flower automaton. TRACE returns `TRUE` therefore if and only if the square automaton is ambiguous.

B. θ Freedom

To check whether a code X is strictly θ-free, CODEGEN forms the flower automaton $\mathcal{F}_\theta$ of the involution of X, i.e., the flower automaton recognizing $\{\theta(x) : x \in X\}$. From this it forms a square automaton $\mathcal{P}_\theta$ using the direct product of $\mathcal{F}_\theta$ with $\mathcal{F}$. This is called INVOLUTION_SQUARE

It then determines whether there is a path in $\mathcal{P}_\theta$ between a state $(0, i)$ and a state $(0, j)$ which passes through at most one state of the form $(k, 0)$, where $k \neq 0$. CODEGEN assembles a list containing all states in INVOLUTION_SQUARE of the form $(0, i)$. TRACE is called with this list as both *initial-states* and *final-states*, and with *check-flag* set to the constant "Two_Words".

When CHECK is called with the flag "Two_Words" or "One_Word", if the recursion level is greater than 1 and the current state is in *final-states* CHECK returns `TRUE`. Otherwise, if *current-state* is of the form $(0, k)$ CHECK will return `FALSE` if the flag is "One_Word" and will set the flag equal to "One_Word" if it is currently "Two_Words". In the latter case or if *current-state* is not of the form $(0, k)$ or if the recursion level is 1, and if *current-state* is in SOURCES, CHECK builds the *next-states* list as described in III-A and returns the result of a recursive call, TRACE(*next-states*, *final-states*, *(updated) check-flag*).

This works because if a code X is not θ-free, then by definition (section II-A, item 3), there is an element $x \in \theta(X)$ and $y,\ z\ \in \Sigma^*$ such that $y\theta(x)z \in X^2$. Let $x = x_1 \cdots x_s$ and $yxz = y_1 y_2 \cdots y_r \theta(x)_1 \theta(x)_2 \cdots \theta(x)_s z_1 z_2 \cdots z_t$, where $0 \leq r = |y|,\ 0 < s = |x|,\ 0 \leq t = |z|$ and $x_i, y_j, z_k \in \Sigma$ for $1 \leq i \leq s, 1 \leq j \leq r$ and $1 \leq k \leq t$. Then $\theta(x)$ is the label of some path $\pi_\theta = e'_1 \cdots e'_s$ in $\mathcal{F}_\theta$. Further, yxz is the label

of some path $\pi_1\pi_2$, for $\pi_1 = f_1 \cdots f_u$, $\pi_2 = g_1 \cdots g_{r+s+t-u}$ petals in $\mathcal{F}$ (or $u = 0$ or $u = r+s+t$ – one of the two can be the empty word). Then the label of $f_{r+1} \cdots f_u$ concatenated with the label of $g_1 \cdots g_{r+s+t-u-r+1} = g_{s+t+1}$ is equal to $\theta(x)$.

Let q_{v_1} be the source of f_{r+1} in $\mathcal{F}$, and let q_{v_i} be the target of f_i in $\mathcal{F}$ for $1 < i < u$. Let $q_{v_{u+i}}$ be the target of g_i ($1 < i \leq s - u$). Let $q'_{v'_i}$, similarly, be the source of e'_i in π' ($1 \leq i \leq s$), where the target of f_s is $0 = q'_{v'_{s+1}} = q'_{v'_1}$. Then $(q'_{v'_i}, q_{v_i}) = q''_{v''_i}$ is in $\mathcal{F}_\theta(Q) \times \mathcal{F}(Q)$. Finally, $q''_{v''_i}$ is the source, and $q''_{v''_{i+1}}$ the target of an edge in $\mathcal{P}_\theta$ for each $1 \leq i \leq s$ by definition, so that these edges form a path we may call $\pi'' = e_1 e_2 \cdots e_s$ in $\mathcal{P}_\theta$. Since π_1 and π_2 are petals in $\mathcal{F}$, there is at most the one state, $q''_{v''_u}$ in the interior of π'' such that $q_{v_u} = 0$ and $q''_{v''_i} \neq 0$.

