

Design and Analysis of DNA Circuits using Probabilistic Model Checking

Luca Cardelli ^{*} Marta Kwiatkowska [†] Matthew R. Lakin ^{*} David Parker [†]

Andrew Phillips^{*}

Abstract

Designing correct, robust DNA circuits is difficult because of the many possibilities for unwanted interference between molecules in the system. DNA strand displacement has been proposed as a design paradigm for DNA circuits and the DSD language as a means of formally expressing these circuits. We demonstrate the use of formal verification techniques, in particular model checking and probabilistic model checking, to analyse both the correctness and performance of such designs. We use the probabilistic model checker PRISM, in combination with DSD, to detect and fix an error in a simple strand displacement program, and to analyse the kinetics of the system. We also illustrate how similar techniques can be used on more complicated systems with potentially infinite chemical reaction networks, such as polymerizing systems.

1 Introduction

Molecular computing is a relatively new field that aims to construct information-processing circuits at the molecular level, using for example DNA. The technology has scope to be applied in a wide range of important application areas such as bio-sensing, biomimetic molecular manufacture and drug delivery. Designing correct and robust DNA circuits, however, is challenging. This is due, in particular, to the possibility of unwanted interference between molecules in the system. The DNA Strand Displacement (DSD) language [20] has been proposed as a means of formally expressing DNA circuits, building on the success of related languages to capture the dynamics of molecular networks [21, 19]. DSD is supported by software that facilitates the design and simulation of DNA circuits.

In this paper, we propose the use of *formal verification* techniques to check the correctness of and identify faulty behaviour in DNA circuit designs. We focus on *model checking*, a fully-automated approach to verification based on the exhaustive exploration of a finite-state model. We also employ *probabilistic model checking*, which generalises these techniques to the analysis of probabilistic models of systems that exhibit stochastic behaviour, for example due to failures or unpredictable components. Whereas conventional (non-probabilistic) model checking techniques can be used to check correctness properties such as “processes 1 and 2 never simultaneously access a shared resource”, probabilistic model checking allows verification of *quantitative* guarantees such as “the probability of an airbag failing to deploy within 0.02 seconds is at most 10^{-6} ”. Furthermore, probabilistic model checking can be used to evaluate a wide range of quantitative properties such as performance, e.g. “what is the expected time for a packet of data to be sent across the network?”. Probabilistic model checking has already been successfully applied to the analysis of systems from a wide range of application areas, from communication protocols like Bluetooth to quantum cryptography. In particular, it has also been used in the domain of systems biology to analyse cell signalling pathways [5, 13].

In the next two sections, we give a brief introduction to DNA strand displacement (DSD) and probabilistic model checking. We then describe the application of the probabilistic model checking tool PRISM [14] to the analysis of a simple DSD program from [6]. We detect and fix a flaw in this program, and then analyse the kinetics of the system. Finally, we discuss how these techniques can be extended to more complicated, potentially infinite-state, systems that use polymerization.

^{*}Microsoft Research, 7 JJ Thomson Avenue, Cambridge, CB3 0FB, UK

[†]Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

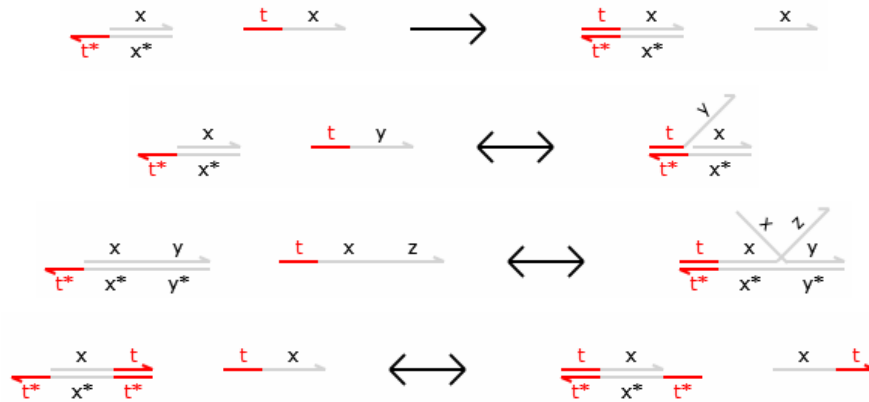


Figure 1: Toehold-mediated DNA branch migration and strand displacement

2 DNA strand displacement

Among the many techniques being developed for molecular computing [12], DNA strand displacement has been proposed as a mechanism for performing computation with DNA strands [23, 9]. In most schemes, single-stranded DNA acts as signals and double-stranded (or more complex) DNA structures act as gates. Various circuits have been demonstrated experimentally [25]. The strand displacement mechanism is appealing because it is autonomous [11]: once signals and gates are mixed together, computation proceeds on its own without further intervention until the gates or signals are depleted (output is often read by fluorescence). The energy for computation is provided by the gate structures themselves, which are turned into inactive waste in the process. Moreover, the mechanism requires only DNA molecules: no organic sources, enzymes, or transcription/translation machinery is required, and the whole apparatus can be chemically synthesized and run in standard laboratories.

The main aims of this approach are to harness computational mechanisms that can operate at the molecular level and produce nano-scale structures under program control, and somewhat separately that can intrinsically interface to biological entities [4]. The computational structures that one may easily implement this way (without some form of unbounded storage) vary from Boolean networks, to state machines, to Petri nets. The last two are particularly interesting because they take advantage of DNA’s ability to encode symbolic information: they operate on DNA strands that represent abstract signals.

The fundamental mechanism in many of these schemes is toehold mediated branch migration and strand displacement [25], which implements a basic step of computation. It operates as shown in Table 1, where each letter and corresponding segment represents a DNA domain (a sequence of nucleotides, C,G,T,A) and each DNA strand is seen as the concatenation of multiple domains. Single strands have an orientation; double strands are composed of two single strands with opposite orientation, where the bottom strand is the Watson-Crick, C-G, T-A, complement of the top strand. The “short” domains hybridize (bind) reversibly to their complements, while the “long” domains hybridize irreversibly; the exact critical length depends on physical conditions. Distinct letters indicate domains that do not hybridize with each other.

In the first reaction of Figure 1, a short toehold domain t initiates binding between a double strand and a single strand. After the (reversible) binding of the toehold, the “ x ” domain of the single strand gradually replaces the top “ x ” strand of the double strand by branch migration. The branching point between the two top “ x ” domains performs a random walk that eventually leads to displacing the “ x ” strand. The final detachment of the top “ x ” strand makes the whole process essentially irreversible, because there is no toehold for the reverse reaction. The second reaction illustrates the case where the top domains do not match: then the toehold binds reversibly and no displacement occurs. The third reaction illustrates the more detailed situation where the top domains match only initially: the branch migration can proceed only up to a certain point and then must revert back to the toehold: hence no displacement occurs and the whole reaction reverts.

The fourth reaction illustrates a toehold exchange, where a branch migration (of strand “ tx ”) leads to a displacement (of strand “ xt ”), but where the whole process is reversible via a reverse toehold binding and branch migration. The first (irreversible) and fourth (reversible) reactions are the fundamental steps that can be composed to achieve

computation by strand displacement.

2.1 The DSD language

The DSD language [20] provides a textual syntax for expressing the structure of DNA molecules such as those portrayed graphically in Figure 1. The semantics of the DSD language defines a formal translation of a collection of DNA molecules into a system of chemical reactions which captures the possible interactions between the strands of DNA. The DSD language includes syntactic and graphical abbreviations which allow us to represent a particular class of DNA molecules in a concise manner. The class of molecules in question is those without secondary structure – that is, only single-stranded DNA sequences may hang off the main double-stranded backbone of the molecule. This rules out complicated tree-like or pseudo-knotted structures, which greatly simplifies the definition of the semantics but still allows a wide variety of interesting systems to be written.

