

Modelling, Simulating and Verifying Turing-Powerful Strand Displacement Systems

Matthew R. Lakin and Andrew Phillips

Microsoft Research, Cambridge, CB3 0FB, UK
aphillip@microsoft.com

Abstract. We demonstrate how the DSD programming language can be used to design a DNA stack machine and to analyse its behaviour. Stack machines are of interest because they can efficiently simulate a Turing machine. We extend the semantics of the DSD language to support operations on DNA polymers and use our stack machine design to implement a non-trivial example: a ripple carry adder which can sum two binary numbers of arbitrary size. We use model checking to verify that the ripple carry adder executes correctly on a range of inputs. This provides the first opportunity to assess the correctness and kinetic properties of DNA strand displacement systems performing Turing-powerful symbolic computation.

1 Introduction

Biomolecular computation devices can interface directly with living tissue [1], opening up exciting new possibilities for autonomous medical intervention at the cellular level. The programmable nature of DNA makes it ideally suited as a material to implement such biomolecular computers. As techniques for DNA synthesis and manipulation continue to improve, we can look towards using DNA to implement more sophisticated computational functions.

Classical work on computability theory has produced a number of equivalent universal computational models, such as Turing machines [2] and stack machines. Both of these paradigms are based on symbolic computation, where computation proceeds via the manipulation of abstract mathematical symbols which denote data values. These paradigms have the virtues of simplicity and compactness, as simple data structures are modified in-place. Nucleic acids are excellent materials for implementing symbolic computation, because distinct symbols can be straightforwardly represented as distinct, non-interfering nucleotide sequences, and data structures can be directly realized in the physical structure of the DNA species.

In this paper we study the design and analysis of biomolecular implementations of universal symbolic computation. Our chosen framework for molecular computation is DNA strand displacement [3], which is an established technique for the principled design of DNA computing systems. Our starting point is the work of Qian et al. [4], who proposed a design of a stack machine using DNA strand displacement. A stack machine consists of finitely many stacks (first-in,

first-out memory storage) and a finite state machine which can add symbols to (push), and remove symbols from (pop), the top of these stacks. In [4] the stack data structures have a direct physical representation as DNA polymers which can interact at one end only. This design is a simple and elegant translation of a universal scheme for symbolic computation into DNA, which can be used to efficiently simulate a Turing machine [2].

We tackle the formal design and analysis of universal DNA computers by encoding them in the DSD programming language [5]. This is a domain-specific language with a well-defined operational semantics that reflects the key assumptions of strand displacement systems. DSD has previously been used to model a range of strand displacement devices, including logic gates and chemical reaction networks. However, previous versions of DSD did not support the formation of extensible polymers, which are required to encode stack data structures in DNA. Furthermore, the DSD simulation algorithm required all models to be compiled to a fixed set of reactions and was therefore unable to simulate Turing-powerful computation, which can generate potentially unbounded numbers of reactions. Hence we extend the DSD semantics and simulation algorithm to support the formation of linear heteropolymers.

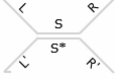
This paper is structured as follows. Section 2 presents an extension of the DSD language syntax and semantics [5] to model the formation of linear DNA heteropolymers, while Section 3 presents a stochastic simulation algorithm, based on [6], for DNA strand displacement systems involving polymers. Section 4 presents an encoding of a stack machine design in the DSD language which is optimised for mechanical verification. Finally, Section 5 presents an implementation of a classic circuit from digital electronics, a ripple carry adder, which computes the sum of two binary numbers of arbitrary size, including results from stochastic simulations and model-checking which provide evidence that the DNA implementation of the adder is correct. To our knowledge, this is the largest DNA strand displacement system to be formally verified.

2 Polymers in DSD

The DSD language was introduced in [5] as a means of formalising the designs of DNA strand displacement systems. Here we recap the basics and extend the semantics to allow polymerisation reactions between complexes.

The syntax of the DSD language is defined in terms of *domains* M and *domain sequences* S , L , R . A domain M represents a nucleotide sequence with explicit information about the orientation of its 3' and 5' ends. We assume that distinct domains are mapped to distinct, non-interfering nucleotide sequences using established techniques [7]. A domain can be a *long domain* N or a *short domain* N^\sim (shown in black in images). We assume that toeholds are sufficiently short to hybridize reversibly (4–10nt) whereas long domains are sufficiently long to hybridize irreversibly (>20nt). A domain sequence S is a concatenation of finitely many domains with the same orientation, whereas domain sequences L and R can potentially be empty. The *complement* S^* of a domain sequence S is the domain sequence that hybridizes with S via Watson-Crick complementarity.

