A DNA Sequence Compiler

Udo Feldkamp[*], Wolfgang Banzhaf[*], Hilmar Rauhe[*†]

February 11, 2000

feldkamp@LS11.cs.uni-dortmund.de
banzhaf@LS11.cs.uni-dortmund.de
rauhe@LS11.cs.uni-dortmund.de

1 Abstract

Various approaches to the self-assembly of molecules have been introduced already[1), 2), 3), 4), 5), 6), 7)]. A step further toward flexible design and construction of precisely defined molecules are approaches to *programmable* self-assembly[5), 10)]. In order to allow arbitrary programming, a sufficient solution of the negative design problem[8), 5), 9)] is needed. We present a computer program which translates formal grammars directly into DNA molecules. It allows the construction of DNA molecules with defined logical structure and physical properties. Applications of the compiler are DNA-computing algorithms, nano-frameworks and the construction of biochips.

2 Introduction

The correspondence of Chomsky-classes of grammars to certain implementations of self-assembling DNA molecules was shown by Winfree et al. [11)]. In particular it was shown that linear molecules are capable to represent regular grammars, branched sequences ("dendrimers") to represent context-free grammars and a certain type of tile-shaped molecules (DX molecules) to represent universal grammars[11), 5)]. The molecules in such a system represent the rules of a grammar. The true computational step is the hybridization and ligation of self-assembling molecules to larger molecules thereby executing the grammar's rules.
In order to allow arbitrary programming the "negative design problem"[8), 9), 5)] has to be solved in a sufficient way. For that purpose we constructed a compiler that directly translates grammars into DNA molecules. At the moment the compiler only translates regular grammars into linear molecules but is not necessarily restricted to it. The compiler was originally developed as a programming tool for an approach to programmable digital DNA[10)]. The linear self-assembling molecules generated by the compiler can be synthesized *in vitro* by an oligonucleotide synthesizer. Those linear molecules can be used not only for the implementation of regular grammars but also for the design of biochips and DNA based cryptography[12)].
The self-assembling molecules ("rule-molecules" or so-called algomers) that the compiler generates consist of a double stranded core sequence and two sticky ends[10)] (Figure 1).
Each algomer represents a rule of a grammar, the core sequence encoding a terminal and the sticky ends encoding variables. As regular expressions are built by replacing the right-end variable of an expression with the right-hand terminal-variable-pair of a rule in which this variable is on its left side, hybridizing complementary sticky ends (which represent the left-hand and right-hand appearance of the same variable) assembles the algomers to longer sequences that represent words of the chosen grammar (called logomers).

[*] Dept. of Computer Science, LS11, University of Dortmund, 44221 Dortmund, Germany
[†] To whom correspondence should be addressed