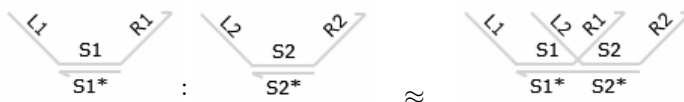
The syntax of the DNA strand displacement language is defined in terms of domain sequences S , L , R , gates G and molecules D . A domain sequence S is a concatenation of one or more domains, which can be long domains N or short domains N^\wedge . Each domain represents a nucleotide sequence, where short domains, also known as toeholds, are assumed to be between 4 and 10 nucleotides in length.

DNA molecules can be single or double stranded. A single upper strand $\langle S \rangle$ denotes a sequence of domains S oriented from left to right on the page, while a single lower strand $\{ S \}$ denotes a sequence S oriented from right to left on the page. A double strand $[S]$ denotes an upper strand $\langle S \rangle$ bound to the complement lower strand $\{ S^* \}$, where S^* denotes a sequence complementary to S . A gate G can be a double stranded molecule $\{ L' \} \langle L \rangle [S] \langle R \rangle \{ R' \}$ with overhanging single strands. This effectively represents an upper strand $\langle L S R \rangle$ bound to a lower strand $\{ L' S^* R' \}$ along the double-stranded region $[S]$. The sequences L , R , L' and R' can potentially be empty, in which case we simply omit them. A gate can also be a concatenation $G1 : G2$ of two gates $G1$ and $G2$ that share a common lower strand, or a concatenation $G1 : : G2$ of two gates that share a common upper strand.

A molecule D can be an upper strand $\langle S \rangle$, a lower strand $\{ S \}$ or a gate G . Multiple molecules $D1, D2$ can be present in parallel, written $D1 | D2$. A domain N can also be restricted to molecules D , written $\text{new } N D$. This represents the assumption that the domain is not used by any other molecules outside of D . We also allow module definitions of the form $X(m) = D$, where m are the module parameters and $X(n)$ is an instance of the module D with parameters m replaced by n . We assume a fixed set of module definitions, which are declared at the start of the program. The definitions are assumed to be non-recursive, such that a module cannot invoke itself, either directly or indirectly via another module.

We define the following conditions for well-formed molecules: (1) both a long domain and its complement are not unbound simultaneously; and (2) Both a sequence of two or more toeholds and its complement are not unbound simultaneously. This ensures that two single-stranded molecules can only interact with each other via complementary short domains. We assume that all molecules are well-formed. This textual syntax and the corresponding graphical representation are presented in Table 1.

If two gates are concatenated along the upper (or lower) strand, we typically omit the colon (or colons) in the graphical representation and simply connect the appropriate strands to form a single continuous strand. For example, we abbreviate the gate on the left below by the graphical abbreviation on the right.



If we fix a translation of domains into actual DNA sequences then we can view a more realistic representation of the above gate which shows the individual base pairs:

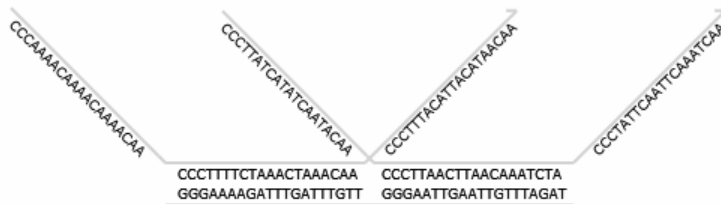




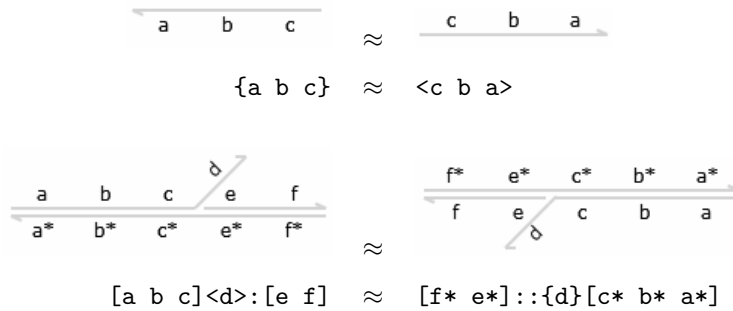


Table 1: Syntax of the DNA strand displacement language, in terms of gates G , molecules D and sequences S , L , R . For each construct, the graphical representation below is equivalent to the program code above, where applicable.

D	syntax	description
G	$\{L'\}\langle L\rangle[S]\langle R\rangle\{R'\}$	Double stranded molecule $[S]$ with overhanging single strands $\langle L\rangle$, $\langle R\rangle$ and $\{L'\}$, $\{R'\}$
		
	$G1:G2$	Molecules with shared lower strand
	$G1::G2$	Molecules with shared upper strand
D	$\{S\}$	Lower strand with sequence S
		
	$\langle S\rangle$	Upper strand with sequence S
		
	G	Gate G
	$D1 \mid D2$ $D1 \ D2$	Parallel molecules $D1$, $D2$
$\text{new } N \ D$	Molecules D with private domain N	
		
$X(n)$	Module X with parameters n	


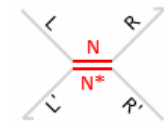
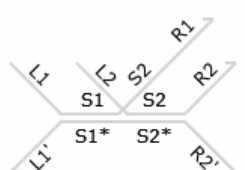

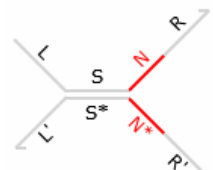
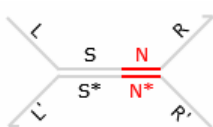
The distinction between “upper” and “lower” strands is simply an artifact of their representation on the page and has no basis in physical reality. Thus we identify DNA molecules in DSD up to rotation symmetry. For example, the following are considered to be equivalent:



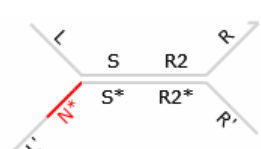
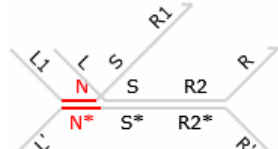
Note that when we rotate a gate, the domains in the double-stranded segments are complemented in the syntax. This is because we assume that the domain which appears in the syntax is the domain on the “upper” side of the double strand (as viewed on the page). We also consider gates to be equivalent up to migration of overhanging branches on double-stranded segments, as illustrated by the following structural congruence rule.

#	before	eq	after
EM	$\{L1'\} \langle L1 \rangle [S1] \langle S \ R1 \rangle :$ $\langle L2 \rangle [S \ S2] \langle R2 \rangle \{R2'\}$	\equiv	$\{L1'\} \langle L1 \rangle [S1 \ S] \langle R1 \rangle :$ $\langle L2 \ S \rangle [S2] \langle R2 \rangle \{R2'\}$

The following table illustrates some of the rules governing interactions between DNA molecules represented in the DSD language. These rules are used to define a translation from a collection of DNA molecules into a system of chemical reactions which can be used to simulate or analyse their behaviour. We present the rules as single-step reduction rules, where the double-headed arrow indicates a reversible reaction. The arrows are labelled with rate values which are used to parameterise an exponential rate distribution.