Table 1. Graphical and textual syntax of the DSD programming language

Syntax	Description	Syntax	Description
$\{\underline{S}\}$	Lower strand with sequence S	$\{L'\}\langle L\rangle[S]\langle R\rangle\{R'\}$	Double stranded complex $[S]$ with overhanging single strands $\langle L\rangle$, $\langle R\rangle$ and $\{L'\}$, $\{R'\}$
$\langle \underline{S} \rangle$	Upper strand with sequence S		
$C1:C2$	Complexes joined by lower strand	$C1::C2$	Complexes joined by upper strand

Domain sequences are used to construct DNA species, as shown in Table 1. A species can either be a single *strand* A or a *complex* C . A strand can either be an *upper* strand $\langle S \rangle$ (drawn with the 3' end towards the right) or a *lower* strand $\{S\}$ (drawn with the 3' end towards the left). We assume that species are equal up to rotation symmetry, so every upper strand has a corresponding lower strand, and vice versa. Complexes are formed by joining one or more *segments* of the form $\{L'\}\langle L\rangle[S]\langle R\rangle\{R'\}$, which consists of a double-stranded region $[S]$ with four overhanging strands. This represents an upper strand $\langle L S R \rangle$ bound to a lower strand $\{L' S^* R'\}$ by hybridization between S and S^* . For compactness, only the upper sequence of the double-stranded region is written explicitly, and we omit empty overhanging strands. Complexes can be formed by concatenating segments either along the lower strand, written $C1:C2$, or along the upper strand, written $C1::C2$.

Systems D typically involve many species in parallel, written $D_1 \mid \dots \mid D_n$. We abbreviate K parallel copies of the same system D as $K*D$. The language also includes features for expressing the logical structure of the system: a domain N can be restricted to the system D , written **new** $N D$, which represents the assumption that N and N^* do not appear outside of D . The language also supports module definitions of the form $X(\tilde{m})=D$, where \tilde{m} is a list of module parameters and $X(\tilde{n})$ is an instance of the module X with the parameters \tilde{m} replaced by values \tilde{n} . We assume a fixed collection of non-recursive module definitions. A key assumption of the DSD language is that species only interact via complementary toeholds: we enforce this by requiring that no long domain and its complement are simultaneously exposed. Finally, we note that the syntax of the DSD language is constrained so that overhanging single strands are the only secondary structure which a complex may possess, which rules out branching structures.

Figure 1 presents elementary reduction rules for the DSD language which formalise basic strand displacement reactions. Rules (RB) and (RU) define the *binding* of a strand to a complex via a complementary toehold, together with the corresponding *unbinding* reaction since we assume that toeholds hybridize reversibly. The rates of these reactions are determined from the toehold N . Rule (RC) accounts for the case when an overhanging toehold in the lower strand is *covered* by the complementary toehold in the upper strand: this is irreversible as the resulting long double-stranded segment is thermodynamically stable. Rules

