Molecular Model Checking*

E. Allen Emerson, Kristina D. Hager, and Jay H. Konieczka

Department of Computer Sciences and Computer Engineering Research Center
The University of Texas at Austin,
Austin TX 78712, USA

Abstract. This paper shows how to perform model checking, a technique for automatic program verification, by a DNA algorithm. Our method depends on two ideas. First, Kripke structures can be compactly represented in a DNA substrate, coding each state and each edge by a strand of DNA. Second, satisfaction of temporal eventualities can be achieved through a self-propagating molecular *chain reaction*.

Keywords: biomolecular computation, DNA algorithm, model checking, formal verification

1 Introduction

Model checking [4] (cf. [5]) is a fully automatic method for verifying correctness of finite state concurrent programs. Model checking is based on an efficient search of the program's global state transition graph, to determine whether the program exhibits correct behavior. Model checking is now widely used by computer hardware vendors to design critical portions of microprocessor circuitry, and is showing promise in software verification. The chief limitation is the state explosion problem where the size of the global state graph grows exponentially with the size of the program itself.

Model checking algorithms ascertain whether or not the program state graph defines a model, in the sense of mathematical logic, of a correctness specification formulated in *temporal logic* [6]. Temporal logic is a formalism for describing change over time, well-suited to describing the ongoing computations of computer systems. Temporal operators along a single computation sequence include "sometimes p" (Fp) and "always p" (Gp); we also have the path quantifiers "along all computation sequences h holds" (Ah) and "along some computation sequence h holds" (Eh). Two critical properties are *reachability* of p, EFp, and (its dual) *invariance* of p,AGp, (which is equivalent to $\neg EF\neg p$). If we let *good* denote a predicate characterizing the set of desirable states that a program should remain in invariantly, while *bad* denotes the complement condition $\neg good$, then a program is either correct, $AGgood$, or has the potential to reach an error state, $EFbad$.

* This work was supported in part by NSF grants CCR-020-5483 and CCR-009-8141.
{emerson,khager,jay}@cs.utexas.edu

In this paper we show how model checking can be implemented in a substrate of DNA. DNA molecules are single and double-stranded strings over the four symbol alphabet $\{\mathtt{A}, \mathtt{G}, \mathtt{C}, \mathtt{T}\}$, where each symbol is physically a *nucleotide* molecule. A key advantage is that these molecules are sufficiently small that an immense number of them will fit in a compact volume. This permits very compact representation of extremely large state graphs. For example, a graph of size 10^{18} states can be represented within 1 litre of DNA.

In our approach to DNA-based model checking, each graph node or *state* s is represented in terms of a single-stranded piece of DNA. In practice, there are a multiplicity of identical DNA strands representing each state. Each graph edge $s \longrightarrow t$ is also represented using a strand of DNA. Roughly speaking, the left and right portions are single-stranded, while the middle portion is double stranded. Again, there is typically a multiplicity of such strands representing each edge. The multiple molecular instances of each state and each edge facilitates an efficient DNA-based algorithm exploiting the massive parallelism inherent in DNA.

To model check EFp amounts to calculating the set of states that can reach the set of states p. The original DNA solution encodes the edges of the graph, but, initially, only representations of the states in p are assumed to be present in the DNA. We say each represented state is *marked*. Thus, the states of p are marked initially. In general, when edge $s \rightarrow t$ is present and the state t has already been marked, then the algorithm next marks state s, by a reaction combining the DNA representation of t with the DNA representation of $s \rightarrow t$ to yield the DNA reperesentation of s. In the DNA solution, the marking will proceed in a self-propagating fashion, i.e., as a *chain reaction*.

The remainder of this paper is organized as follows: Section 2 overviews DNA and its use in algorithms. Section 3 gives a more detailed description of the algorithm at the level of multisets of production rules. Section 4 details the DNA implementation. Section 5 describes our experimental results. The concluding Section 6 discusses related work and suggests future directions.