3 Specifications

The rules of a regular grammar have the form A → xB or A → x, where A and B are variables and x is a terminal. As variables and terminals have fixed positions in a rule, only the rules set and the start variable are required inputs to define the grammar; the variable and terminal sets can be extracted from the rules. The algomers are sequences with a double stranded core sequence representing the terminal of a certain rule and two sticky ends (one on each strand) representing the variables of this rule (see Figure 1). In addition to their logical structure, algomers can be defined to have certain physical, chemical and biological properties. This is done to meet design criteria as well as to favor proper thermodynamical behavior and to avoid possible errors.

```
Rules                       Algomers
                                                        _
               HindIII           s0                     A
S:=s0A    5′      agctt caacacatggagttacacgc                 3′
          3′          a gttgtgtacctcaatgtgcg  gcctttgtag  5′
                                                        _
               HindIII           s1                     A
S:=s1A    5′      agctt gaaaaaattggactcggggc                 3′
          3′          a ctttttttaacctgagccccg  gcctttgtag  5′
                                                        _
               HindIII           s2                     A
S:=s2A    5′      agctt gctcctagaagtctacaagc                 3′
          3′          a cgaggatcttcagatgttcg  gcctttgtag  5′
                                                        _
               HindIII           s3                     A
S:=s3A    5′      agctt cttctgccatacaactaggc                 3′
          3′          a gaagacggtatgttgatccg  gcctttgtag  5′


                                                      _____
                  I              e                    BamHI
I->e      5′  gtcttgtgtc cttgtttaatacaggggcgc  g           3′
          3′             gaacaaattatgtccccgcg  cctag       5′
```

```
Rules                       Algomers
                                                      _
                A               0                     B
A->0A   5′  cggaaacatc ggatttggcaacaacctgag                3′
        3′             cctaaaccgttgttggactc  gaaaatcggg  5′
                                                      _
                A               1                     B
A->1A   5′  cggaaacatc caaccaggattaagccatgc                3′
        3′             gttggtcctaattcggtacg  gaaaatcggg  5′
                                                      _
                B               0                     C
B->0C   5′  cttttagccc ggatttggcaacaacctgag                3′
        3′             cctaaaccgttgttggactc  cctctaatgg  5′
                                                      _
                B               1                     C
B->1C   5′  cttttagccc caaccaggattaagccatgc                3′
        3′             gttggtcctaattcggtacg  cctctaatgg  5′
                                                      _
                C               0                     D
C->0D   5′  ggagattacc ggatttggcaacaacctgag                3′
        3′             cctaaaccgttgttggactc  ggcgtttatc  5′
                                                      _
                C               1                     D
C->1D   5′  ggagattacc caaccaggattaagccatgc                3′
        3′             gttggtcctaattcggtacg  ggcgtttat   5′
                                                      _
                D               0                     E
D->0E   5′  ccgcaaatag ggatttggcaacaacctgag                3′
        3′             cctaaaccgttgttggactc  gtctcgtatg  5′
                                                      _
                D               1                     E
D->1E   5′  ccgcaaatag caaccaggattaagccatgc                3′
        3′             gttggtcctaattcggtacg  gtctcgtatg  5′
                                                      _
                E               0                     F
E->0F   5′  cagagcatac ggatttggcaacaacctgag                3′
        3′             cctaaaccgttgttggactc  gcatcttgac  5′
                                                      _
                E               1                     F
E->1F   5′  cagagcatac caaccaggattaagccatgc                3′
        3′             gttggtcctaattcggtacg  gcatcttgac  5′
                                                      _
                F               0                     G
F->0G   5′  cgtagaactg ggatttggcaacaacctgag                3′
        3′             cctaaaccgttgttggactc  ctgccaatag  5′
                                                      _
                F               1                     G
F->1G   5′  cgtagaactg caaccaggattaagccatgc                3′
        3′             gttggtcctaattcggtacg  ctgccaatag  5′
                                                      _
                G               0                     H
G->0H   5′  gacggttatc ggatttggcaacaacctgag                3′
        3′             cctaaaccgttgttggactc  gacttcactg  5′
                                                      _
                G               1                     H
G->1H   5′  gacggttatc caaccaggattaagccatgc                3′
        3′             gttggtcctaattcggtacg  gacttcactg  5′
                                                      _
                H               0                     I
H->0I   5′  ctgaagtgac ggatttggcaacaacctgag                3′
        3′             cctaaaccgttgttggactc  cagaacacag  5′
                                                      _
                H               1                     I
H->1I   5′  ctgaagtgac caaccaggattaagccatgc                3′
        3′             gttggtcctaattcggtacg  cagaacacag  5′
```

Figure 1: Implementation of a grammar of 32-bit datatypes. Left: the four start algomers identifying the four bytes and one end algomer. Right: the algomers encoding the eight bits of a byte.

There are two specific algomers, called the start and end terminator respectively, appearing at the start and end of a logomer. "Start" acts as priming site of the forward primer during readout and can be used as a position mark to distinguish logomers with otherwise identical structure. Using four different start terminators for example allows construction of a 32-bit data type out of four 1-byte logomers where each byte's position is marked. Also, each byte can be read out separately. The outer sticky ends of the start and end terminator are not used as variables but as cloning sites to allow the logomers to be pasted into a cloning vector[10].

The most important requirement for a successful and error-free application is the avoidance of non-specific hybridizations. These can occur between the sticky ends of algomers, during PCR and in further applications of the logomers. Therefore, the sequences representing the terminals and the variables should be as unique as possible.

For the purpose of constructing unique sequences a graph-algorithm was developed by Niehaus[14] to produce a pool of n_b-unique sequences. A pool of sequences is said to be *n_b-unique* if all sequences of the pool have common substrings of at most length $n_b - 1$, i.e. any subsequence in the pool of length n_b is unique. *Uniqueness* on the other hand is defined as $1 - (n_b - 1)/n_s$, a ratio to measure how much of a sequence of length n_s is unique. For example 20-mers that are 10-unique have common subsequences of at most 9 subsequent nucleotides and a uniqueness of 55%.

In order to construct unique sequences, a DNA sequence is considered to consist not of single nucleotides but of overlapping subsequences of a defined length n_b (so called base strands) (Figure 2). Base strands are organized as nodes of a directed graph where every base strand has exactly 4 predecessors and 4 successors. Every successor contains its predecessing base strand minus the first nucleotide and plus one nucleotide out of the alphabet of DNA nucleotides (a, c, g, t) at the 3'-end. Sequences are generated as paths through the graph. During generation of the sequences a base strand may be used only once, so that the paths of any two sequences do not have a node in common. Likewise if a node is used, its complement may not be used in any other sequence while self-complementary base strands may not be used at all (Figure 3). Thus the algorithm guarantees all generated sequences to have a maximum common subsequence of at most $n_b - 1$ nucleotides and so are n_b-unique.

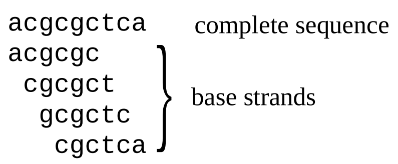


Figure 2: A sequence of length n_s consisting of $(n_s - n_b + 1)$ overlapping base strands of length n_b.