#	before	red	after
RB	$\langle L \ N^- \ R \rangle \mid \{L' \ N^- \ R'\}$ 	$\begin{matrix} \leftarrow N^+ \\ \leftarrow \\ N^- \end{matrix}$	$\{L'\} \langle L \rangle [N^-] \langle R \rangle \{R'\}$ 
RD	$\{L1'\} \langle L1 \rangle [S1] \langle S2 \ R1 \rangle : \\ \langle L2 \rangle [S2] \langle R2 \rangle \{R2'\}$ 	$\xrightarrow{S2^-}$	$\{L1'\} \langle L1 \rangle [S1 \ S2] \langle R1 \rangle \{R2'\} \mid \\ \langle L2 \ S2 \ R2 \rangle$ 
RC	$\{L'\} \langle L \rangle [S] \langle N^- \ R \rangle \{N^* \ R'\}$ 	$\xrightarrow{N^-}$	$\{L'\} \langle L \rangle [S \ N^-] \langle R \rangle \{R'\}$ 

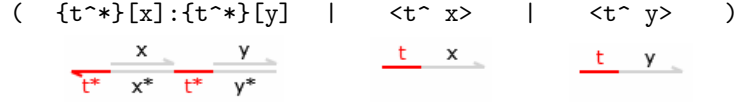
In order to improve efficiency and reduce the size of the resulting chemical reaction networks we may ignore reactions where a strand binds onto a gate but cannot perform any subsequent reaction other than an unbinding. In this case we replace the RB rule from the previous table with the following binding rule.

#	before	red	after
PM	$\langle L1 \ N^- \ S \ R1 \rangle \mid \\ \{L' \ N^* \} \langle L \rangle [S \ R2] \langle R \rangle \{R'\}$ 	$\xrightarrow{N^+}$	$\{L'\} \langle L1 \rangle [N^-] \langle S \ R1 \rangle : \\ \langle L \rangle [S \ R2] \langle R \rangle \{R'\}$ 

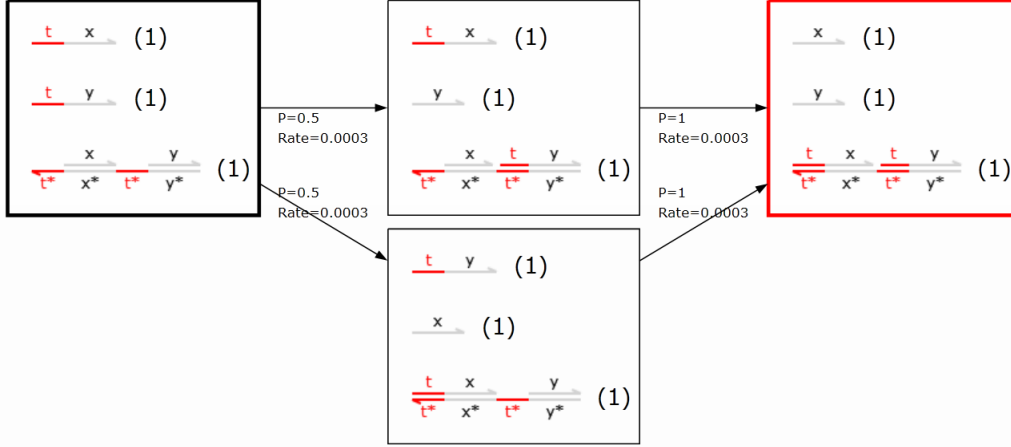
2.2 The Visual DSD tool

The Visual DSD tool is an implementation of the DSD language. It can be run either in a web browser or from the command line. As well as implementing the compilation of DNA molecules into chemical reactions, the tool includes other features such as stochastic and deterministic simulation, visualisation of the chemical reaction network and SBML export of the model. In this paper, we develop additional functionality based on an exhaustive enumeration of the *state space* of a DSD system. A *state* is a possible configuration of the system, i.e. a multiset of species where the multiplicity of a particular species is referred to as its *population*. The state space is a graph of all possible transitions between states, using reactions generated from the semantics of the DSD language. This contrasts with stochastic simulation, where only a single path through the state graph is explored. Furthermore, we can annotate each transition between states with the *rate* at which it occurs, for which values can be determined experimentally. More precisely (see [10]) the delay before the occurrence of the transition can be assumed to be modelled by an exponential distribution, making the resulting annotated model a continuous-time Markov chain.

We illustrate these concepts with the following simple example program that includes a DNA gate and two strands which may bind onto it.



The binding reactions of the $\langle t^* x \rangle$ and $\langle t^* y \rangle$ strands onto the gate could happen in either order and produce the same end result, which is illustrated by the fact that there are two (equally probable) paths through the state graph shown below.



If we were to run a stochastic simulation the simulator would randomly choose to explore one path or the other, whereas by enumerating the entire state space we essentially explore all possible paths. This means that state space enumeration allows for more kinds of analysis compared to a simple stochastic simulation, although the downside is that it can be very computationally expensive as the state spaces can grow very rapidly as species are added to the system.

3 Probabilistic Model Checking and PRISM

Model checking is an automated formal verification technique, based on the exhaustive construction and analysis of a finite-state model of the system being verified. The model is usually a labelled state-transition system, in which each state represents a possible configuration of the system and each transition between states represents a possible evolution from one configuration to another. The desired correctness properties of the system are typically expressed in temporal logics, such as CTL (Computation Tree Logic) or LTL (Linear-time Temporal Logic). We omit here a precise description of these logics (see, for example, [8, 3] for detailed coverage); instead we give below some typical CTL formulae, along with their corresponding informal meanings:

- $\forall \square \neg(\text{access}_1 \wedge \text{access}_2)$ - “processes 1 and 2 never simultaneously access a shared resource”;
- $\forall \diamond \text{end}$ - “the algorithm always eventually terminates”
- $\exists (\neg \text{fail} \text{ U } \text{end})$ - “it is possible for the algorithm to terminate without any failures occurring”

Once the desired correctness properties of the system have been formally expressed in this way, they can then be verified using a *model checker*. This performs an exhaustive analysis of the system model, for each property either concluding that it is satisfied or, if not, providing a counterexample illustrating why it is violated.

Probabilistic model checking is a variant of model checking for the verification of systems that exhibit stochastic behaviour. In this case, the models that are constructed and analysed are augmented with quantitative information regarding the likelihood that transitions occur and the times at which they do so. In practice, these models are typically Markov chains or Markov decision processes. To model systems of reactions at a molecular level, the appropriate model is *continuous-time Markov chains* (CTMCs), in which transitions between states are assigned (positive, real-valued) rates. These values are interpreted as the rates of negative exponential distributions.

Formally, letting $\mathbb{R}_{\geq 0}$ denote the set of non-negative reals and AP be a fixed, finite set of atomic propositions used to label states with properties of interest, a CTMC is a tuple (S, \mathbf{R}, L) where:

- S is a finite set of *states*;
- $\mathbf{R} : (S \times S) \rightarrow \mathbb{R}_{\geq 0}$ is a *transition rate matrix*;
- $L : S \rightarrow 2^{AP}$ is a *labelling* function which associates each state with a set of atomic propositions.

The transition rate matrix \mathbf{R} assigns rates to each pair of states, which are used as parameters of the exponential distribution. A transition can only occur between states s and s' if $\mathbf{R}(s, s') > 0$ and, in this case, the probability of the transition being triggered within t time-units is $1 - e^{-\mathbf{R}(s, s') \cdot t}$. Typically, in a state s , there is more than one state s' for which $\mathbf{R}(s, s') > 0$; this is known as a *race condition* and the first transition to be triggered determines the next state. The time spent in state s before any such transition occurs is exponentially distributed with the rate $E(s) = \sum_{s' \in S} \mathbf{R}(s, s')$, called the *exit rate*. The probability of moving to state s' is given by $\mathbf{R}(s, s')/E(s)$.