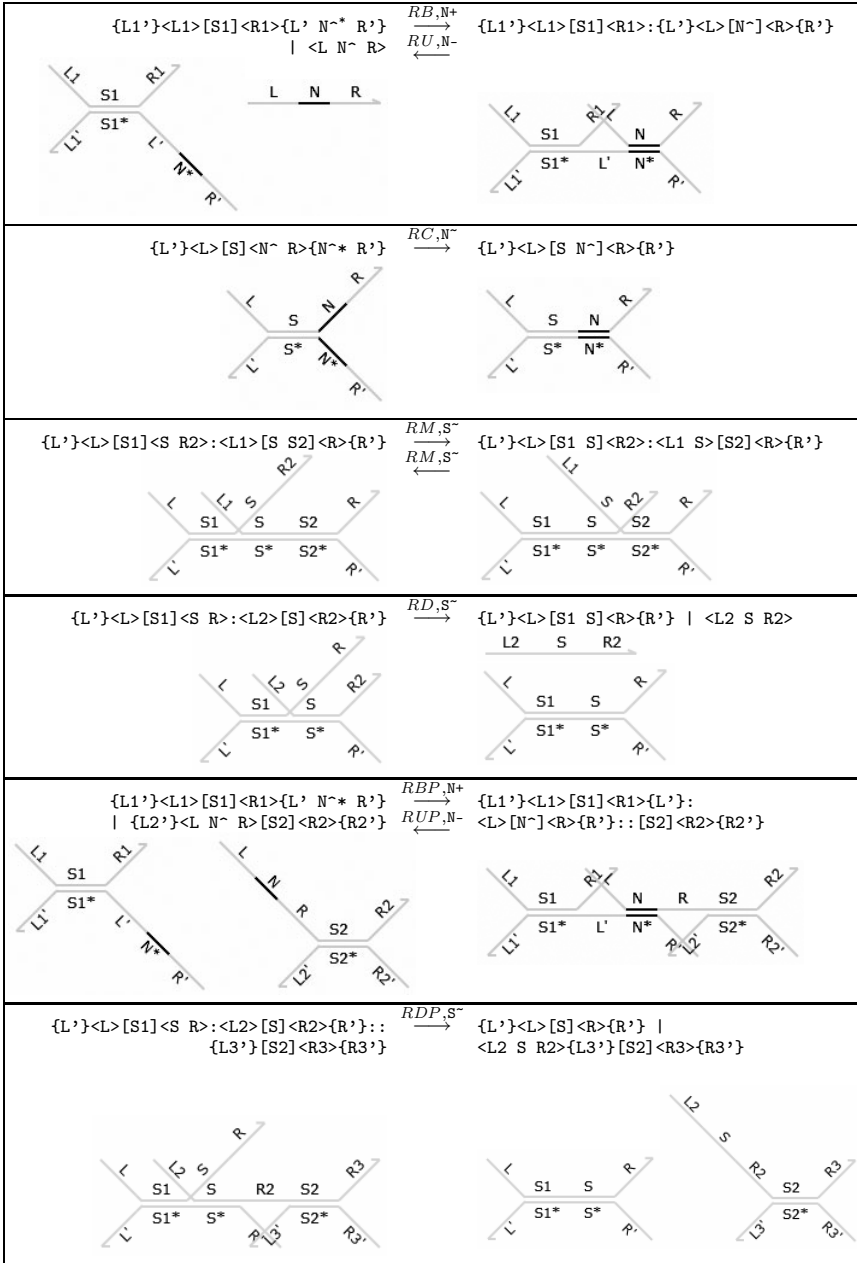


Fig. 1. Elementary reduction rules of the DSD language with polymers. We let S^- denote the migration rate of a domain sequence S , and we let N^+ and N^- denote the binding and unbinding rates, respectively, of a toehold N^- . We assume that $\text{fst}(R_2) \neq \text{fst}(S_2)$ for rule (RM). This ensures that branch migration is maximal along a given sequence and that rules (RM) and (RD) are mutually exclusive.

(RM) and (RD) define *branch migration* and *strand displacement* reactions, respectively. In each of these, the overhanging junction in the upper strand performs a random walk which, in the case of rule (RD), completely displaces a strand from the complex. Note that branch migration is a reversible process whereas strand displacement is irreversible.

The final two rows in Figure 1 present additional reduction rules which do not feature in previous published semantics for the DSD language [5]. These rules permit complexes to interact with each other to form larger complexes which we refer to as polymers. Rule (RBP) allows two complexes to bind on a shared toehold to form a longer complex, and rule (RUP) allows the larger complex to break apart when the toehold unbinds. Rule (RDP) extends the strand displacement rule (RD) to the case where the displaced strand was previously holding two complexes together. Note that the reduction rules ensure that the only toeholds which may interact are located in the main trunk of the complex as opposed to in the overhangs: this prevents the formation of branching structures while permitting the growth of linear heteropolymers.

The rules presented in Figure 1 define the basic forms of reduction in the extended DSD language. However, these reactions may take place within larger contexts, so to complete the language semantics we require some additional contextual rules. These include adding segments on either side of the reacting segment, mirroring the species horizontally and vertically, and rotating them. We omit the contextual rules here for reasons of space. In the case of rule (RBP), we note that the overhangs containing the complementary toeholds must appear at the very ends of the complexes: in other words, polymers can only interact end-to-end. This can be formalised by a careful choice of contextual rules which only allow additional structure at one end of interacting complexes. This restriction is necessary to prevent branching structures from arising dynamically.

3 Stochastic Simulation of Polymerising Systems

The standard Gillespie algorithm for exact stochastic simulation [8] requires that the entire chemical reaction network (CRN) of all possible reactions between reachable chemical species must be known before the simulation begins. However, in the case of DNA strand displacement systems with polymers we cannot necessarily pre-compute the CRN because it may be infinite, as we could (in principle) keep adding monomers to produce an ever-increasing polymer chain.

We avoid this problem by using the just-in-time simulation algorithm from [6]. This is an extension of the Gillespie algorithm in which compilation of species interactions is interleaved with simulation steps. The just-in-time simulation algorithm can be summarised as follows:

1. Compute the CRN of all possible initial reactions between the initial species only, *without* recursively computing reactions involving the products of those initial reactions.
2. Compute reaction propensities according to the Gillespie algorithm, and randomly select the next reaction with probability proportional to its propensity.

3. Execute the next reaction by modifying the species populations and incrementing the simulation time according to the Gillespie algorithm.
4. If executing the reaction produced any new species which have not yet been seen in the system then compute any interactions between the new species and the existing species in the system, and expand the CRN accordingly.
5. Repeat from step 2.

Thus we dynamically update the set of possible reactions as the simulation proceeds, rather than computing all possible reactions up front. Hence we compute only the needed subgraph of the CRN. This can offer significant speedups when the CRN is very large, and is the only feasible approach when the CRN is infinite, as in the case of most polymerising systems. The stochastic simulation is exact since all probabilities are computed exactly at each step.

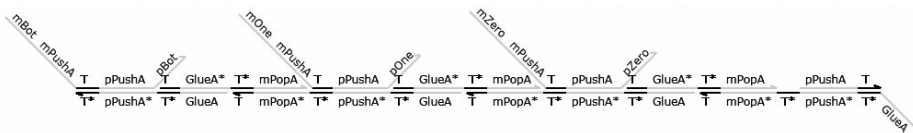
4 Modelling Stack Machines in DSD

In this section we present a novel stack design which is a variant of the stack encoding from [4]. Our design was formalised, visualised and analysed using the Visual DSD tool¹. Our primary goal in designing a new stack implementation is to produce an encoding which is amenable to automated verification. Thus we aim to eliminate speculative stack manipulation reactions and irreversible steps in reaction gates which could occur at any time after the outputs are produced.