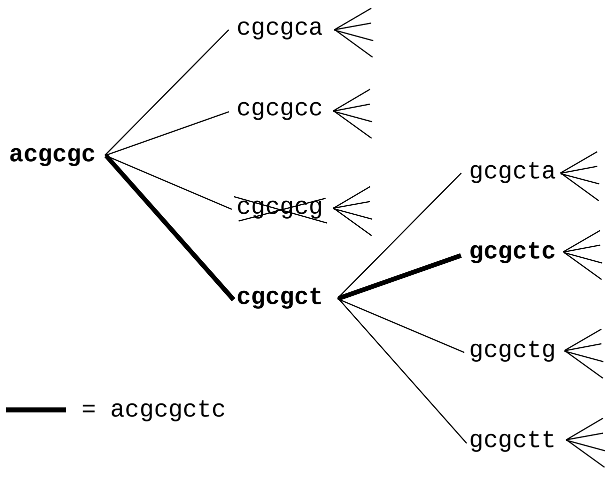


Figure 3: Graph of base strands. A path of m nodes represents a sequence of length n_b + m– 1. The node `cgcgcg` is self-complementary and therefore marked as forbidden.

This algorithm is still not sufficient to construct sequences that can be used for DNA computing and the production of biochips. First, in order to construct molecules that are working properly *in vitro,* certain physical, chemical and biological requirements have to be applied to the generated sequences. Second, in

order to construct new sequences that are compatible (in terms of being n_b-unique) to an existing pool of old sequences, the compiler must be able to process the old sequences to their base strands and to subtract these from the pool of all available base strands. Third, in order to create unique variables (*sticky ends*) that can attach to a multitude of terminals, the algorithm must be applied in parallel. Fourth, depending on design, additional measures should be applicable to sustain uniqueness.
There are several requirements to the physical properties of the constructed sequences that are essential for a successful *in vitro* approach, such as a uniform melting temperature (to prevent mismatches and biases in the probability distribution of the ligation process[15)]), GC-base-pairs at the ends of the sequences to avoid fraying (to prevent in vitro errors and deviations from the two-state-transition model used for thermodynamic estimations), no occurance of three or more consecutive guanine nucleotides (to prevent several undesirable phenomena[16)]), different lengths for terminal and variable sequences (to uphold stability of the algomers while hybridizing to logomers) and bounds for the GC-ratio of the used base strands (to approximate the two-state-transition model).
A drawback of the chosen understanding of uniqueness regarding common subsequences is the possible similarity of sequences said to be n_b-unique for a not small enough value of n_b. E.g., the subsequences ...`aaaagaaaa`... and ...`aaaacaaaa`... are very similar and could both anneal to ...`ttttctttt`..., but are 5-unique. Therefore the maximum number of identical bases for all possible shifts of two compared sequences and their complements (maximal homology) can be used as an additional measure of similarity.
Since the whole *in vitro* approach is based on the *controlled* interaction between DNA and enzymes, the uniqueness of the constructed sequences can itself be seen as their major chemical property. Besides, a whole set of enzymatic interactions such as interaction with restriction endonuclease, ligase, polymerase, methylase etc. is essential. Therefore, the sequences should contain (or sometimes definitively not contain) certain enzymatically active sites.
Biological properties are essential under several aspects. First, biological techniques such as hybridization, ligation, PCR and cloning are used *in vitro* for implementation. Cloning for example requires inclusion of appropriate cloning sites and design with respect to the cloning vector. Second, if the constructed molecules are used *in vivo* or under otherwise sensitive conditions, the presence or absence of specific biological meaningful sequences such as start-, stop-, promotor- or specific recombinational sites might be required. For example the compiler can be instructed to avoid sequences that contain start codons to prevent accidential translation even though it is highly unlikely to produce biologically meaningful information by coincidence.
In order to reuse existing molecules or to generate sequences that are compatible to existing biological sequences, the compiler is able to process predefined sequences to their base strands and subtracts these from the pool of all available base strands.

4 Implementation

The compiler contains a core module (called sequence generator) which uses the base strands of a given length to construct sequences regarding the requirements listed above. On different stages of the construction process different requirements are implemented. All requirements can be applied in the form of filters either to the pool of base strands or to the pool of sequence candidates (see Figure 4), except uniqueness and the prevention of fraying which are implemented within the construction.
The generation of sequences consists of the following steps:
1. Create a directed graph out of the base strands of a certain length and apply all defined filters to the base strands by marking those as forbidden that do not satisfy the criteria. Among these filters there are logical and structural filters (i.e. self-complementary sequences), physical filters (i.e. GC-ratio, melting temperature, triple G sequences), chemical filters (i.e. enzymatic sites) and biological filters (i.e. start-, stop-, promotor-, recombinational sequences). Also base strands of already existing sequences can be excluded at this stage from being used again.