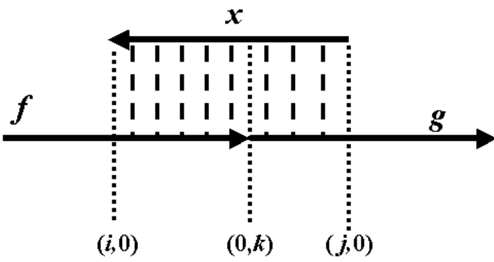

Fig. 5. Justification of the θ-free algorithm. *If X is not θ-free, there exist $x, f, g \in X$ as shown. The square automaton of the flower automaton with the flower of X contains a path between states $(0, i)$ and $(0, j)$ with at most one state of the form $(k, 0)$ ($k \neq 0$).*

Conversely, if such a path may be traced in $\mathcal{P}_\theta$, this shows the existence of a sequence of edges in $\mathcal{F}$ beginning at symbol $r + 1$ ($r \geq 0$) of some petal π_1 and ending either within that petal or, after reaching its end, continuing s edges into another petal π_2 of $\mathcal{F}$, the label of which is equal to $\theta(x)$ for some $x \in X$. But then there exist $y, z \in \Sigma^*$ such that $y\theta(x)z = w_1 w_2$, where w_i is the label of π_i, so X is not θ-free.

C. θ-Compliance

To check whether a code X is strictly θ-compliant, CODEGEN forms $\mathcal{F}_\theta$ and $\mathcal{P}_\theta$ and assembles a list of $\mathcal{P}_\theta$ states with one zero coordinate as described above in III-B.

Then TRACE attempts to find a path in $\mathcal{P}_\theta$ which demonstrates that X is not θ-compliant. This means a call to TRACE, just as in III-B, except with the flag "One_Word". It will return then TRUE if a path in $\mathcal{P}_\theta$ indicates that a word in the code is the complement of some subword of X.

The justification of this algorithm is similar to that for strict θ-freedom. The non-strict versions of both require minor variations on the strict versions.

IV. IMPLEMENTATION OF THE ALGORITHMS

We first present pseudo-code for the main algorithm, followed by computation time calculations and the collection of objects through which CODEGEN is implemented. In the pseudo-code, the I parameter is a list of initial states, F is a list of final states, L is the check level flag passed by reference between the routines, R is the recursion level, and N is the list of states created for purposes of recursion.

M (found in SINGLETRACE IV-A.2) is a restraint on the depth of recursion. Currently this constant is set to f^4, where f is the number of states in $\mathcal{F}$.

A. Main Algorithm

TRACE is the entry-point to the main algorithm. It receives a list of initial states, a list of final states, and a flag indicating what it should look for in the TRACEing of SQUARE or SQUARE_INVOLUTION. It sends these, and the integer 1, as the initial recursion level to the interior function MULTITRACE. It returns the value it receives from MULTITRACE after renewing the SOURCES structure.

Algorithm 1 TRACE(I, F, L)

```
1: Trace ← MULTITRACE(I, F, L, 1)
2: Renew SOURCES
3: Return Trace
```

1) MULTITRACE: receives a list of initial states, a list of terminal states, the flag, and an integer equal to the recursion level. If it has not exhausted the initial state list, it sends the next state on this list, together with the final states, the flag at its current values, and the recursion level to the function SINGLETRACE. If SINGLETRACE returns TRUE, so does MULTITRACE. Otherwise, when the initial state list is exhausted, MULTITRACE returns FALSE.

Algorithm 2 MULTITRACE(I, F, L, R)

```
1: if R = MaximumRecursions then
2:    Return FALSE
3: end if
4: Index ← 0
5: while Index < #F do
6:    Multitrace ← SINGLETRACE(I[Index],F,L,R)
7:    if Multitrace = TRUE then
8:       Return TRUE
9:    end if
10:   Index ← Index+1
11: end while
12: Return FALSE
```

2) SINGLETRACE: is the connection to the actual path exploration accomplished in the function CHECK.