4.1 A Variant Stack Encoding

The stack design from [4] has the property that fuel monomers specific to the various symbols that might be pushed onto the stack are continually interacting with the stack, in the hope that the symbol strand itself may arrive to complete the reaction, as in Figure 3 of [4]. Furthermore, that Figure shows that the fuel strands which can deconstruct the stack are also continually interacting with the stack, in the hope that other species may arrive to complete the reaction. This means that there are always a large number of possible stack-based interactions, and consequently the graph of possible states for these systems is very large indeed, making it infeasible to perform analyses such as model checking on the resulting CTMC.

In order to efficiently simulate a Turing machine, more than one stack is needed for data storage. Thus we must assign a unique *type* to each stack so that they can be correctly addressed. In our stack encoding, the stack $[[::1::0]$, of type *A*, is represented by the following DNA complex. Note that we write the top of the stack on the right-hand side in our textual notation, to match the visualisations.



¹ Visual DSD is available online at <http://research.microsoft.com/dna>.

In our encoding, a symbol 1 on stack A is represented by a bound upper strand of the form $\langle \text{mOne mPushA T}^{\sim} \text{pPushA pOne} \rangle$, which we refer to as the *push strand* 1_A . In this paper we use three kinds of symbol: a special *bottom* symbol \perp which signals an attempt to pop from an empty stack and two symbols corresponding to 1 and 0 respectively. Symbols are represented using a history-free scheme similar to that used in [4], except that we separate the nucleotide sequences on either side of the toehold into two long domains: one specific to the stack type (A here) and one specific to the symbol in question. We restrict ourselves to ASCII syntax, writing mX and pX for the negative (towards the 5' end) and positive (towards the 3' end) sequences, which were referred to as ^-X and ^+X respectively in [4].

Each stack complex has a single exposed T^{\sim} toehold, which serves as the initiation site for both *push* and *pop* reactions. The reaction to *pop* a symbol from stack A is initiated by the *pop strand* $\text{PopA} = \langle \text{mPopA T}^{\sim} \text{pPopA} \rangle$ which begins the clockwise sequence of reversible reactions shown in Figure 2. The fuel species FA1–4 are assumed to be present in abundance. Overall, these reversible reactions interconvert between the stack $A = [::1$ and the PopA strand, and the stack $A = []$ and the 1_A strand. The *push* reaction is initiated by a push strand and is obtained as the reverse of the above reaction scheme, reading anti-clockwise in Figure 2. When attempting to pop from an empty stack the reactions proceed as in Figure 2, except that the resulting complex is not a valid stack structure. The *bottom strand* $\perp_A = \langle \text{mBot mPushA T}^{\sim} \text{pPushA pBot} \rangle$ serves as an error indicator, signalling that an attempt has been made to pop from an empty stack.

Our stack design allows us to initiate pushing or popping by the interaction of a single strand with a stable stack complex, without speculative binding and unbinding reactions as in [4]. Furthermore, we can use a smaller set of backbone monomers for each stack type: for a given stack type A we only require the four fuel species FA1–4 from Figure 2 because any symbol can be joined to the main backbone of the stack by the common pPushA domain. Hence the number of domains required scales with the sum of the number of stacks and the number of symbols, whereas in the encoding of [4] it scales with the product (because there the separate nucleotide sequences denoting the stack and the symbol parts of the $\langle \text{mOne mPushA T}^{\sim} \text{pPushA pOne} \rangle$ strand are merged so the strand has the form $\langle \text{mOneA T}^{\sim} \text{pOneA} \rangle$). In our design it is crucial that the pop strand employs the history-free encoding from [4], so it can initiate a leftward displacement reaction to break apart the polymer structure.

4.2 Implementing a Stack Machine in DNA

A stack machine consists of finitely many stacks along with a finite state control program. Thus, a configuration of a stack machine consists of the current state and the current contents of the stacks. As discussed above, the symbols in a given stack are encoded in the nucleotide sequences of the overhanging single strands attached to the polymer backbone of the corresponding stack complex. We encode the current state of the machine by a single complex of the form