2 DNA

DNA (Deoxyribo-Nucleic-Acid) forms the genetic blueprint of all living cells and many viruses. DNA molecules can be in the form of both single and double stranded strings over the alphabet $\mathbf{N} = \{\mathtt{G}, \mathtt{A}, \mathtt{T}, \mathtt{C}\}$ of component molecules individally called *nucleotides*. The single-stranded strings may be viewed as conventional strings in $\mathbf{N}^+$; physically, adjacent nucleotides along the single strands are "horizontally" bound together by strong covalent bonds. The single strands also combine to form double strands. Each nucleotide has a *Watson-Crick complement*: `G` is complementary to `C`, while `A` is complementary to `T`. A (weak) hydrogen bond will form between Watson-Crick complements. A single horizontal strand such as `AAGTC` will thus anneal to its complementary strand `TTCAG` to yield the double stranded molecule

```
AAGTC
TTCAG
```

due to the complementarity of AA with TT, G with C, T with A, and C with G. The strength of this "vertical" hydrogen bonding of the two horizontal single strands into the double strand increases as the length of the double stranded DNA molecule increases. Because of DNA's properties described above, a solution of single stranded DNA molecules will accrete into long double stranded molecules — provided that the single-stranded molecules sequentially align with respect to Watson-Crick complementarity. Heat can be used to disassociate this "vertical" hydrogen bonding, breaking down a double strand to its consituent, complementary double strands. In general, longer strands require greater temperature to disassociate or "melt".

It should be noted that each individual strand has a directionality; the 5'-phosphate group of the first nucleotide begins the strand while the 3'-hydroxyl group ends the final strand. By convention, such 5' to 3' strands are the top strands. Strands that run in the anti-parallel 3' to 5' direction represent the bottom strands. We could thus depict the above as:

$$5' - \texttt{AAGTC} - 3'$$
$$3' - \texttt{TTCAG} - 5'$$

In most cases, it is not necessary to display the directionality, since we can rely on the notation of top versus bottom strands.

"Sticky ends" refer to an unmatched sequence of bases found at the end of a double stranded molecule, for example:

```
GATACA
CTATGTCCCC
```

Such sticky ends are easily bound any by complementary single strand. or complementary sticky end. For instance, GGGG would bind to the above sticky end molecule to yield

```
GATACAGGGG
CTATGTCCCC
```

The feasibility of using DNA to implement computation was first demonstrated in the fundamental work of Adleman [1]. Adleman showed how to use DNA to encode and solve an NP-complete problem, Hamilton Path, over graphs. Nodes and edges of the graph were encoded as DNA strands, in such a way that the natural reaction of the short strands encoding nodes and edges led to the formation of long strands of DNA encoding paths through the graph. All possibe (simple) paths through the graph were formed in this way, with very high probability. Note that due to the proliferation of DNA molecules each (simple) path is represented in high multiplicity by many identical DNA strands. Recombinant DNA techniques filter the resulting strands to ensure that (i) each starts with the designated start node v_0 and ends with the designated sink node v_n; (ii)

each is of length n nodes, where n is the size of the graph; and (iii) each node v_i appears along each retained path. The set of strands after step (iii) corresponds to the set of Hamilton paths. In the wet lab, Adleman's method proceeds in steps, involving a reactions in a number of *test tubes*, each filled with the appropriate DNA solution. An initial test tube contains DNA solution for the encoded graph. It spontaneously generates all the paths. Next this test tube is "poured" into a second tube so that PCR can amplify paths that start and stop properly (step (i)). Step (iii) involves n test tubes.

Adleman's algorithm for the Hamilton Path problem was a pioneering demonstration of the feasibility of DNA computation. But because Hamilton Path is an NP-complete problem the algorithm was *space inefficient*. The algorithm generates all of the exponentially many candidate solution paths. This generation step proceeds reasonably rapidly because of the massive parallelism. But as the solutions are all generated "at once" it can require a exponential amount of DNA. Another complication, is that it was, in some respect, time inefficient. It required about one week of wet lab work. In large part, this was because it required the use of multiple test tube steps. A reaction in the i^{th} step must complete before the $i+1^{\text{st}}$ step can begin. These factors conspired to limit the size of the problem that could be handled. In Adleman's original experiment, he dealt with a graph of size seven (7) nodes.

3 Model Checking

Model checking is a fully automatic method for verifying correctness of finite state concurrent programs, based on an efficient search of the program's global state transition graph, to determine whether or not the graph defines a model of the correctness specification formulated in temporal logic. Temporal logic provides a formalism for precisely describing change over time. Temporal operators along a computation sequence include "sometimes p" (Fp) and "always p" (Gp); we also have the path quantifiers "along all computation sequences h" (Ah) and "along some computation sequence h" (Eh).

Model checking is now widely used by computer hardware vendors to design critical portions of microprocessor circuitry, and is showing promise in software verification. The chief limitation is the state explosion problem where the size of the global state graph grows exponentially with the size of the program itself.