A CTMC can be augmented with *rewards*, attached to states and/or transitions of the model. Formally, a *reward structure* for a CTMC is a pair (c, C) where:

- $c : S \rightarrow \mathbb{R}_{\geq 0}$ is a *state reward function*;
- $C : (S \times S) \rightarrow \mathbb{R}_{\geq 0}$ is a *transition reward function*.

State rewards can represent either a quantitative measure of interest at a particular time instant (e.g. the number of phosphorylated proteins in the system) or the rate at which some measure accumulates over time (e.g. energy dissipation). Transition rewards are accumulated each time a transition occurs and can be used to compute, e.g. the number of protein bindings over a particular time period.

Properties of CTMCs are, like in non-probabilistic model checking, expressed in temporal logic, but are now quantitative in nature. For this, we use probabilistic temporal logics such as CSL [1, 2] and its extensions for reward-based properties [17]. For example, rather than verifying that ‘the protein always eventually degrades’, using CSL allows us to ask ‘what is the probability that the protein eventually degrades?’ or ‘what is the probability that the protein degrades within t hours?’. Reward-based properties include ‘what is the expected time that proteins are bound within the first t time units?’ and ‘what is the expected number of phosphorylations before relocation occurs?’. For further details on probabilistic model checking of CTMCs, see for example [2, 17].

PRISM [14] is a probabilistic model checking tool developed at the Universities of Birmingham and Oxford. It provides support for several types of probabilistic models, including CTMCs. Models are specified in a simple, state-based language based on guarded commands. Support for several other high-level model description languages has been made available through language-level translations to the PRISM modelling language. Of particular relevance here is PRISM’s ability to import SBML [15] specifications, which have an underlying CTMC semantics. Translations from stochastic process algebra such as PEPA and the stochastic π -calculus [21] have also been developed.

PRISM can then be used to specify and verify a range of properties of CTMCs, including those expressed in the logic CSL and the reward-based extension of [17]. The underlying computation performed to apply probabilistic model checking involves a combination of:

- *graph-theoretical algorithms*, for conventional temporal logic model checking and *qualitative* probabilistic model checking;
- *numerical computation*, for *quantitative* probabilistic model checking, i.e. calculation of probabilities and reward values.

Graph-theoretical algorithms are comparable to the operation of a conventional, non-probabilistic model checker. For numerical computation, PRISM typically solves linear equation systems or performs transient analysis. Due to the size of the models that need to be handled, the tool uses iterative methods rather than direct methods. For solution of linear equation systems, it supports a range of well-known techniques including the Jacobi, Gauss-Seidel and SOR (successive over-relaxation) methods; for transient analysis of CTMCs, it employs uniformisation.

One of the most notable features of PRISM is that it uses state-of-the-art *symbolic* approaches, using data structures based on binary decision diagrams [16]. These allow for compact representation and efficient manipulation of large, structured models by exploiting regularities exhibited in the high-level modelling language descriptions. The tool actually provides three distinct *engines* for numerical solution: the first is purely symbolic; the second uses

sparse matrices; and the third is a hybrid, using a combination of the two. The result is a flexible implementation which can be adjusted to improve performance depending on the type of models and properties being analysed.

PRISM also incorporates a discrete-event simulation engine. This allows approximate solutions to be generated for the numerical computations that underlie the model checking process, by applying Monte Carlo methods and sampling. These techniques offer increased scalability, at the expense of numerical accuracy. Using the same underlying engine, PRISM includes a tool to perform manual execution and debugging of probabilistic models. Other functionality provided by the user interface of the tool includes a graph-plotting component for visualisation of numerical results and editors for the model and property specification languages.

4 Model Checking DSD Programs

In this section, we illustrate the use of model checking to detect design errors in and evaluate the performance of DNA strand displacement (DSD) programs. We also discuss an extension to a system that can construct DNA polymers.

4.1 Two-domain transducers

Our case study involves the *two-domain* gate scheme proposed by Cardelli in [6], where implementations for transducers, join gates and fork gates are given in a subset of the DSD language [20]. The two-domain scheme includes only simple strands consisting of a short toehold domain and a longer recognition domain and simple gates without overhanging strands. These restrictions are useful in practice for a number of reasons: the gate designs only require a single toehold, the molecules are simple to construct and the lack of overhangs means that the kinetics of the interactions are easier to predict. The following figure after [6] shows some DNA molecules which can be constructed under this restricted syntax.



We will concern ourselves with the simplest circuit from [6]: the signal transducer. A transducer T_{xy} turns a signal strand $\langle t^{\wedge} x \rangle$ into a signal strand $\langle t^{\wedge} y \rangle$. A chemical reaction network for this program is shown in Figure 2. The graph was constructed using the Visual DSD tool following assumptions from [26] about the kinetics of toehold binding, unbinding and strand displacement reactions. We assume that toehold binding is sufficiently slow relative to branch migration, strand displacement and toehold unbinding, such that toehold binding has a finite rate while the other reactions happen instantaneously.

4.2 A faulty design: Serial transducers

We begin by analysing a DSD program for the serial transducer “ $T_{xy} \mid T_{yz} \mid \langle t^{\wedge} x \rangle$ ”, which should translate the input signal $\langle t^{\wedge} x \rangle$ into the intermediate signal $\langle t^{\wedge} y \rangle$ and then translate this into the output signal $\langle t^{\wedge} z \rangle$. The DSD script for this program is as follows.

```

new t
new a
def T(N,x,y) =
  ( N *  $\langle t^{\wedge} a \rangle$ 
  | N *  $\langle y t^{\wedge} \rangle$ 
  | N *  $t^{\wedge} * : [x t^{\wedge}] : [a t^{\wedge}] : [a]$  (* Input gate *)
  | N *  $[x] : [t^{\wedge} y] : [t^{\wedge} a] : t^{\wedge} *$  (* Output gate *) )
(  $\langle t^{\wedge} x \rangle \mid T(1,x,y) \mid T(1,y,z)$  )

```

This simple script defines a domain t which is used throughout as a toehold domain and a domain a which is a long recognition domain. The module $T(N,x,y)$ expands out to give N copies of the signal transducer gate T_{xy} . Finally the initial configuration of the system is specified, which includes one copy of T_{xy} and one copy of T_{yz} along with a single input signal $\langle t^{\wedge} x \rangle$.

We analysed this program by enumerating the reaction network using Visual DSD, exporting it to SBML and then importing this to the PRISM probabilistic model checker for analysis. PRISM is able to exhaustively explore the