2. As long as there are still sequences to generate and there are possible start nodes (not forbidden, not yet used and not yet tried as start nodes), choose one of them randomly. Here, fraying of the 5'-end can be prevented by limiting the choice such that the first nucleotide of the start node is a G or C. With each chosen start node do the following:
2a. As long as the path has not the desired length, choose a successor node which is not forbidden nor used randomly and mark it and its complement as used. If the last successor of the path is to be chosen, fraying of the 3'-end can be prevented in this step by limiting the choice accordingly.
If a node has no usable successor left before the path has reached its full length, backtracking is used to search another path, thereby marking the no longer used nodes as unused.
2b. If a path has reached its full length, the newly generated sequence is filtered by applying certain logical, physical, chemical or biological criteria, such as melting temperature, GC-ratio or homology. If a sequence does not meet the criteria backtracking is initiated, otherwise it is added to the output pool.
2c. If backtracking leads the algorithm back to the start node and none of its four successors is available, this search is terminated and another start node is chosen by the outer loop.

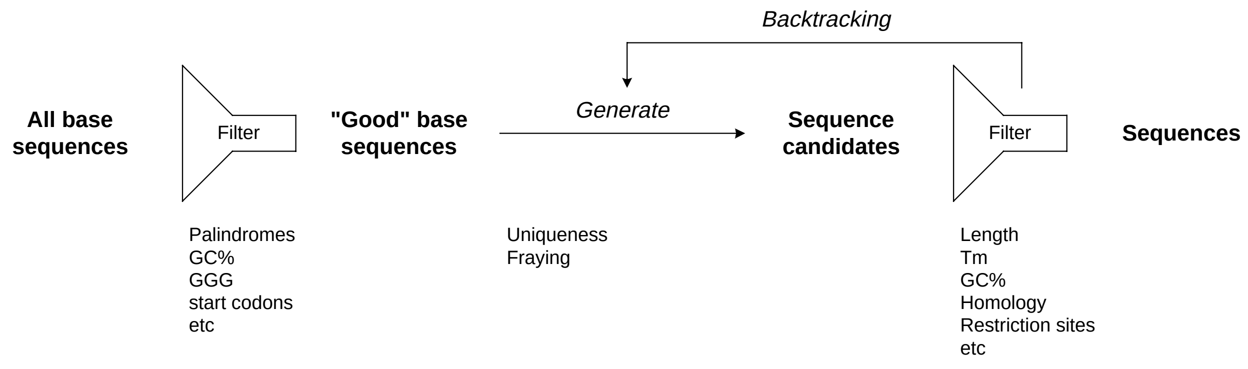


Figure 4: Sketch of the DNA sequence generator.

The compiler first parses the given ruleset of the grammar. It then uses the generator module to generate the terminal and variable sequences used in the algomers.
Though the compiler can construct all sequences completely *de novo*, the construction of *sticky ends* is particularly needed for construction of self-assemblying molecules (nano-frameworks). Because variables (sticky ends) have to be attached to a multitude of terminals that can be represented by complex molecules, the original Niehaus-graph algorithm[14)] as described above is not sufficient to grant uniqueness. Instead, the compiler uses a "parallel extension" uniqueness algorithm. This algorithm makes use of several path constructions executed in parallel and thus is able to meet several additional requirements:
If sequences generated as a n_b-unique sequence pool were concatenated to algomers, and later to logomers, "new" base strands would emerge around the junctions, which hadn't been considered in the construction process (Figure 5). Thus, in worst case, the length of a repeatedly appearing subsequence could increase to $2 * (n_b – 1)$ ($n_b – 1$ nucleotides in each direction from the junction).
Any variable sequence can (and usually will) have more than one adjacent terminal sequence and vice versa, so several junctions must be regarded simultaneously when generating one terminal or variable sequence.
If one terminal sequence has more than four adjacent variables (or vice versa), uniqueness even has to be violated for a successful compilation because the terminal sequence path cannot branch into more than four junctions (see Figure 6). If a terminal sequence is adjacent to n variable sequences at least for the next $\lfloor \log_4(n) \rfloor$ positions base strands must be used more than once. These violations must be tolerated in a controlled way so that they are still prevented if not necessary.

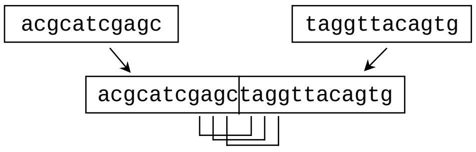


Figure 5: Concatenation of sequences leads to the occurrence of base strands not regarded in the construction of the sequences (here for $n_b = 4$).