3) CHECK: performs the various checks described in III-A and III-B. After that it returns TRUE or FALSE from a recursive call to TRACE. To make this call, it uses the targets of edges in PATH which have the state it is checking as a source.

B. Computation Time

This algorithm uses calls to standard string routines, and the number of these calls is proportional to the total depth of all

Algorithm 3 SINGLETRACE(S, F, L, R)

1: OldFlag $\leftarrow L$
2: SingleTrace $\leftarrow$ `FALSE`
3: $D \leftarrow D + 1$
4: **if** $R = M$ **then**
5: SingleTrace $\leftarrow$ `FALSE`
6: **else**
7: SingleTrace $\leftarrow$ CHECK(S, F, L, R)
8: **if** SingleTrace=`FALSE` and OldFlag is unaltered **then**
9: Remove S from SOURCES
10: **end if**
11: **end if**
12: $L \leftarrow$ OldFlag
13: Return SingleTrace

Algorithm 4 CHECK(S, F, L, R)

1: **if** L and R meet the required conditions and S is in F **then**
2: Return `TRUE`
3: **else if** S is in SOURCES **then**
4: $L \leftarrow$ new value, if necessary
5: $N \leftarrow$ the targets of edges in SQUARE with source S
6: Return MULTITRACE($N, F, L, R + 1$)
7: **else**
8: Return `FALSE`
9: **end if**

recursions it makes. Since each such call either returns a `TRUE` and halts, or eliminates a target of $(0, 0)$ from SOURCES, i.e., one state in $\mathcal{P}$ or $\mathcal{P}_\theta$, the computation time – at the most – is proportional to $\binom{s}{2}$, where s is the number of states in $\mathcal{P}$ or $\mathcal{P}_\theta$. It is therefore $O(s^2)$.

Suppose that Σ contains n symbols $a_1, \ldots, a_n$. Every edge in the flower has one of these n labels. The number of edges in the flower, l, is the total length of all words in X by construction. If x_i is the number of occurrences of a_i in X, where $\Sigma = \{a_1, \ldots, a_n\}$, then there must be

$$\sum_{i=1}^{n} x_i^2$$

edges in $\mathcal{P}$. Thus, s, the number of states in $\mathcal{P}$ is $O(l^2)$, where $l = \sum_1^n x_i$. But by the observation above, the computation time of whether X is a code is $O(l^4)$.

For this reason, the setting of M to f^4 (f is the number of states in $\mathcal{F}$) is more than generous.

Suppose the length of the longest word in X is m. Then the recursion level in the computation of whether X is θ-compliant or θ-free is bound by one or two times the length of the longest word in X, respectively. Thus computation time totals at most

$$\binom{\binom{m}{2}}{2} \text{or} \binom{\binom{2m}{2}}{2}, \text{ respectively.}$$

Note that the number of states in $\mathcal{P}_\theta$ is

$$\sum_{i=1}^{n} x_i y_i,$$

where y_i is the number of occurences in X of the symbol $\theta^{-1}(a_i)$.

In designing CodeGen, we observed that checking the remaining θ properties using the same algorithm demands a disproportionate amount of memory. The program makes these checks using standard string methods on appropriately constructed objects. The computation time turns out to be linear. We discuss the implementation next.

C. *Object-Oriented Implementation*

CodeGen was designed in Visual Basic 6.0, which offers a limited form of OOP. Its objects therefore have no inheritance. There are eight language-theoretic objects used: the classes Point, Pointlist, Alphabet, Word, WordList, Involution, Automaton, and Language.