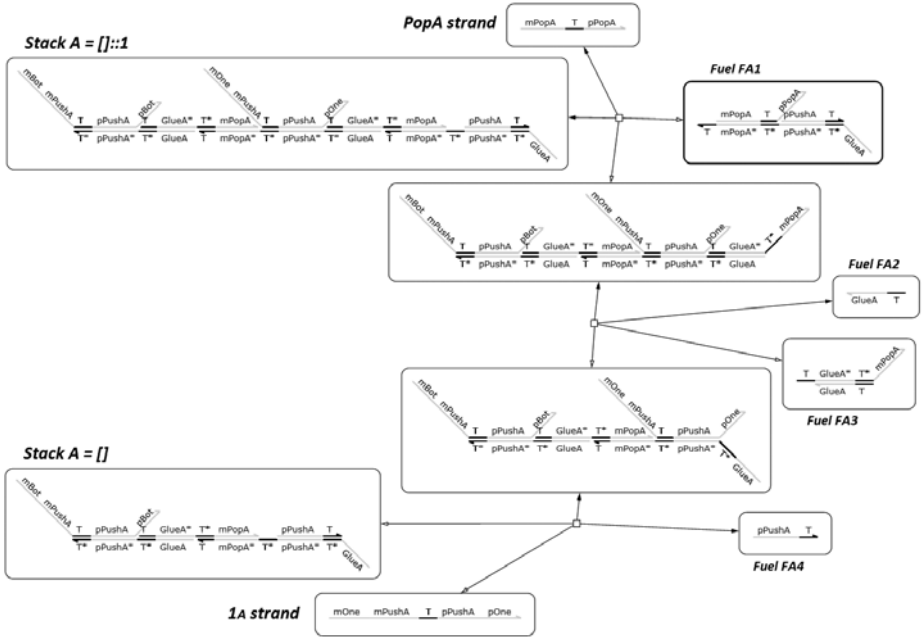


Fig. 2. Example CRN for reversible stack manipulation reactions: pushing and popping a non-empty stack

$S1 = \langle H \ T \ pS1 \rangle$, where we refer to H as the history domain and where $pS1$ is a domain which informs us that the machine is currently in state 1. The history domain is irrelevant when determining the current state of the system, and we will see below why we allow state strands to have an arbitrary history domain. We require that only one state strand is present in solution at any one time, so there can be no confusion over the current state of the machine.

Stack manipulation operations are implemented as described in Section 4.1, and we encode state transitions using chemical reaction gates. These accept as input the current state strand and the output strand from the stack manipulation reaction occurring in that state and produce as output the state strand and stack manipulation initiator strand corresponding to the next state according to the stack machine program. Figure 3 presents the CRN for a reaction gate implementing the reaction $1_A + S1 \longrightarrow S3 + PopB$, which assumes that the symbol 1 has just been read from stack A in state 1, and the transition is to state 3 where we must pop from stack B . In the CRN, the bold nodes denote the species initially present. This gate accepts the input strands $1_A = \langle mOne \ mPushA \ T \ pPushA \ pOne \rangle$ and $S1 = \langle H \ T \ pS1 \rangle$ (where H is an arbitrary history domain) and produces the output strands $S3 = \langle mA \ T \ pS3 \rangle$ and $PopB = \langle mPopB \ T \ pPopB \rangle$. Here, the domain mA is a private history domain which is unique to this particular reaction gate. This allows us to use a long fuel strand with multiple toeholds to eject both of the outputs and render the

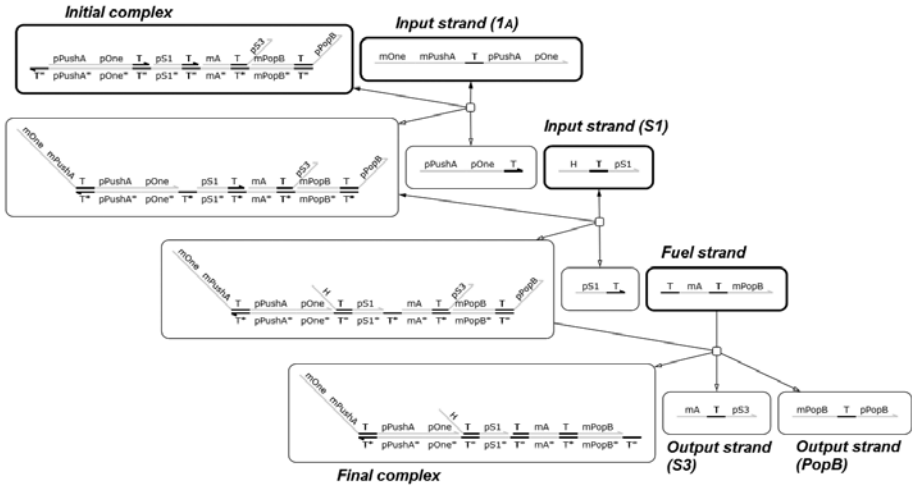


Fig. 3. Example CRN for an irreversible stack machine transition. Nodes with a bold outline indicate species required to be present initially.

complex unreactive in a single step. This helps to restrict the number of states in the CTMC because the chemical reactions corresponding to different steps of the stack machine computation are separated by these irreversible displacement reactions. We are left with a final complex which we consider to be unreactive because there is no other species in the system which can displace the entirety of the long fuel strand. We do not add an extra toehold to the fuel strand to completely seal off the complex because this can lead to unwanted interference caused by fuel strands reacting with the stack monomers.

A stack machine terminates when it enters an *accepting* or *rejecting* state, which can have no outgoing transitions. State transitions which enter one of these states are implemented using a reaction gate similar to that from Figure 3, except that in this case we *can* add an extra toehold to the fuel strand in order to completely seal off the final complex without causing unwanted interference. Reaction gates which implement transitions into an accepting or rejecting state do not produce a strand to initiate another stack operation and this, together with the fact that the fuel strand completely seals off the complex, means that all chemical reactions in the system cease. Hence the CTMC has a well-defined terminal state from which no reactions are possible. This makes it more convenient to ask questions about the final state of the machine.