While there are many correctness properties that can be expressed in temporal logic, the vast majority of importance are simple *safety* properties asserting that nothing bad ever happens. Safety can be captured by a temporal logic formula of the form $AGgood$ which asserts that along all computations at all times condition *good* holds. In other words, *good* holds invariantly. The negation of $AGgood$, denoted $\neg AGgood$, is equivalent to $EFbad$, where we understand *bad* to be the complemented condition $\neg good$. We will focus in this paper on computing the set of states where EFp holds; when p denotes the set *bad* of error states, we can thus also calculate from $EFbad$ the complementary set $AGgood$. Since most software programs are "buggy", as are hardware designs, in most

phases of their development, checking $EFbad$ is in fact the critical activity in most practical applications of model checking.

We are given (labelled) state transition graph $M = (S, R, L)$ where S is a finite set of *states* (graph nodes); $R \subseteq S \times S$ is a binary *transition relation* on S whose members (s, t) are called *transitions* (edges) and denoted $s \rightarrow t$; and a *labelling* function L which assigns to each state $s \in S$ a set $L(s)$ of atomic facts or propositions true at s (such as p, *good*, etc.) A *path* $x = s_0, s_1, \ldots, s_n$ in M is a sequence of states such that for each index $i < n$ edge $s_i \rightarrow s_{i+1}$ is in R. We say EFp holds true of a state s_0 in M, denoted $M, s_0 \models EFp$, provided there exists a path $s_0, s1, \ldots, s_n$ in M such that p holds at s_n.

Model checking of EFp is can be performed efficiently by a backward reachability analysis. Calculating the set of states of M where the formula EFp is true can be done in polynomial (in fact, linear) time in the size of M, on a standard, digital computer. The algorithm starts with p, calculates the predecessors of those states, the predecesssors or the predecessors, and so forth until stabilization. At that time EFp is calculated.

There is an essential difference in the computational difficulty for the two graph problems: model checking EFp versus Hamilton Path. Hamilton Path, an NP-complete problem, can be hard for even small graphs; the algorithm of [1] requires generation of exponentially many potential solutions and an exponential amount of DNA. The model checking reachability problem can be solved time efficiently in-place using a simple "marking" algorithm, as described below. Model checking is nonetheless hard in practice because the state graph is can be immense.

4 The Algorithm

Our algorithm as well as that of [1] are both based on physically representing graphs as (multi-)sets of short DNA strings ordered pairs of states. However, whereas the [1] encoding is designed to promote the formation of long DNA strands representing paths through the graph, our encoding is designed to prevent the formation of such chains. Instead, our encoding of states and edges in the DNA is designed so that satisfaction of temporal modalities such as EFp propagates via a molecular *chain reaction* through the DNA solution. Moreover, for single temporal modalities, our approach is a single test tube reaction.

The algorithm for temporal possibility EFp over a state graph $M = (S, R)$ is based on back propagation from p:

- Each state $s \in S$ is represented by a single-stranded piece of DNA.
- Each edge $s \longrightarrow t$ is represented by a a strand of DNA: the left and right portions are single-stranded sticky ends while the middle portion is double stranded.
- The abstract algorithm can be viewed as "marking" a canonical copy of the graph M. First, mark all states in p. Then mark all states reachable, by back-traversing one graph edge, from some other already marked state. This marking will propagate automatically so that ultimately all states that

can reach P are marked. The abstract algorithm terminates when the ever-increasing set of marked states stabilizes; such stabilization is guaranteed because the graph is finite.

- The DNA implementation of the algorithm will exploit the enormous parallelism afforded by DNA computing. The states in p will be represented in multiplicity by a proliferation of DNA strands. Each edge in R will also be represented in multiplicity by a proliferation of DNA strands. The marking will proceed in a self-propagating fashion, i.e., as a *chain reaction*. The DNA algorithm terminates when the chain reaction terminates so that no new states are marked.
- Each act of marking corresponds to a "production rule" of the form $s \longrightarrow t, t \vdash s$. When a (DNA occurrence of a) marked state t, encounters some (DNA occurrence of an) edge $s \longrightarrow t$, it combines on the right end. This enables an enzyme (as discussed in the next section) to perform a vertical cut in (the DNA molecule for) the edge $s \longrightarrow t$ resulting in (the DNA representation of) state s being marked with its molecule freed. The molecule for s can now propagate the reaction.
- Initial state s_0 of M satisfies EFp exactly when, upon stabilization, (a DNA occurrence of) s_0 is marked. We also say that M, s_0 is a model of EFp.

5 DNA Implementation Issues

In this section we provide details re. the implementation of the above algorithm in DNA. We find it convenient to describe the DNA implementation of the algorithm in terms of forward reachability. Given $M = (S, R, L)$ we let $M^r = (S, R^r, L)$ be the graph obtained from M by simply reversing its arcs: $u \rightarrow v \in R^r$ iff $v \rightarrow u \in R$. Trivially, a path from s to t in M corresponds to a reverse path from t to s in M^r. Thus, computing the set of states where EFp holds over M by backward reachability from p in M can be described in terms of forward reachability from p in M^r. Each back propagation rule $t \longrightarrow s, s \vdash t$ corresponds to forward propagation rule $s \longrightarrow t, s \vdash t$.