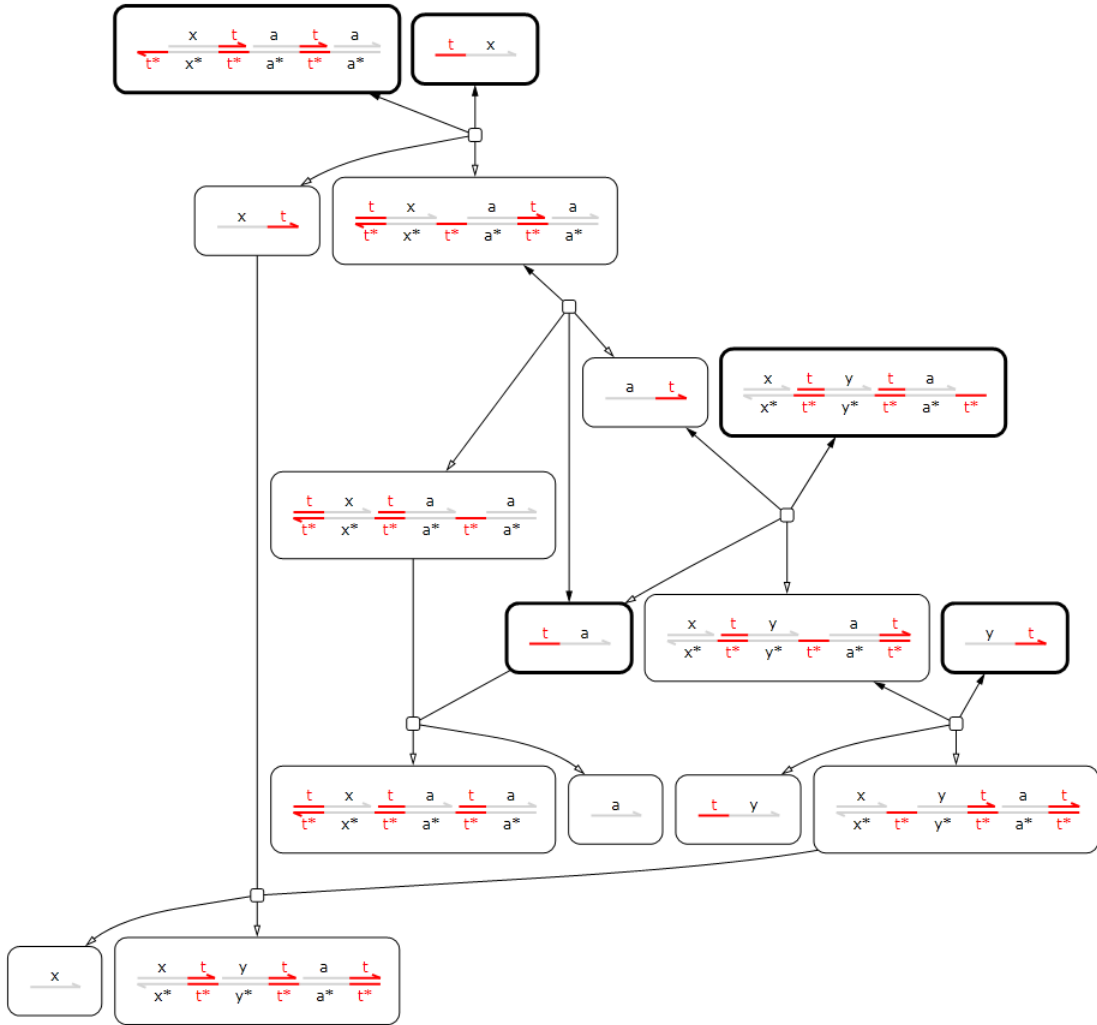


Figure 2: Chemical reaction network for a single two-domain transducer

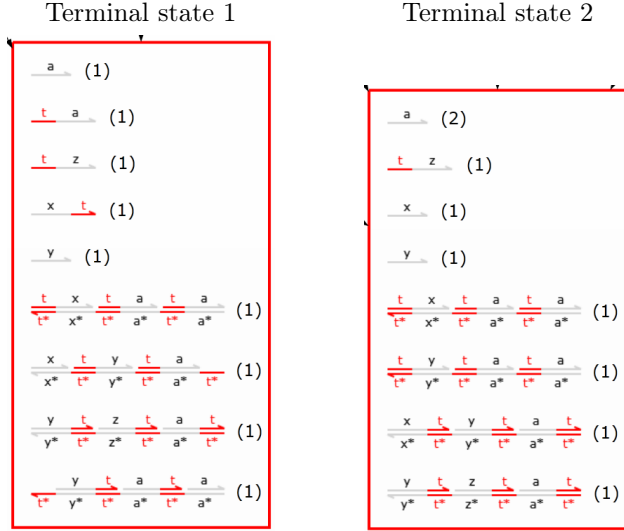
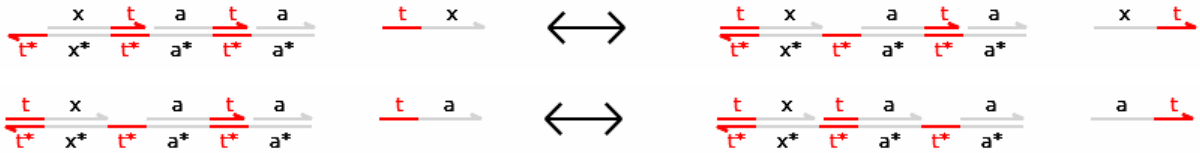


Figure 3: Terminal states for the first transducer design

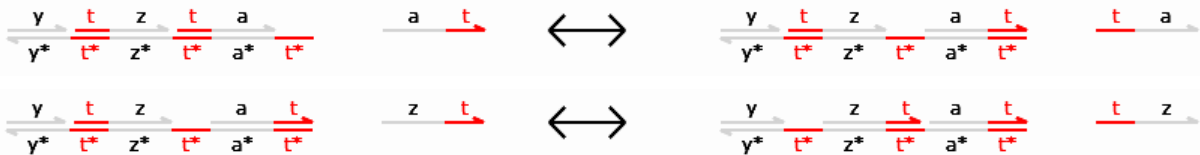
corresponding CTMC model that captures all possible behaviour of the system. For this relatively small program, there are 56 states and 147 transitions. One way to check the correctness of the serial transducer program is to verify that it is always possible to get to a configuration in which the output signal $\langle \overline{t} \overline{z} \rangle$ is present, and that in all cases there is an intermediate state in which $\langle \overline{t} \overline{y} \rangle$ is present. We can express this with the CTL properties $\forall \Diamond tz$ and $\neg(\exists(-tyUtz))$ where ty and tz are atomic propositions labelling states of the model in which the populations of the signals $\langle \overline{t} \overline{y} \rangle$ and $\langle \overline{t} \overline{z} \rangle$, respectively, are 1.

PRISM can be used to check the validity of such properties. In this case, we find that the first property is true, but the second is violated. Investigation shows that the model includes two *terminal* (or “deadlock”) states, i.e. states from which no further transitions are possible. The populations of the various molecular species in these two terminal states are illustrated in Figure 3.

In this case, terminal state 2 is the result that we would anticipate: this state contains the output strand $\langle \overline{t} \overline{z} \rangle$ along with the inert garbage left over from correct execution of the two transducers. However, terminal state 1 is incorrect – even though the output strand $\langle \overline{t} \overline{z} \rangle$ is produced we see that some constituent molecules of the transducers are left unused and with exposed toehold domains (i.e. they are not inert). Model checking the second property also produces a counterexample, i.e. a trace through the model that illustrates why the property fails. In this case, the trace ends in terminal state 1. The first few reactions proceed as one would expect.



The problem arises because the $\langle \overline{a} \overline{t} \rangle$ strand can now interact with molecules from the T_{yz} transducer, causing the following reactions.