1) The Point class is a wrapper for tuples, with .X and .Y as its primary members.
2) The PointList class wraps an array of Points, useful for storing paths in an Automaton and offers expected functions as .Add, .Remove, etc.
3) The Alphabet class treats a set of symbols like a bag, and is also able to display its members properly. It ignores attempts to .Add symbols it already contains.
4) The Word class wraps a string and makes it behave like a word.
5) The WordList class wraps an array of Words, and its .Add function returns `FALSE` when an attempt is made to add a word it already contains.
6) The Involution class contains the assignment pattern of an involution.
7) The Automaton class is where the TRACE algorithm takes place. This class knows if it represents a flower or square automaton and behaves like a finite graph. Its most significant data member is Paths, an array of PointLists which hold the edges traced in the automaton, as in the graph representing it. Its .MakeFlower and .MakeSquare functions behave as described in III-.1.
8) The Language class has a WordList and an Involution. It holds three Automatons as well, representing its flower automaton ($\mathcal{F}$), square automaton ($\mathcal{P}$), and the product of the involution of its flower with its flower ($\mathcal{P}_\theta$). It offers the methods .AddRandomWord, .AddWord, and .Test, which call the TRACE routine of the appropriate automata. The results are then available in public class members.

Through these objects CodeGen limits the number of string operations used in testing for $\theta(k, m_1, m_2)$-subword compliance and in testing for whether a code is θ-k. These are standard routines which construct Language objects and attempt to add the involution of words to them. Since the number of k-length subwords is a linear function of the size of a code, these tests are performed in linear time.

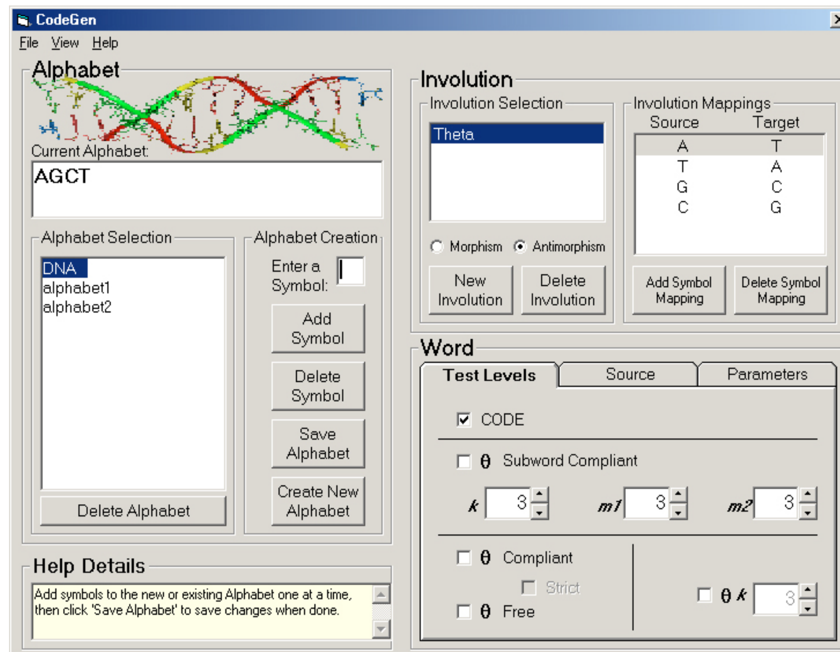

Fig. 6. Main Screen

V. The CodeGen Interface

The interface allows the user to obtain new sets of code words or expand an existing set of code words. All capabilities are presented on a main screen, consisting of three panels labeled **Alphabet**, **Involution**, and **Word** (Figure 6).

These may be accessed in any order, but an alphabet must be selected or created before an involution can be set, and an involution must be selected before testing or generating code words. After the language has been tested and verified, it can be saved.

A. Alphabet

The DNA alphabet $\{A, T, C, G\}$ is present and selected by default. Other alphabets can be created, saved or deleted and may contain up to 256 numeric or character symbols. The number of alphabets is currently limited to twelve.

B. Involution

The DNA (**Theta**) involution is present and selected by default, and cannot be altered. Up to three different involutions may be defined over each alphabet. **Morphism** may be selected for regular concatenation of words, or **Antimorphism** for Watson-Crick complementarity, as discussed in II-A.

C. Word

Language operations take place on three tabs in **Word** (although see V-D): **Test Level**, **Source**, and **Parameters**.