The most important implementation issue is how to appropriately encode the states s and edges $s \longrightarrow t$ as DNA strands. Each state s is encoded as a ("top") single strand of DNA of length n. We choose n so that strands of length n are sufficient to code the state space of interest, which should be of size at most 4^n). For each state s, let $\mathtt{s}$ denote the n-symbol string in $\mathbf{N}$ that is the code of s, and we assume that $\mathtt{s} = \mathtt{s_a s_b}$ is decomposed into a head portion $\mathtt{s_a}$ and a tail portion $\mathtt{s_b}$. Each transition $s \rightarrow t$ is coded by a DNA molecule with single-stranded "sticky" left and right ends, and a double-stranded middle portion, of the form:

$$\begin{array}{l} \mathtt{s_a s_b x t_a} \\ \hline \mathtt{x t_a t_b} \end{array}$$

where x denotes a "spacer" string.

The DNA implementation works as follows. The code $\mathtt{s}$ for each state s appears embedded in the codes for all transitions involving s. However, $\mathtt{s}$ does not occur free in the solution. Nor does the Watson-Crick complement $\overline{\mathtt{s}}$ appear free in the solution, for any s — except for the initial state(s) s_0 which do occur free in the initial solution, because we assume each initial state s_0 is marked. In general, each state s that has been marked is represented by a proliferation of copies of $\overline{\mathtt{s}}$, the Watson-Crick complement of the code for s. The production rule $s, s \longrightarrow t \vdash t$ is fired as follows. First, whenever a copy of $\overline{\mathtt{s}}$ binds to a copy of the complementary $\mathtt{s}$ embedded in an occurrence of the encoded transition, this enables an appropriately chosen *restriction enzyme*. The restriction enzyme re is a protein which recognizes a specific DNA subsequence and cleaves the sequence in two at an offset from the recognition site. Here, re is engaged because the newly formed juxtaposition $\overline{\mathtt{s_b x}}$ completes the *recognition sequence* of the enzyme; thus enabled the enzyme re cuts the covalent bonds of the transition at an appropriate distance so as to pop loose $\overline{\mathtt{t}}$. Now t is marked. The propagation of marking is effected through the resulting DNA chain reaction.

We depict a reaction step as follows:

$$\overline{\mathtt{s_a s_b}} \;+\; \frac{\mathtt{s_a s_b x t_a t_b}}{\mathtt{x t_a t_b}} \;\Longrightarrow\; \frac{\mathtt{s_a s_b x t_a}}{\mathtt{s_a s_b x t_a t_b}} \;\overset{re}{\Longrightarrow}\; \frac{\mathtt{s_a s_b x t_a}}{\mathtt{s_a s_b \overline{x}}} \;+\; \overline{\mathtt{t_a t_b}}$$

Our use of a restriction enzyme facilitates the traversal of the graph, without building all paths, which caused an exponential space blowup for [1]. Our state and edge encodings must be viable within the same environmental parameters as the restriction enzyme.

The DNA implementation exploits the fact that if many copies of the marked state $\overline{\mathtt{s}}$ and many copies of the code for transition $s \longrightarrow t$ are present, then many of them will encounter each other and bind, firing the transition. firing the transition. This is justified physically by thermodynamic considerations. Different DNA molecules in solution diffuse "completely" from region regions of high concentrations to regions of low concentrations, as documented by everyday observation with liquids and gasses as well as in countless DNA wet lab experiments.

In the laboratory, to ensure termination of the algorithm, it suffices to let the reaction run for an amount of time determined by the the amount of DNA solution, temperature, and so forth. In practice, a fixed bound of a certain number of hours is sufficient.

6 Experimental Validation

We have validated the basic feasility of our ideas by developing a nucleotide level simulator, and running a variety of experiments on small graphs. The simulator is written in Java. A pre-processor compiles a state graph, described as a set of transitions, into a a corresponding test tube of DNA strings. The simulator then takes as input the test tube of nucleotide strings encoding the graph to be model checked. It simulates interactions at the DNA nucleotide level. It caters for the the existence of multiple strands and their interaction. The action of

the restriction enzyme was implemented by a specific subroutine. Our simulator did not cater for modelling the probability of specific molecular interactions and the effects of temperature and salinity. We tested our DNA algorithm in the simulator on several small example graphs, confirming correct operation of the algorithm.