These reactions produce the output strand $\langle \overline{z} \overline{t} \rangle$. There are some subsequent reactions which tidy up as many as possible of the molecules with exposed toeholds, but there are some non-inert molecules left. Thus the problem is

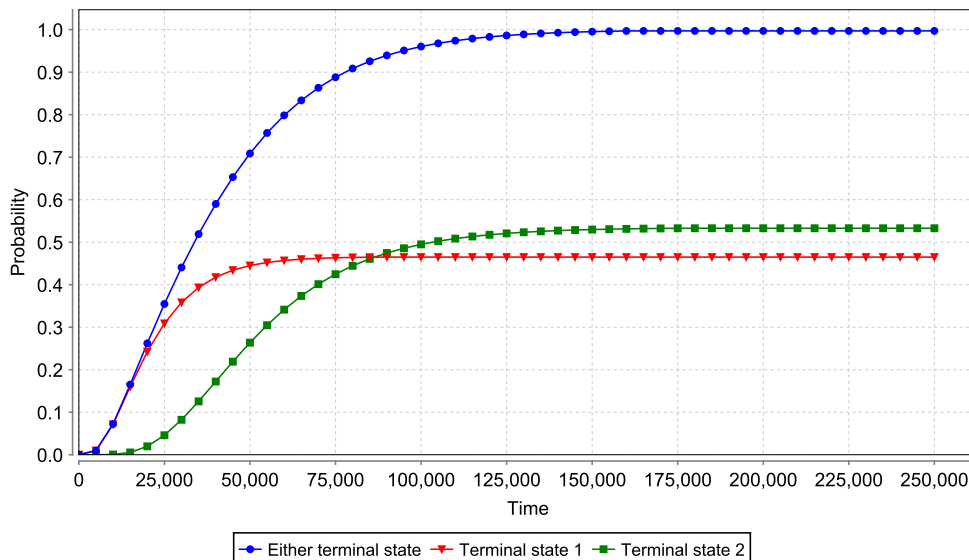


Figure 4: Termination probabilities over time for the first transducer design

that the $\langle a \tau \rangle$ strand can prematurely activate the T_{yz} transducer to produce $\langle \tau z \rangle$, skipping the intermediate step of producing the $\langle \tau y \rangle$ strand and leaving parts of the transducers unused. This problem arises because the two transducers share a common recognition domain a which allows them to interfere with each other. The possibility of such errors in transducer implementations was noted by Cardelli [6] where it was also noted that state-space analysis is required to prove that correct termination is guaranteed, even for simple examples like those we consider here.

PRISM can also evaluate the likelihood that such an error will occur. Figure 4 shows the probability of ending up in each terminal state against time. We note that as time increases, the probability that the program terminates tends to 1. This reflects the fact that the only loops in the program are from reversible reactions – the presence of irreversible garbage collection reactions means that the system tends to be pulled towards a terminating state. When time is small the system is more likely to terminate in the error state (terminal state 1) than the correct state (terminal state 2). This reflects the fact that the path to terminal state 1 is much shorter than the path to terminal state 2, because the incorrect execution omits some of the transducer steps. When time is large this is reversed and the system is slightly more likely to terminate in the correct state. To four decimal places, the overall probabilities of reaching a particular terminal state are as follows, along with the PRISM queries which were used to compute them:

$$\begin{aligned}
 P(\text{correct termination}) &= 0.5343 & (P=?[F \text{ “correct_deadlock”}]) \\
 P(\text{error termination}) &= 0.4656 & (P=?[F \text{ “error_deadlock”}])
 \end{aligned}$$

where “correct_deadlock” and “error_deadlock” are PRISM labels (i.e. atomic formulae) which pick out terminal state 2 and terminal state 1 respectively. Thus our program is only slightly more likely to execute correctly than to execute incorrectly.

If we run multiple transducers and input strands in parallel, however, an interesting effect occurs. With n copies of T_{xy} , n copies of T_{yz} and n input strands $\langle \tau x \rangle$, the probability of terminating in the correct state increases as n increases. This happens because the other copies of the transducers can help to “unblock” a transducer that has taken the wrong path. Figure 5 illustrates this trend as n increases up to 4. For larger values of n , the models become too large to analyse. The probability of landing in an “error” state includes any terminal state which is not the correct one (when $n > 2$ there are multiple terminal states which are incorrect). This suggests that, if the population of transducers is large enough, we can simplify the construction of the system by sharing a single toehold between different transducer circuits and still have a high probability of producing the correct behaviour.

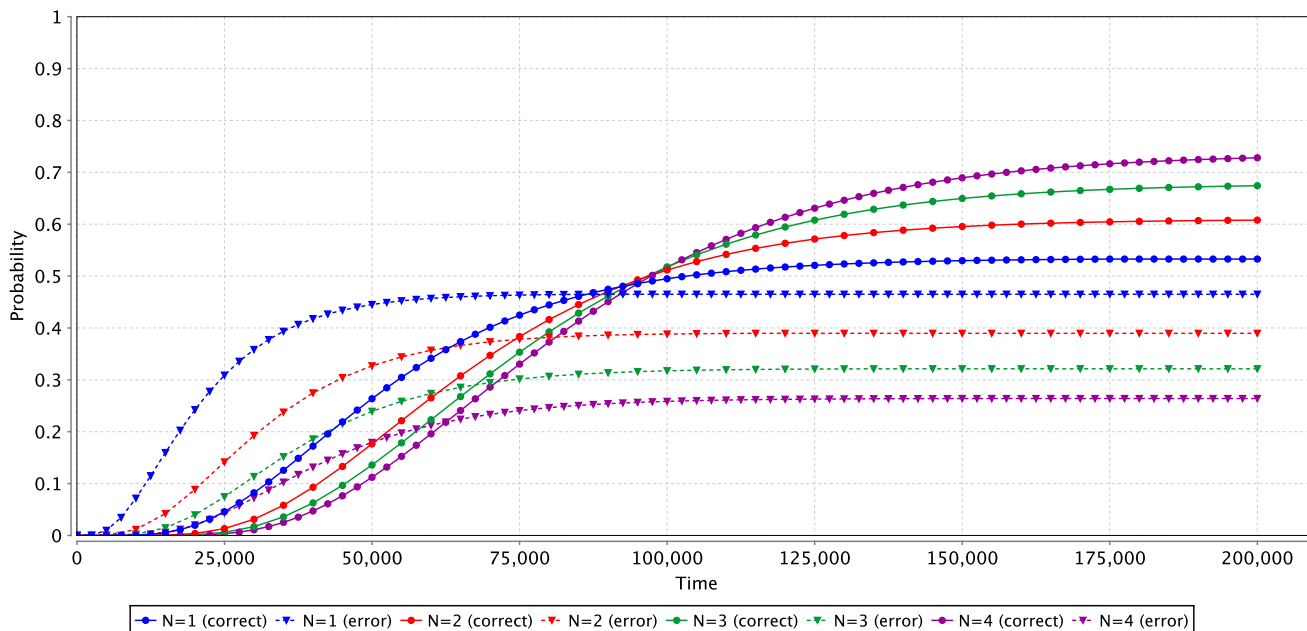


Figure 5: Termination probabilities for multiple parallel instances of our first transducer design

4.3 A repaired serial transducer design

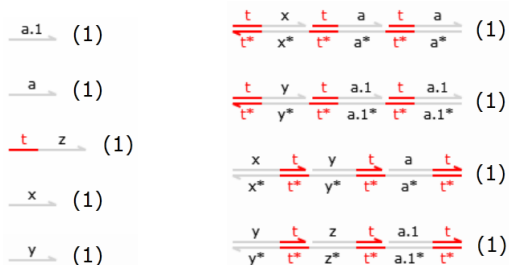
As mentioned above, the problem with our program is that the T_{xy} and T_{yz} transducers both use the recognition domain a which allows unwanted cross-talk between the two transducers. We can fix this bug by moving the declaration of a into the $T(N, x, y)$ module, producing the following DSD script.

```

new t
def T(N, x, y) =
  new a
  ( N * <t^ a>
  | N * <y t^>
  | N * t^*: [x t^]: [a t^]: [a] (* Input gate *)
  | N * [x]: [t^ y]: [t^ a]: t^* (* Output gate *) )
( <t^ x> | T(1, x, y) | T(1, y, z) )

```

In this program, the `new a` declaration will be evaluated once for T_{xy} and once for T_{yz} , producing two different recognition domains which DSD tags as a and $a.1$. Thus, the `new a` declaration is a means of specifying which DNA domains need to be distinct when multiple circuits are created. This suffices to prevent cross-talk – the state space for this program contains 107 transitions between 41 states and just one terminal state. The populations of the species in the terminal state are as shown below. These are as one would expect from serialised executions of the two transducers – there is a single output strand $\langle t^ z \rangle$ along with the inert garbage from two transducer gates.