5 DSD Stack Machine Example: Ripple Carry Adder

As a non-trivial proof of concept we implemented a binary adder in DSD using the stack machine encoding described above. Figure 4 presents the stack machine program for a ripple carry adder which iteratively sums the corresponding bits

from two binary numbers while maintaining a carry bit. The binary numbers are stored in stacks by using different symbols to denote 0 and 1. States which involve popping from a stack have three outgoing transitions (depending on whether one, zero or bottom was popped) and states which involve pushing onto a stack have just one outgoing transition. For the sake of clarity, the state graph in Figure 4 omits a rejecting state along with the transitions into this state: the missing transitions are from state 2 when 0 is popped from stack B , from states 3 and 4 when $B = []$ and from states 5, 6 and 7 when $C = []$. These transitions signal an error when the two inputs are of different lengths or when the carry bit is not present as expected.

The machine reads input from stacks A and B (without loss of generality we assume that both inputs comprise the same number of bits) and takes its carry bit from C (initially zero). For each pair of input bits the stack machine implements a full adder and by iterating the loop we get the effect of a ripple carry adder. When it terminates, the machine has written the sum of the two inputs into X along with a carry-out bit in C . Due to the first-in, first-out nature of the stack data structure, the endianness of the output in X is flipped relative to that of the inputs, though this could be rectified by a subsequent reversing operation if necessary.

5.1 Stochastic Simulation

Figure 5 presents an example² of a stochastic simulation for 1-bit addition with inputs $A = []::1$, $B = []::0$ and $C = []::0$. This plot was obtained using the simulation algorithm described in Section 3. It shows which of the state strands has population 1 at a given time during the run, which allows us to trace the execution of the machine. Comparing the sequence of states from this timeline with the state diagram from Figure 4 shows us that the machine did in fact go through the expected sequence of states. Furthermore, the contents of the output stacks at the end of this simulation run were $X = []::1$ and $C = []::0$, which agrees with the truth table from Figure 4. Thus we have some preliminary evidence that our stack machine program is working correctly.

The simulation plot from Figure 5 also gives us some information regarding the kinetic behaviour of the stack machine implementation. In particular, we observe that the machine spends far longer in states 6 and 10 than in any of the other states. These bad kinetics are caused by the excess of reaction complexes relative to the single stack complex. If a strand could bind either to a stack or to a reaction complex, it will be far more likely to bind to the reaction complex as they are present in excess. We can attenuate this effect to an extent in our simulations by reducing the population of fuels. However, we must strike a balance between providing enough fuel to finish the computation and maintaining reasonable kinetics. Furthermore, in general computations may be arbitrarily long and we may not know the optimal amount of fuel in advance. This is not an

² DSD and PRISM source code for the models discussed in Section 5 are available online at <http://research.microsoft.com/dna/dna17.zip>.

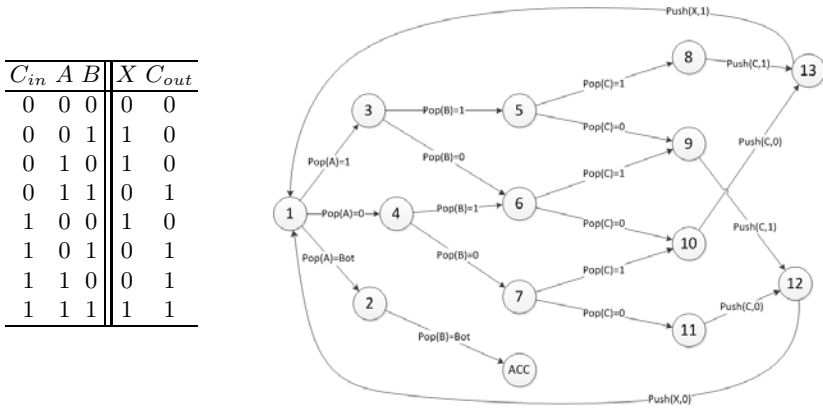


Fig. 4. (Left) Truth table for a 1-bit full adder, which takes two bits and a carry bit as input and produces an output bit and a carry output bit. (Right) State diagram for a stack machine implementation of a ripple carry adder, where state 1 is the initial state.

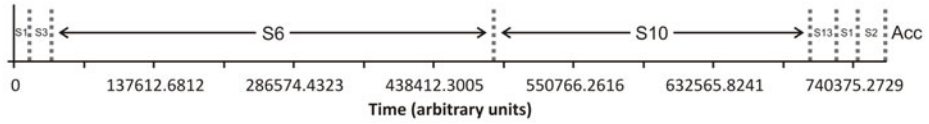


Fig. 5. Example stochastic simulation plot showing the populations of state strands during an execution of the ripple carry adder stack machine. The populations switch between zero and one as the stack machine moves through the sequence states defined by program.

artefact of our stack machine encoding—the issue also exists with the original design proposed in [4]. However, in that paper there was no stochastic simulation available to observe the kinetic behaviour of the stack machine.