7 Discussion

We believe that DNA model checking has the potential to significantly increase its scope of automated verification. Before its full practical potential is realized, however, there are a number of engineering issues that must be addressed. We discuss some of those here, providing them as suggested directions for future research:

(i) fault tolerance: it known that DNA can exhibit "nonspecific binding" where two single strands that are almost, but not quite Watson-Crick complements can combine to produce a "faulty" double-stranded segment. For instance, `GGGGGACCCCC` can combine with `CCCCCGGGGGG`. The first 5 positions and the last 5 positions combine correctly in a complementary fashion. However, `A` is associated with `G` in the middle position, anchored by the correct left prefix and right suffix. Most DNA algorithms, including the one we have described, simply ignore this potential problem. It can be ameliorated by physico-chemical means, keeping the DNA solution at the proper temperature and salinity. A mathematical heuristic is to use state encodings that replicate a pattern. For instance, state s might have an associated nucleotide sequence $\overline{s}$ and corresponding encoding $\overline{ssss}$, which is the 4-fold replication. If the probability of a nonspecific binding along $\overline{s}$ is p, $p < 1$, then the probability of specific binding along the encoding is reduced to p^4. Future attention needs to be devoted to determining if "nonspecific" bindings are problematic in practice.

(ii) The restriction enzyme *re* has a finite length between the recognition site for enablement and the action site where it cuts. This can limit the usable length of DNA strands encoding states, and adversely impact the heuristic in item (ii). However, it is possible that advances in protein engineering will ameliorate this difficulty.

(iii) Our algorithm, like that of Adleman [1], presupposes that the state graph has already been represented in the DNA substrate. In practice, the problem of constructing from the program text its corresponding state graph M in the DNA solution needs to be addressed. A theorem of [2] suggests a way to solve this in principle.

8 Conclusion

We have shown that model checking can be implemented in a DNA substrate. The area of DNA (graph) algorithms was pioneered by Adleman [1]. As noted, Adleman's algorithm in general used exponential space as it was oriented to

solving NP-complete problems. The thrust of work in this area is in fact to exploit the massive parallelism of DNA to solve conventionally intractable problems.

Our work exploits DNA differently. We use its capacity to physically, explicitly store astronomically large state graphs. Its massive parallelism in the form of difusing chain reactions automatically propagate temporal satisfiability (of reachability and eventuality properties). There is no exponential space blowup.

Perhaps the work closest to ours is that in [2]. Independently, [2] had proposed DNA-based algorithms for dynamic programming and outlined a graph reachability algorithm. There are several key distinctions between our work and theirs. First, their work is not intended for program verification applications. Second, their work involves many iterations of emptying and filling test tubes. Essentially on the $i+1$-st iteration nodes just discovered to be in $EF^{\leq i+1}p$ are processed. Such multiple test tube operations can be very time consuming and make questionable the suitability of their approach for use in verification, where 100's or 1000's of iterations might be required to traverse a typical hardware design. In contrast, the self-propagating nature of our algorithm permits all computation to be done in one basic test tube step. A third technical distinction is that our work exploits double-stranded DNA while theirs only uses single-stranded DNA.

Our work has demonstrated the basic scientific feasibility of DNA model checking. We believe this is an approach to model checking that has the potential to significantly increase its scope of automated verification. Before its full practical potential is realized, there are a number of engineering issues that must be addressed. We discuss some of those here providing them as suggested directions for future research:

In any case, we believe this approach is promising and merits additional study.

References

1. Adleman, L., "Molecular Computation of Solutions to Combinatorial Problems," Science, Vol. 266, 11 November 1994, pp. 1021-1023.
2. Eric B. Baum and Dan Boneh, "Running dynamic programming algorithms on a DNA computer", http://www.cs.princeton.edu/ dabo/, 1996.
3. A. Brenneman and A. Condon, "Strand Design for Bio-Molecular Computation", (Survey Paper), to appear, http://www.cs.ubc.ca/ condon/#Papers
4. E. M. Clarke and E. A. Emerson, "The Design and Synthesis of Synchronization Skeletons Using Temporal Logic", *Proceedings of the Workshop on Logics of Programs*, IBM Watson Research Center, Yorktown Heights, New York, Springer-Verlag Lecture Notes in Computer Science, #131, pp. 52–71, May 1981.
5. Jean-Pierre Queille, Joseph Sifakis, " Specification and verification of concurrent systems in CESAR", Symposium on Programming 1982: 337-351.
6. Amir Pnueli, "The Temporal Logic of Programs", FOCS, 1977.