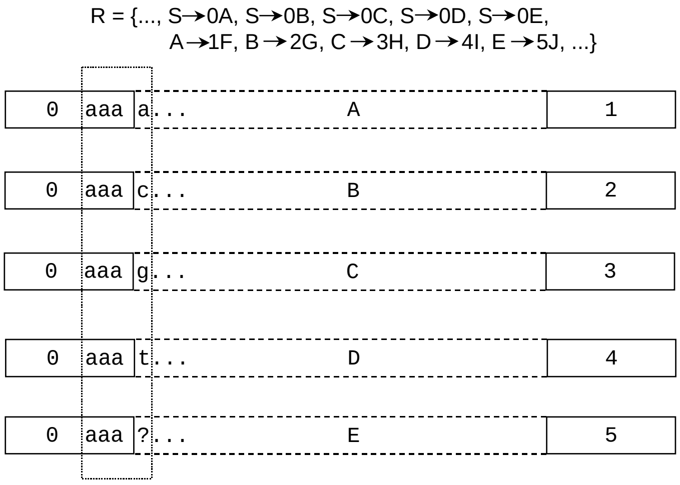


Figure 6: Tolerated uniqueness violation. The compiler tolerates uniqueness violation because the terminal path for 0 must branch into more than four variable paths. The compiler also limits the multiple use of the base strand used in the last path to this position (dotted box).

The compiling process can be divided into the following steps:
1. Read one type of sequences (in general the terminal sequences) from existing molecules if present or otherwise generate them with the algorithm described above.
2. For each variable collect all pairs of terminals whose sequences will embrace the respective variable sequence in a logomer from the rules set and bundle them into groups, one group for each variable. Align their paths such that between each pair a gap of the length of the variable path plus 2 * (n_b – 1) remains for the junctions and the variable sequences themselves (Figure 7).
3. Choose the last base strand of each left hand terminal sequence as the start node of the according path.
4. Build the terminal-variable junctions by applying the path search loop for all paths in parallel, regarding the joining of paths and the possibly necessary uniqueness violation.
5. Build the variable sequences as common path extensions of the corresponding terminal paths in parallel.
6. Build the variable-terminal junctions in parallel, regarding the branching of paths and the possibly necessary uniqueness violations again.
Annotations to these steps:
1. It is also possible to reuse terminal sequences and create additional sequences to expand a grammar. For this purpose the base strands are extracted from the reused sequences and are excluded from the construction pool.
2. Even “empty” terminal sequences can be aligned if variables shall be left open.
3. If the terminal sequence should be empty (or for any other reason be shorter than the base strand length used for the variables) the “missing” nucleotides are chosen randomly.
4. Searching “in parallel” means first searching the immediate successor node for all paths (in all groups), then searching all second successors etc. Joining is implemented by choosing the same successor for all paths in one group respectively. The prevention of fraying is realized in this step by limiting the choice of the first successors. A necessary violation, i.e. the multiple usage of one base strand, is allowed only within one step of the parallel growth so that these multiple occurences are limited to the same position in all paths. One of the compiler’s parameters defines in how many positions from the junction

such violations are to be tolerated.
5. Again, all paths within one group have the same successors so that the variable path is a common extension of those paths.
6. Here the $n_b - 1$ successor nodes for each path are not chosen randomly but predetermined by the first $n_b - 1$ nucleotides of the right hand terminal sequence. Otherwise the usual path searching routine is used. Again, uniqueness violations are limited to occur within a "column" (see description to step 4).
If backtracking is needed because a path is blocked or a completed sequence does not meet the filter criteria not all paths track back as this could lead to infinite loops of two or more paths initiating backtracking alternately. Therefore, backtracking is used only for the "guilty" path to search another path. The other paths are "frozen" in the position where the backtracking was initiated until the "guilty" one reaches this position again. Then the parallel search continues.