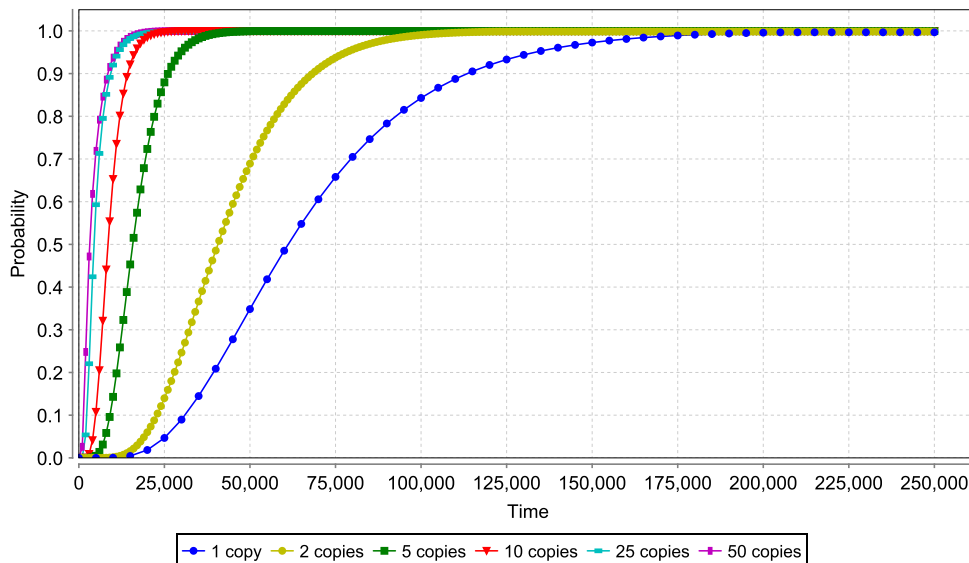


Figure 6: Termination probabilities over time for the second transducer design, with a single input strand and multiple copies of the transducer.

Having verified that our modified program is executing correctly, we can again use probabilistic model checking to explore the kinetics of the system. Figure 6 shows the effect of adding additional copies of the modified T_{xy} and T_{yz} transducer circuits, though still with only a single input strand $\langle \tau^x \rangle$. Note that it is perfectly acceptable for multiple copies of a particular transducer to share a single recognition domain, because they cannot cross-talk in a way which causes the system to behave incorrectly. This is reflected in the fact that there is still only one terminal state, irrespective of how many additional copies are added. As expected, increasing the population of the transducer circuits reduces the expected time to reach a terminal state. Thus, adjusting the number of copies of the transducer circuit can be used to fine-tune its response time. The plot clearly illustrates diminishing returns as the population of the transducer circuit increases, due to the reversible nature of the reactions. The plot also illustrates a more general point: that it is still possible to analyse subsystems with large numbers of molecules, provided the number of parallel execution paths is reasonably small. The fact that there is only a single input strand significantly limits the parallel nature of the system. However, when we consider systems with both multiple input strands and multiple transducer circuits in parallel, the state space becomes intractable after only 4 copies of the input. Thus, significant challenges remain before we can effectively analyse practical systems.

If we increase the populations of both the modified transducer circuit *and* input strands, as described in Figure 5, we find that the system still only has one terminal state. This reflects the fact that the transducer circuits cannot cross-talk because each transducer circuit has a unique recognition domain. The size of the state graph increases to 593 states (2615 transitions) when $n = 2$ and 4870 states (28341 transitions) when $n = 3$, which means we are still faced with a combinatorial explosion in the number of states as the number of circuits and inputs is increased.

4.4 Hairpin-free hybridization chain reaction

The approach to state-space analysis described above relies on the fact that the chemical reaction network for the system is finite and can hence be computed in its entirety. However, many systems of interest do not have finite chemical reaction networks and therefore cannot be analysed using this technique. These include implementations of Turing-powerful computational models such as stack machines [22] and other systems which can assemble DNA polymers [7]. In this section we describe work on an alternative modelling strategy which allows us to analyse the state graph corresponding to the reachable subset of a potentially infinite chemical reaction network.

As an example we use a hairpin-free variant of hybridization chain reactions (HCR). This is timely as HCR was recently used as a mechanism to trigger cell death when cancer markers are present [24]. A DSD script for a simple example of three monomers assembling to form a three-unit polymer is as follows.

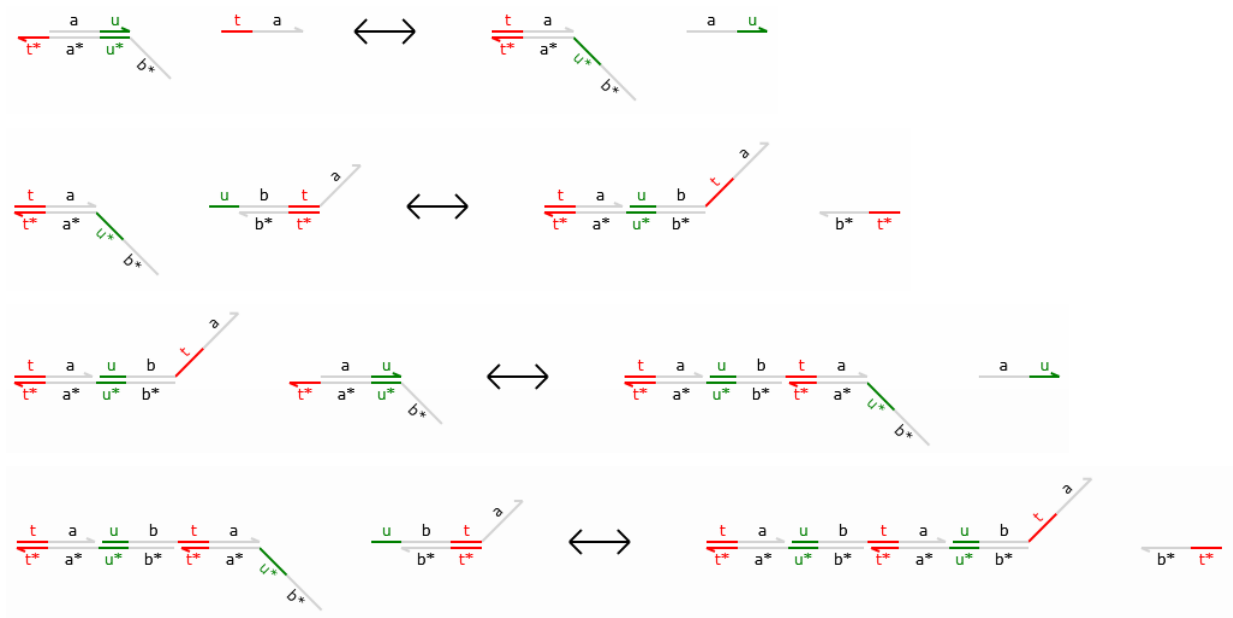


Figure 7: Example reactions from the hairpin-free HCR scheme

```

def XS = 1000
def N = 3
( <t~ a>          (* Initiator *)
| N * {t~*}[a u~]{b*} (* Monomer 1 *)
| XS * [a]{u~*}      (* Collector 1 *)
| XS * [b]<t~>      (* Collector 2 *)
| N * <u~>[b t~]<a>  (* Monomer 2 *) )

```

The formation of the polymer chain consists of a series of simple strand displacement and polymerization reactions as shown in Figure 7, with additional irreversible reactions where the “collectors” perform garbage collection to keep the reaction moving in the forward direction. Since the chemical reaction network is infinite we cannot directly export it to SBML as above. In the general case, in order to determine the set of reactions that are actually possible it is necessary to explore the state space to some degree. This effectively rules out generating the state space from the set of reactions. We have solved this problem by enumerating the reachable state space of the system within the Visual DSD tool itself. By taking the populations of species into account we construct the reachable state space incrementally, using a just-in-time (JIT) compilation approach similar to that described in [18]. This produces a state space with 50 states and 133 transitions, and a single terminal state. The initial and terminal states of the system are shown in Figure 8.