5.2 Model Checking

We used the PRISM model checker [9] to verify that the ripple carry adder, given particular inputs, satisfies certain properties expressed as temporal logic formulae. To demonstrate that the stack machine works correctly for given inputs, we used PRISM to check that the following properties hold of the CTMC of the system. We give informal descriptions as well as example PRISM queries:

1. the system always goes through the correct sequence of state transitions and eventually reaches a terminal state which contains the expected output species (P=? [F(state_is_X & F(... & F(state_is_Y & ‘deadlock’ & outputs_correct) ...)]);

Input A			Input B			Output X	Output C	Result
MSB	LSB	Value	MSB	LSB	Value	LSB	MSB	Value
0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	1
0	0	0	1	0	2	0	1	2
0	0	0	1	1	3	1	1	3
0	1	1	0	0	0	1	0	1
0	1	1	0	1	1	0	1	2
0	1	1	1	0	2	1	1	3
0	1	1	1	1	3	0	0	4
1	0	2	0	0	0	0	1	2
1	0	2	0	1	1	1	1	3
1	0	2	1	0	2	0	0	4
1	0	2	1	1	3	1	0	5
1	1	3	0	0	0	1	1	3
1	1	3	0	1	1	0	0	4
1	1	3	1	0	2	1	0	5
1	1	3	1	1	3	0	1	6

Fig. 6. Table of verification results for all possible pairs of 2-bit inputs to the ripple carry adder (with initial carry bit zero). Output values in **boldface** were computed using PRISM and are known to be the final state of the system irrespective of which particular trajectory the system follows.

2. there is always precisely one complex for each stack type ($\text{stack_X}=1$); and
3. there is always *at most* one state strand ($\text{state_strands}\leq 1$). This is not an equality because the state strand may be bound to a reaction complex.

In the above examples, `outputs_correct` returns true if the state contains the expected output species and `state_is_X` returns true if a state strand corresponding to state X is present in solution. The “`deadlock`” label identifies a terminal state of the CTMC and `stack_X` and `state_strands` return the populations of all stack complexes corresponding to stack X and the total population of state strands, respectively. The temporal logic formula $\mathbf{F}\phi$ holds if the system must eventually reach a state satisfying ϕ .

We used PRISM to verify the correctness of all possible pairs of 2-bit inputs (with the initial carry bit set to zero). The results are presented in Figure 6. We were able to show that all four properties listed above hold for all 16 different input pairings, and that we observe the correct output species in the terminal state in all cases. We similarly verified a larger system with two 8-bit inputs, to show that the model checking approach can scale to larger inputs.

Finally, Figure 7 shows how the numbers of states and transitions in the CTMC scale with the initial number of bits in the inputs stacks A and B . Thanks to our reaction gate design we see linear increases in numbers of both states and transitions with increasing input size.

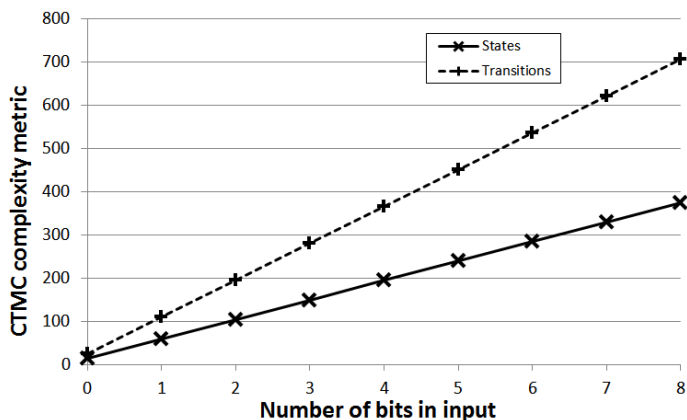


Fig. 7. CTMC complexity metrics. Each point was calculated for a single pair of inputs of that size: the values of the metrics are identical or very similar for different inputs of the same size.

6 Related Work

Theoretical work on the computational power of stochastic chemical reaction networks has shown that chemical systems with polymerisation are Turing-powerful [10,11] but also that *finite* stochastic chemical reaction networks can simulate register machines (and hence Turing machines) with an arbitrarily small probability of error [12,13]. The trick here is to use the *populations* of certain species to denote the numerical values stored in the registers. Jiang et al. [14] have demonstrated how imperative code (which may include arithmetic and while-loops) can be compiled down to stochastic chemical reactions, again using molecular populations to store numerical values. This approach relies on a chemical clock signal to synchronise operations, in order to minimise errors. It is believed that this combination of features is sufficient to make the system Turing-powerful.

Turning to symbolic approaches, Rothmund [15] proposed a design for a universal Turing machine which uses restriction enzymes and ligases to perform operations on a tape encoded as a double-stranded DNA complex. We have already cited the stack machine encoding proposed by Qian et al. [4] as the inspiration for the work reported in this paper.