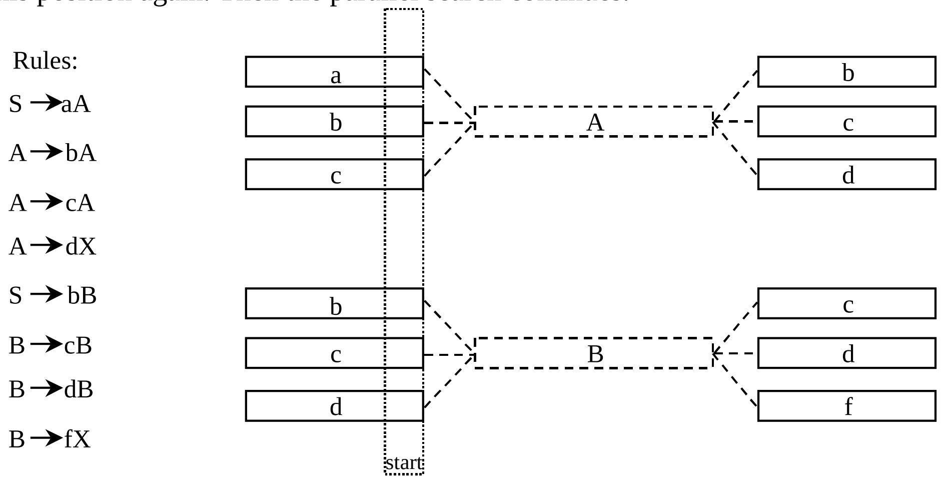


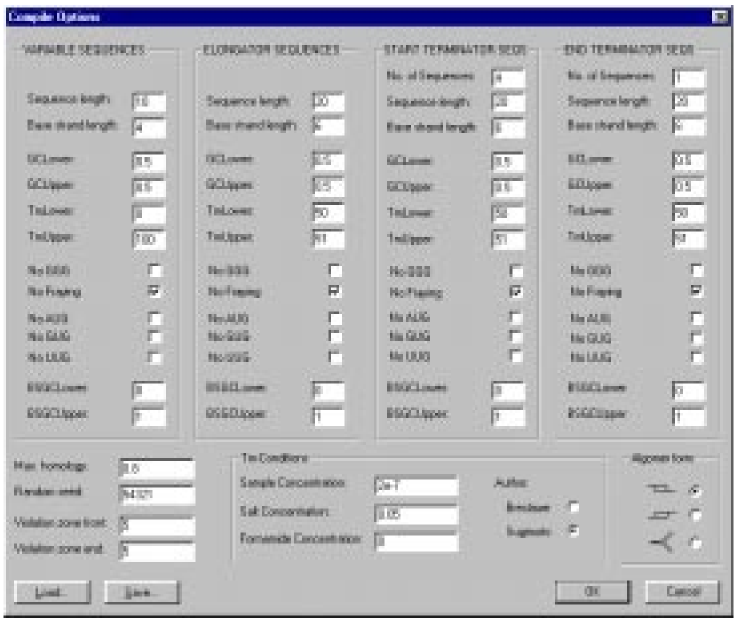
Figure 7: Parallel extension. The sequences representing A and B are generated by extending the paths of the terminal sequences in parallel. The dotted column contains the start nodes for the extension algorithm.

Note that the parallel lengthening of all variable paths allows regarding not only all joining terminal-variable junctions and branching variable-terminal junctions, but also vice versa, controlling the according uniqueness violations.
If some variable sequences are to be reused and others are to be generated, the already existing sequences are also included in the parallel extension algorithm and thus can be taken into account with respect to controlled uniqueness violations. Their completely predetermined paths are simply reproduced in the path search routine.

5 Results

In the context of our work on digital DNA[10)] we used the compiler to construct a library of DNA molecules for representation of 32-bit datastructures using the grammar $G_{32b2} = (\Sigma, V, R, S)$, with $\Sigma := \{0, 1, e, s_0, s_1, s_2, s_3\}$, $V := \{S, A, B, C, D, E, F, G, H, I\}$, start-symbol S and R:= $\{S \rightarrow s_i A, A \rightarrow 0B, A \rightarrow 1B, B \rightarrow 0C, B \rightarrow 1C, C \rightarrow 0D, C \rightarrow 1D, D \rightarrow 0E, D \rightarrow 1E, E \rightarrow 0F, E \rightarrow 1F, F \rightarrow 0G, F \rightarrow 1G, G \rightarrow 0H, G \rightarrow 1H, H \rightarrow 0I, H \rightarrow 1I, I \rightarrow e \text{ with } i = \{0, 1, 2, 3\}\}$.
The grammar describes the production of random bytes where every byte carries information about the position within a 32-bit datastructure. It produces logomers of the form $s_0\{x\}_8e$, $s_1\{x\}_8e$, $s_2\{x\}_8e$, $s_3\{x\}_8e$, where $\{x\}_8$ is a random concatenation of 8 bits and s_i is the byte position (see Figure 1).
All generated molecules are guaranteed to have a maximum overlap of 5bp (to be 6-unique), have 50%GC, a melting temperature between 50°C and 51°C (for the terminal sequences), a maximal homology of 0.8 and no fraying (see Figure 1 and Figure 8). The grammar was translated in 160 seconds

on a PC with a 300MHz Pentium and 128 MB RAM.


Compile Options

	VARIABLE SEQUENCES	ELONGATOR SEQUENCES	START TERMINATOR SEQS	END TERMINATOR SEQS
No. of Sequences:			4	1
Sequence length:	12	20	20	20
Base strand length:	4	5	5	5
GCLower:	0.5	0.5	0.5	0.5
GCUpper:	0.5	0.5	0.5	0.5
TmLower:	0	50	50	50
TmUpper:	100	51	51	51
No GGG	☐	☐	☐	☐
No Fraying	☑	☑	☑	☑
No AUG	☐	☐	☐	☐
No GUG	☐	☐	☐	☐
No UUG	☐	☐	☐	☐
BSGCLower:	0	0	0	0
BSGCUpper:	1	1	1	1