As we can see, the sets of three monomer units and the single initiator strand $\langle t^{\sim} a \rangle$ have been used up and a three-unit polymer has been produced. Constructing the state space within DSD in this way is a very attractive means of analysing systems such as hairpin-free HCR and stack machines. Some straightforward programming work is needed to enable tighter integration between the state space graph produced in DSD and the PRISM model checker, to allow these models to be verified and quantitatively analysed. The ability to tag particular states of interest within DSD so that PRISM can immediately recognise them would also be very useful.

5 Future work

We hope to extend these techniques to larger, more complex examples. In particular, probabilistic model checking may provide a way to quantify the effect of so-called “leak” reactions on the dynamic behaviour of systems. Leaks are interactions which are not mediated by a toehold – instead, the bonds holding a long double-stranded recognition domain together begin to fray at the edges, allowing another strand to move in and displace the recognition domain.

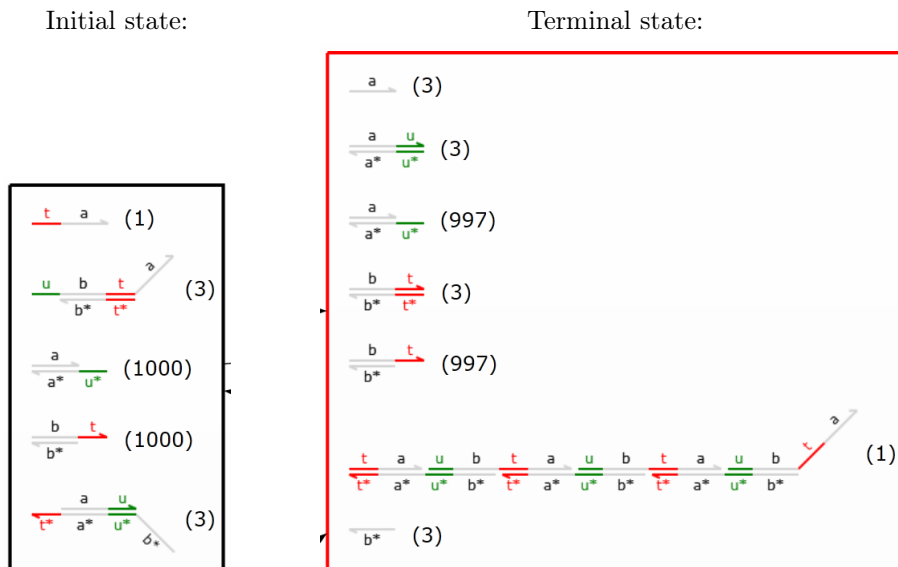


Figure 8: Initial and terminal states of the three-monomer hairpin-free HCR system

As such, the rates of leak reactions are very low but if a leak reaction does take place it may dramatically alter the trajectory of the system. Hence it would be helpful to verify that the probability of a leak reaction taking place is below some threshold, for example.

6 Conclusion

We have used probabilistic model checking to detect an error in a DSD program, verify that the modified version is correct and examine how the dynamics of the system changes as the populations of transducers increase. We have also motivated language-integrated exploration of the state-space graph for systems where the network of all possible chemical reactions would be infinite.

References

- [1] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking continuous time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.
- [2] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.
- [3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
- [4] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, and E. Shapiro. Programmable and autonomous computing machine made of biomolecules. *Nature*, 414(22), 2001.
- [5] M. Calder, V. Vyshemirsky, D. Gilbert, and R. Orton. Analysis of signalling pathways using continuous time Markov chains. *Transactions on Computational Systems Biology VI*, 4220:44–67, 2006.
- [6] L. Cardelli. Two-domain DNA strand displacement. In *DCM*, 2010.
- [7] L. Cardelli and G. Zavattaro. Turing universality of the biochemical ground form. *Mathematical Structures in Computer Science*, 20(1):45–73, 2010.
- [8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

- [9] W. Fontana. Pulling strings. *Science*, 314(8), 2006.
- [10] D. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.
- [11] S. J. Green, D. Lubrich, and A. J. Turberfield. DNA hairpins: fuel for autonomous DNA devices. *Biophysical Journal*, 91, 2006.
- [12] M. Hagiya. Towards molecular programming. In G. Ciobanu and G. Rozenberg, editors, *Modelling in molecular biology*. Springer, 2004.
- [13] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science*, 319(3):239–257, 2008.
- [14] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’06)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.
- [15] Michael Hucka, Andrew Finney, Herbert M. Sauro, Hamid Bolouri, John C. Doyle, Hiroaki Kitano, Adam P. Arkin, Benjamin J. Bornstein, Dennis Bray, Athel Cornish-Bowden, Autumn A. Cuellar, Sergey Dronov, Ernst Dieter Gilles, Martin Ginkel, Victoria Gor, Igor I. Goryanin, Warren J. Hedley, T. Charles Hodgman, Jan-Hendrik S. Hofmeyr, Peter J. Hunter, Nick S. Juty, Jay L. Kasberger, Andreas Kremling, Ursula Kummer, Nicolas Le Novère, Leslie M. Loew, Daniel Lucio, Pedro Mendes, Eric Minch, Eric D. Mjolsness, Yoichi Nakayama, Melanie R. Nelson, Poul F. Nielson, Takeshi Sakurada, James C. Schaff, Bruce E. Shapiro, Thomas S. Shimizu, Hugh D. Spence, Jörg Stelling, Kouichi Takahashi, Masaru Tomita, John M. Wagner, Jian Wang, and the rest of the SBML forum. The Systems Biology Markup Language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 9(4):524–531, 2003.
- [16] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004.
- [17] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM’07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.
- [18] L. Paulevé, S. Yousseff, M. R. Lakin, and A. Phillips. A generic abstract machine for stochastic process calculi. In *Computational Methods in Systems Biology*, 2010.
- [19] Andrew Phillips and Luca Cardelli. Efficient, correct simulation of biological processes in the stochastic pi-calculus. In *Computational Methods in Systems Biology*, volume 4695 of *LNCS*, pages 184–199. Springer, September 2007.
- [20] Andrew Phillips and Luca Cardelli. A programming language for composable DNA circuits. *J R Soc Interface*, 6 Suppl 4:S419–S436, Aug 2009.
- [21] C. Priami, A. Regev, E. Shapiro, and W. Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80(1):25–31, 2001.
- [22] L. Qian, D. Soloveichik, and E. Winfree. Efficient Turing-universal computation with DNA polymers. In *DNA16*, 2010.
- [23] G. Seelig, D. Soloveichik, D. Y. Zhang, and E. Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314(8), 2006.
- [24] S. Venkataraman, R. M. Dirks, C. T. Ueda, and N. A. Pierce. Selective cell death mediated by small conditional RNAs. *Proceedings of the National Academy of Sciences*, 2010.
- [25] B. Yurke and A. P. Mills Jr. Using DNA to power nanostructures. *Genetic Programming and Evolvable Machines Archive*, 4(2):111–122, 2006.

- [26] D. Y. Zhang and E. Winfree. Control of DNA strand displacement kinetics using toehold exchange. *Journal of the American Chemical Society*, 131:17303–17314, 2009.