*1) **Test Level**:* (see bottom right of Figure 6) sets the desired properties of the language. **Code** is basic (item 1 in II-A). Checking **Subword Compliant** enables detection of the "hairpin" DNA hybridization (item 4 in II-A), avoiding complementary subwords of prefixes or suffixes of individual words. The disallowed subword (hybridization) length is chosen in **k**. The distances between the subword and its complement (the length of the "hairpin" loop) cannot be shorter than the value set in **m1** or longer than the value set in **m2** .

If the Θ **Compliant** box is checked, this enables verification of the θ-compliant property. If the Θ **Free** box is checked, this enables verification of the θ-free property (see items 2 and 3 in II-A.). Verification of the θ-k property (item 5 in II-A) – the strongest of the θ properties – is enabled if the **Θ-k** box is checked. This rules out the presence of complementary subwords (cross-hybridizing sequences) of the length set in **k**, if this is possible (see II-B).

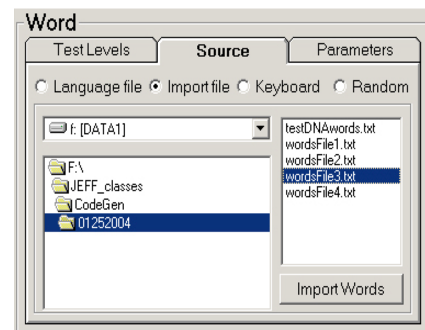

Fig. 7. Language Source Selection Tab

*2) **Source**:* (Figure V-C.1) is where the code words are actually created. It has four radio buttons.

- **Language file** This displays all of the previously saved language files in the current directory, and allows the user

to select one. These language files contain the relative alphabet and involution, plus the words and previously defined properties of the language. Once a language file is selected, all of the words are displayed and the user may add or delete words, or test the language.

- **Import file** The directory information is displayed to allow the user to locate and select a text file of words to be imported into the current language. Words in the text file should be one per line and are imported with only two restrictions: they must contain symbols from the current alphabet, and no words may be duplicated. After importing words, the new language is displayed in the **Language View** pop-up window and can be tested for the desired properties.
- **Keyboard** Words may be added directly from the keyboard. Limitations are that words must consist of symbols from the current alphabet and that words cannot be duplicated. The list of words currently in the language is displayed and can be modified or tested at any point.
- **Random** The program will generate random code words according to the chosen **Test Levels** and **Parameters** (see V-C.3). Words are reported as they are generated until the request is met or until the process exceeds program limitations (see IV-A). The code words and the properties of the generated language are then displayed in the **Language View** window.

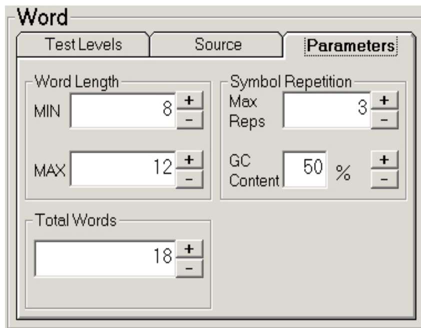

Fig. 8. Random Language Generation Parameters Selection Tab

*3) **Parameters**:* (see Figure 8) sets values for the generation of code words. **MIN**imum and **MAX**imum word lengths display and can be adjusted. **Max Reps** sets the maximum number of consecutive symbol repetitions, while **GC Content** sets the maximum percentage of total Cs and Gs in each word for DNA based languages. **Total Words** displays and sets the number of words in the language. All values change to reflect the properties of words currently in the language. A language meeting these parameters (and the **Test Levels**, see V-C.1) may then be generated using **Word→Source→Random→Generate**.