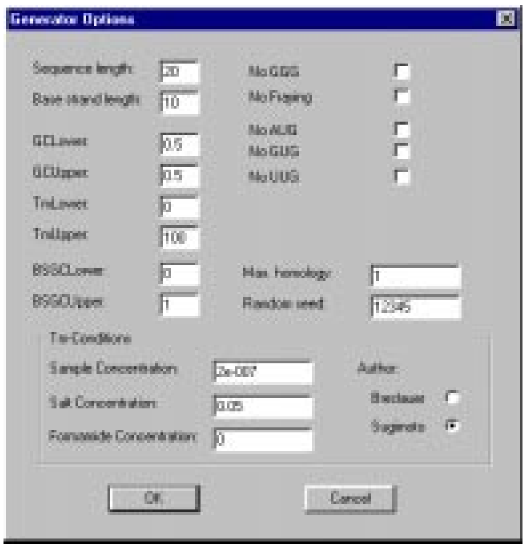
Max. homology: 0.6
Random seed: 54321
Violation zone front: 5
Violation zone end: 5

Tm Conditions
Sample Concentration: 2e-7
Salt Concentration: 0.05
Formamide Concentration: 0
Author: Breslauer ○ Sugimoto [✓ ●]

Algorithm form: [✓ ●] ○ ○

Figure 8: Screenshot of the compiler options used for translating the 32-bit grammar. The melting temperature (T_m) for the variables is not explicitely bounded, because for such short sequences the melting temperature is estimated as depending on the GC-ratio only, while for longer sequences the nearest-neighbor method is used.

The compiler was tested for *de novo* construction of oligonucleotide sets that can be used for biochips. A set of DNA sequences fixed to a chip has to meet two diverging needs: The sequences should be as unique as possible while having the most uniform possible melting temperatures.
The compiler could generate large sets of oligonucleotides with the defined properties (Figure 9) within the limits set by the requirements (see Figure 10). Because computational time increases exponentially with the base strand length, oligo sets with a uniqueness not lower than 50% are realistic. In case of 20-mers with 50% GC-ratio and 50% uniqueness this means around 140,000 out of 167,960 possible sequences. This is sufficient for todays biochips that contain up to 10,000 sequences on a single chip (Affymetrix, 25µm technique).


Generator Options

Sequence length: 20
Base strand length: 10
GCLower: 0.5
GCUpper: 0.5
TmLower: 0
TmUpper: 100
BSGCLower: 0
BSGCUpper: 1

No GGG ☐
No Fraying ☐
No AUG ☐
No GUG ☐
No UUG ☐

Max. homology: 1
Random seed: 12345

Tm-Conditions
Sample Concentration: 2e-007
Salt Concentration: 0.05
Formamide Concentration: 0
Author: Breslauer ○ Sugimoto [✓ ●]

Figure 9: Compiler options for the generation of sequence sets for biochips.

When restricted only to uniqueness, the compiler could find up to 86% of all possible sequences. Additional requirements to the oligonucleotides lowered the yield and increased compilation time. While strict parameters can lower the yield dramatically, GC-ratios around 50% typically had only minor impact to the yield of sequences with 50% - 55% uniqueness.

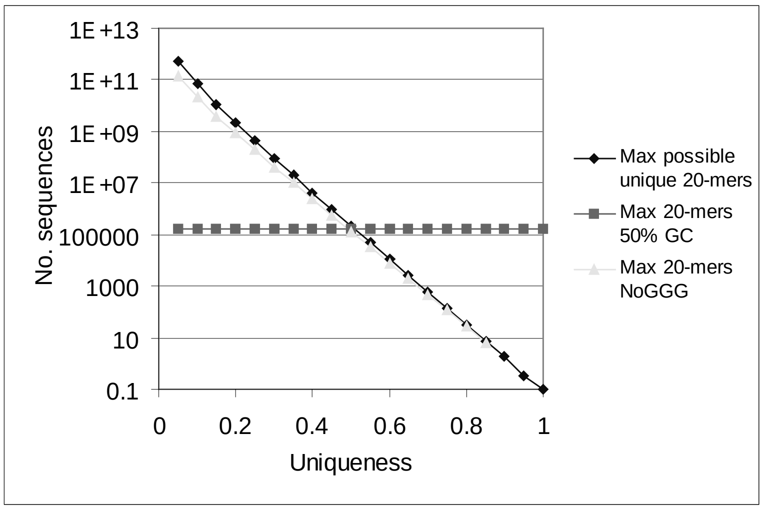


Figure 10: Certain requirements set upper limits to the number of sequences that can be generated by the compiler. The maximum number of 20-mers with 50%GC is $2^{20} \cdot \binom{n_s}{n_b}$= 167,960. The maximum number of 20-mers with 50% uniqueness is $\frac{4^{n_b}}{2\cdot(n_s - n_b + 1)}$ = 209,715. For 20-mers with 50% uniqueness a yield of 71% (148500) is expexted. For 20-mers with 55% uniqueness a yield of 73.3% (34,931 sequences) could be achieved. More restrictive is the exclusion of base strands with multiple-G subsequences. This leads to a loss in yield of 25% to over 40% (relative to the maximal possible unique sequences) around 50% uniqueness.