7 Discussion

From an experimental viewpoint, the main issue with the stack machine designs presented in this paper and in [4] is that they call for a single complex to represent each stack. This is problematic for a number of reasons: it is difficult to produce a single complex with a given design in the lab and it introduces numerous points of failure into the system. If one stack becomes corrupted or forms unwanted secondary structure then the whole system fails. Thus it would be desirable

to invent an alternative stack machine design in which there are many copies of each stack complex (and many copies of the state strand) and the updates to the stacks are synchronised, for example using a clock signal such as that proposed in [14]. This would probably require a different scheme for representing stacks, because the reversible stack manipulation primitives used above, and in [4], mean that stack operations could be undone before the synchronisation actually occurs.

We noted in Section 5.1 that increasing the initial populations of fuels, in order to enable long-running computations, can have adverse effects on the simulation kinetics. In the model this can be addressed by using the `constant` keyword of the DSD language to specify that the populations of certain species (such as fuels) should be fixed throughout the simulation. In practice, a more complex experimental setup would be required in which the population of fuels can be replenished, either continually or at regular intervals. In principle, constant replenishment of DNA fuel should allow long-running, or even unbounded, computations (assuming that all computation steps are error-free).

In Section 5.2 we used model checking to provide some formal verification that our stack machine examples work as expected. We were able to demonstrate some scalability by similarly verifying the result of adding a pair of eight-bit inputs. As shown in Figure 7, the size of the CTMC for our stack machine programs varies linearly with the sizes of the inputs. This was a key goal which motivated various design choices, such as the use of private history domains on the single strands which denote the current state of the machine. In general, however, the brute force approach to model checking does not scale to large systems with many different species and large populations. It may be possible to exploit work on modular model checking [16] to avoid this problem.

Another limitation of model checking is that it only verifies properties of the collection of starting species. We were able to verify that all 2-bit inputs are summed correctly, but we cannot derive a proof that the ripple carry adder works correctly for all input sizes. We would probably need to use an interactive theorem prover to prove such results mechanically. This would require formalising the DSD language in said theorem prover, which could be a valuable exercise in itself.

Acknowledgements. We thank Dave Parker for help using PRISM, and Erik Winfree and Lulu Qian for useful discussions on the stack machine design from [4].

References

1. Venkataraman, S., Dirks, R.M., Ueda, C.T., Pierce, N.A.: Selective cell death mediated by small conditional RNAs. *Proc. Natl. Acad. Sci. U S A* 107(39), 16777–16782 (2010)
2. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Mathematical Society* s2-42(1), 230–265 (1937)
3. Zhang, D.Y., Seelig, G.: Dynamic DNA nanotechnology using strand-displacement reactions. *Nat. Chem.* 3, 103–113 (2011)

4. Qian, L., Soloveichik, D., Winfree, E.: Efficient turing-universal computation with DNA polymers. In: Sakakibara, Y., Mi, Y. (eds.) DNA 16 2010. LNCS, vol. 6518, pp. 123–140. Springer, Heidelberg (2011)
5. Phillips, A., Cardelli, L.: A programming language for composable DNA circuits. *J. R. Soc. Interface* 6(suppl 4), S419–S436 (2009)
6. Paulevé, L., Youssef, S., Lakin, M.R., Phillips, A.: A generic abstract machine for stochastic process calculi. In: Proc. CMSB 2010, pp. 43–54. ACM, New York (2010)
7. Zhang, D.Y.: Towards domain-based sequence design for DNA strand displacement reactions. In: Sakakibara, Y., Mi, Y. (eds.) DNA 16 2010. LNCS, vol. 6518, pp. 162–175. Springer, Heidelberg (2011)
8. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.* 115, 1716–1733 (2001)
9. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
10. Bennett, C.H.: The thermodynamics of computation—a review. *Int. J. Theor. Phys.* 21(12), 905–939 (1982)
11. Cardelli, L., Zavattaro, G.: Turing universality of the biochemical ground form. *Math. Struct. Comp. Sci.* 20(1), 45–73 (2010)
12. Soloveichik, D., Cook, M., Winfree, E., Bruck, J.: Computation with finite stochastic chemical reaction networks. *Nat. Comput.* 7, 615–633 (2008)
13. Cook, M., Soloveichik, D., Winfree, E., Bruck, J.: Programmability of chemical reaction networks. In: Condon, A., Harel, D., Kok, J.N., Salomaa, A., Winfree, E. (eds.) *Algorithmic Bioprocesses*, pp. 543–584. Springer, Heidelberg (2009)
14. Jiang, H., Riedel, M.D., Parhi, K.K.: Synchronous sequential computation with molecular reactions. In: *Design Automation Conference*, San Diego, California, USA, June 5–10 (2011)
15. Rothmund, P.W.K.: A DNA and restriction enzyme implementation of Turing machines. In: Lipton, R.J., Baum, E.B. (eds.) *DNA Based Computers: DIMACS Workshop*, held April 4, pp. 75–120. American Mathematical Society, Providence (1996)
16. Kupferman, O., Vardi, M.Y.: An automata-theoretic approach to modular model checking. *ACM T. Progr. Lang. Sys.* 22(1), 87–128 (2000)