D. Language View window

From the menu bar **View→Show Language** displays all the words and properties of the current language in a separate window (see Figure 9). The language may then be tested

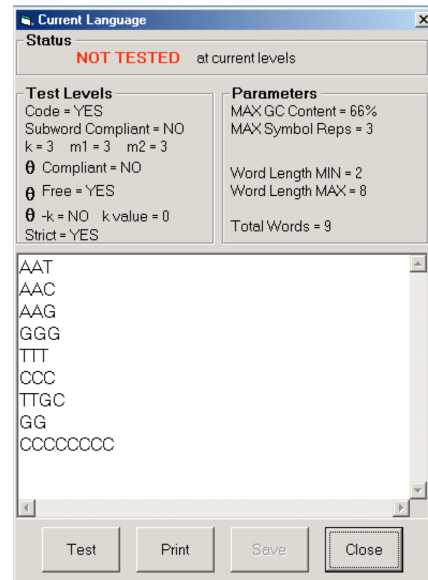

Fig. 9. Language Display Screen

at these levels using the **Test** button. The window updates to display the properties and values the language supports. The **Status** label then changes to read "VERIFIED at current levels". **Save** will then store the language – namely, code words, parameters, and verified levels.

Finally, the user may change desired test levels and retest the current language, delete any or all of the words, create a new language as spelled out in V-C.2, or set up new alphabets (see V-A) and involutions (see V-B).

VI. Limits and What is Yet to Be Done

CodeGen version 1 is an object-oriented program that has been developed in Microsoft Visual Basic 6.0. It currently runs only on Microsoft Windows platforms. The recursion in the main algorithm demands at least 128 MB of memory. It is an ongoing project. Version 2 should be a translation to Java for multi-platform use, and to take advantage of the more complete object-oriented features of that language.

At present, CodeGen cannot generate or operate with words or alphabets consisting of more than 256 symbols, although the language size is limited only by available memory.

Version 2 will deal with a more serious issue. CodeGen will at present generate a maximum of 4096 random code words before giving notice that the search is incomplete. The word-generation technique uses a typical pseudo-random number generator and may words that are already in the language, thus failing to produce the requested code words even if when request is combinatorially possible to satisfy.

This is addressed already in various theorems in [14], where several methods are outlined for using a relatively small number of shorter code words to generate infinite sets of words of any length satisfying particular requirements. As mentioned in II-A, for instance, if X is a code and all possible concatenations of two words from X is θ-k, then X^* is θ-k if the words in X are at least k symbols long. The next step

in the development of the algorithms we present here is to supplement pseudo-random generation with code generation techniques drawn from [14].

The languages and properties described here are theoretical but have been shown in the laboratory to give hybridization-free code words (see [19]). It is to be understood, though, that CODEGEN is not an attempt simulate to *in vitro* DNA behavior *in silico* as in the `Edna` project of Garzon [11], [10], but an attempt to model DNA's chemical properties algebraically.

This project is part of the wider effort to facilitate generation of code words and DNA sequences for various application purposes including biotechnology and computation encoding.

VII. ACKNOWLEDGMENTS

Development of these algorithms, their programmatic implementation, and the character of its user interface are in large part creditable to the continuous firm guidance and valuable advice of Natasha Jonoska. In addition, the authors are indebted to the anonymous reviewers of this paper for their many helpful suggestions.

This work has been supported by grant EIA-008615 from the National Science Foundation, USA.

REFERENCES

[1] M. Arita, S. Kobayashi, "DNA sequence design using templates", *New Generation Computing*, vol. 20, pp. 263-267, 2002.

[2] E. B. Baum, "DNA sequences useful for computation," unpublished article, available at: http://www.neci.nj.nec.com/homepages/eric/seq.ps, 1996.

[3] J. Berstel, D. Perrin, *Theory of Codes*, Orlando: Academic Press, 1985.

[4] J. H. Chen, N. R. Kallenbach, N. C. Seeman, "A specific quadrilateral synthesized from DNA branched junctions," *Journal of the Am. Chem. Soc.*, vol. 111, pp. 6402-6407, 1989.

[5] J. Chen, N. C. Seeman, "Synthesis from DNA of a molecule with the connectivity of a cube," *Nature* vol. 350, pp. 631-633, 1991.