6 Discussion

The compiler shown here translates formal grammars directly into self-assembling DNA "rule molecules" (algomers). It can generate the molecules of a given ruleset fully *de novo* or can complete a set of molecules by constructing new molecules which are compatible (in terms of uniqueness and homology) to the already existing molecules. The completion of molecule sets is especially useful when reusing DNA molecules already available *in vitro*. In particular, the compiler can generate *sticky ends* (the variables) to predefined terminals. Therefore the compiler is not limited to the translation of regular grammars to linear molecules which is implemented yet, translation of higher grammars to more complex molecules such as DX-molecules[5)] and Ψ-molecules[7)] is currently under development. The reusability of still existing sequences enables the compiler to include biological sequences into the construction process. Thus, for example the construction of artificial chromosomes such as YACs (Yeast artificial chromosomes) and BACs (Bacterial artificial chromosomes) can be done with the compiler.
Another application of the compiler is the creation of sets of oligonucleotides for the construction of biochips. For that purpose the compiler can generate molecule sets with defined uniqueness and thermodynamical behavior *de novo* or assemble biological DNA sequences such as genomic DNA into

oligonucleotides.
The compiler can directly be connected to an oligonucleotide synthesizer to create an interface between silicon and DNA. This follows the philosophy of DNA/Silicon hybrid computing[10] where components of both worlds are integrated more closely. The compiler might evolve to or become part of the operating system of such an hybrid system.

The compiler is available for download at:
http://ls11-www.cs.uni-dortmund.de/molcomp/DNACompiler/

7 References

1) J. Chen, N.C. Seeman. Synthesis from DNA of a molecule with the connectivity of a cube. *Nature*, **350**, 631-633, (1991)
2) George M. Whitesides, John P. Mathias, Christopher T. Seto, Molecular Self-Assembly and Nanochemistry: A Chemical Strategy for the Synthesis of Nanostructures. *Science*, **254**, 1312-1319, (1991)
3) Christof M. Niemeyer, Takeshi Sano, Cassandra L. Smith, Charles R. Cantor, Oligonucleotide-directed self-assembly of proteins: semisynthetic DNA – streptavidin hybrid molecules as connectors for the generation of macroscopic arrays and the construction of supramolecular bioconjugates. *Nucleic Acids Research*, **22**(25), 5530-5539, (1994)
4) Chad A. Mirkin, Robert L. Letsinger, Robert C. Mucic, James J. Storhoff, A DNA-based method for rationally assembling nanoparticles into macroscopic materials. *Nature*, **382**, 607-609, (1996)
5) Erik Winfree, Furong Liu, Lisa A. Wenzler, Nadrian C. Seeman, Design and self-assembly of two-dimensional DNA crystals. *Nature*, **394**, 539-544, (1998)
6) Eugene R. Zubarev, Martin U. Pralle, Leiming Li, Samuel I. Stupp, Conversion of Supramolecular Clusters to Macromolecular Objects. *Science*, **283**, 523-526, (1999)
7) Matthias Scheffler, Axel Dorenbeck, Stefan Jordan, Michael Wüstefeld, Günter von Kiedrowski, Self-Assembly of Trisoligonucleotidyls: The Case for Nano-Acetylene and Nano-Cyclobutadiene. *Angewandte Chemie Int. Ed.*, **38**(22), 3312-3315, (1999)
8) N.C. Seeman: De Novo Design of Sequences for Nucleic Acid Structural Engineering. *Journal of Biomolecular Structure & Dynamics*, **8**(3), 573-581, (1990)
9) Anthony G. Frutos, Qinghua Liu, Andrew J. Thiel, Anne Marie W. Sanner, Anne E. Condon, Lloyd M. Smith, Robert M. Corn, Demonstration of a word design strategy for DNA computing on surfaces. *Nucleic Acids Research*, **25**(23), 4748-4757 (1997)
10) Hilmar Rauhe, Gaby Vopper, Udo Feldkamp, Wolfgang Banzhaf, Jonathan C. Howard, Digital DNA molecules. (also submitted, DNA6).
11) Erik Winfree, Xiaoping Yang, Nadrian C. Seeman, Universal Computation via Self-assembly of DNA: Some Theory and Experiments, *Proceedings of the 2nd DIMACS Meeting on DNA Based Computers, Princeton University, June 20-12*, (1996)
12) André Leier, Christoph Richter, Wolfgang Banzhaf, Hilmar Rauhe, Cryptography with DNA binary strands. (*Biosystems*, submitted)
13) D.T. Burke, G.F. Carle, M.V. Olson, Cloning of large segments of exogenous DNA into yeast by means of artificial chromosome vectors. *Science* **236**(4803), 806-812, (1987)
14) Jens Niehaus, DNA Computing: Bewertung und Simulation, *Diploma thesis at the University of Dortmund, Dept. of Computer Science, LS11*, 116-123, (1998)
15) Udo Feldkamp, Ein DNA-Sequenz-Compiler, *Diploma thesis at the University of Dortmund, Dept. of Computer Science, LS11*, (1999)
16) D. Sen, W. Gilbert, A sodium-potassium switch in the formation of four-stranded G4-DNA. *Nature* **344**, 410-414, (1990)