[6] R. Deaton et al, "A DNA based implementation of an evolutionary search for good encodings for DNA computation," *Proceedings of the IEEE Conference on Evolutionary Computation*, ICEC-97, pp. 267-271, 1997.

[7] D. Faulhammer, A. R. Cukras, R. J. Lipton, L. F. Landweber, "Molecular Computation: RNA solutions to chess problems," *Proceedings of the National Academy of Sciences of the United States*, vol. 97 (4), pp. 1385-1389, 2000.

[8] U. Feldkamp, S. Saghafi, W. Banzhaf, H. Rauhe, "DNASequenceGenerator - A Program for the construction of DNA sequences," *Proceedings of the 7th International Meeting on DNA Based Computers*, N. Jonoska, N. C. Seeman, eds., *LNCS*, vol. 2340, pp. 179-188, 2001.

[9] U. Feldkamp, H. Rauhe, W. Banzhaf, "Software Tools for DNA sequence design," *Genetic Programming and Evolvable Machines*, vol. 4 pp. 153-171, 2003.

[10] M. Garzon, D. Blain, K. Bobba, A. Neel, M. West, "Self-assembly of DNA-like structures In Silico," *Genetic Programming and Evolvable Machines*, vol. 4, pp. 185-200, 2003.

[11] M. Garzon, R. Deaton, D. Reanult, "Virtual test tubes: a new methodology for computing," *Proceedings of the 7th International Symposium on String Processing and Information Retrieval*, *IEEE Computer Society Press*, pp. 116-121, 2000.

[12] M. Garzon, K. Bobba, B. Hyde, "Digital information encoding on DNA," *Aspects of Molecular Computing*, N. Jonoska, G. Paun, G. Rozenberg, eds., *LNCS* vol. 2950, pp. 153-166, 2004.

[13] N. Jonoska, D. Kephart, K. Mahalingham, "Generating DNA code words," *Congressus Numerantium*, vol. 156, pp. 99-110, 2002.

[14] N. Jonoska, K. Mahalingham, "Languages of DNA based code words," *Preliminary Proceedings of the 9th International Meeting on DNA Based Computers*, J. Chien, J. Reif, eds., pp. 58-68, 2003.

[15] L. Kari, S. Konstantinidis, E. Losseva, G. Wozniak, "Sticky free and overhang free DNA languages," *Acta Informatica*, vol. 40 (2), pp. 119-157, 2003.

[16] Z. Li, "Construct DNA codewords using backtrack algorithm," preprint.

[17] S. Hussini, L. Kari, S. Konstantinidis, "Coding properties of DNA languages," *Proceedings of the 7th International Meeting on DNA Based Computers*, N. Jonoska, N. C. Seeman, eds., *LNCS*, vol. 2340, pp. 107-118, 2001.

[18] Q. Liu et al, "DNA computing on surfaces," *Nature*, vol. 403, pp. 175-179, 2000.

[19] K. Mahalingam, "Involution Codes: with application to DNA coded languages," preprint.

[20] A. Marathe, A. E. Condon, R. M. Corn, "On combinatorial DNA word design," *Proceedings of the 5th International Meeting on DNA based computers*, 1999. Also appears in: *Journal of Computational Biology*, vol.8 (3), pp. 201-219, 2001.

[21] A. J. Ruben, S. J. Freeland, L. F. Landweber, "PUNCH: An evolutionary algorithm for optimizing bit set selection," *Proceedings of the 7th InternationalMeeting on DNA Based Computers*, N. Jonoska, N. C. Seeman, eds., *LNCS*, vol. 2340, pp. 260-270, 2001.

[22] N. C. Seeman, "De Novo design of sequences for nucleic acid structural engineering," *Journal of Biomolecular Structure & Dynamics*, vol. 8 (3), pp. 573-581, 1990.

[23] E. Winfree, F. Liu, L. A. Wenzler, N. C. Seeman, "Design and self-assembly of two dimensional DNA crystals," *Nature*, vol. 394, pp. 539-544, 1998.