Natural Computing (2018) 17:161–182
https://doi.org/10.1007/s11047-018-9671-4

Petri-net-based 2D design of DNA walker circuits

David Gilbert[1] · Monika Heiner[2] · Christian Rohr[2]

Published online: 28 February 2018

Abstract
We consider localised DNA computation, where a DNA strand walks along a binary decision graph to compute a binary function. One of the challenges for the design of reliable walker circuits consists in leakage transitions, which occur when a walker jumps into another branch of the decision graph. We automatically identify leakage transitions, which allows for a detailed qualitative and quantitative assessment of circuit designs, design comparison, and design optimisation. The ability to identify leakage transitions is an important step in the process of optimising DNA circuit layouts where the aim is to minimise the computational error inherent in a circuit while minimising the area of the circuit. Our 2D modelling approach of DNA walker circuits relies on coloured stochastic Petri nets which enable functionality, topology and dimensionality all to be integrated in one two-dimensional model. Our modelling and analysis approach can be easily extended to 3-dimensional walker systems.

Keywords Stochastic Petri nets · Coloured Petri nets · DNA walker systems · Design assessment · Leakage transitions · Structural analysis · Qualitative analysis · Stochastic analysis · Simulative model checking

1 Introduction

DNA computing building on DNA strands (molecules) interacting by DNA strand displacement (DSD) is a research focus in computer science and nanomedicine alike (Boemo et al. 2016). DSD can be thought of as a formal computing language (Phillips and Cardelli 2009) for the engineering of DNA-only chemical controllers, sensors, etc. (Chen et al. 2013). Two DSD categories can be distinguished (Boemo et al. 2016). (1) In *floating DNA systems*, DNA strands are freely moving molecules in a well-mixed solution; i.e., there are no geometric constraints preventing two molecules from interacting. (2) *Localised DNA systems* impose constraints by tethering DNA strands (anchorages) to a rigid lattice, forming a DSD circuit. An additional DNA strand (walker) may move along the lattice organised in origami tiles, thus performing a computation, e.g., by walking along a binary decision tree, possibly reduced to a directed acyclic graph (DAG), yielding a *binary decision DAG*, in the following briefly called DAG. Different options for programming a given DSD circuit are known to force a walker to follow a specific path (Boemo et al. 2016).

There are a couple of challenges for the design of reliable DSD circuits. DSD circuits are inherently undirected, and thus do not directly encode DAGs. A walker may take a shortcut or even jump into another path; the latter is known as a leakage transition. Therefore, the experimental design of DSD circuits clearly calls for tool support.

Floating systems are supported by the Microsoft Visual DSD tool (Lakin et al. 2011), while localised systems are considered in Dannenberg et al. (2015) and Barbot and Kwiatkowska (2015); none supports the automated identification of leakage transitions. Modelling DNA computing devices with freely moving molecules closely resembles modelling approaches for chemical reaction networks as they are widely used in systems and synthetic biology; e.g., we could deploy Petri nets as umbrella language opening the doors to qualitative, stochastic and deterministic

✉ David Gilbert
David.Gilbert@brunel.ac.uk

Monika Heiner
Monika.Heiner@b-tu.de

Christian Rohr
Christian.Rohr@b-tu.de

[1] Brunel University London, Uxbridge UB8 3PH, UK

[2] Brandenburg Technical University Cottbus-Senftenberg, Postbox 10 13 44, 03013 Cottbus, Germany

analysis techniques, as we have previously demonstrated in Gilbert et al. (2007), Heiner et al. (2008) and Blätke et al. (2015).

1.1 Contributions

In this paper we consider localised DNA computation. We start from the modelling approach for walker circuits introduced in Dannenberg et al. (2015) and represented as stochastic Petri nets in Barbot and Kwiatkowska (2015), with the purpose of stochastic analysis to assess the reliability of the circuit design. To assist the circuit designer by a more detailed assessment, we refine the stochastic analysis, complemented by merely qualitative, and thus computationally less expensive analyses. More specifically we discuss the automated identification of leakage transitions and how to quantitatively compare different circuit designs for a given DAG. Leakage can be reduced by employing a circuit layout topology that optimises the distance between any two anchorages to avoid potential leakage transitions, and which in general can be achieved by increasing the area of the circuit for a given size (in terms of number of anchorages).

However, one goal of DNA circuit design is in fact to minimise the circuit area (Jung et al. 2015). Thus the ability to identify leakage transitions is an important step in the process of optimising DNA circuit layouts where the aim is to minimise the computational error inherent in a circuit while minimising the area of the circuit. This trade-off is quantified by a combination of structural and probabilistic analysis techniques including performability measures building on impulse rewards.

Moreover, we show how coloured Petri nets can be used to obtain a generic template for specifying DNA walker systems, while preserving the ability of a mathematically rigorous assessment of the system specification. This template may be easily adjusted to different stepping scenarios or distance notions without requiring programming skills. In this paper, we consider 2-dimensional walker systems. However, the extension of our flexible modelling approach to the 3-dimensional case is straightforward.

1.2 Outline

In the next section we discuss the modelling of DNA walker circuits, first as planar undirected graphs, which we convert into Petri nets to be able to analyse their execution, and finally into coloured Petri nets to obtain a concise and flexible circuit specification incorporating 2D topology information. Afterwards we introduce in Sect. 3 our new technique to identify leakage transitions, followed by a brief overview on Petri net related analysis techniques with a special focus on stochastic analyses in Sect. 4. We demonstrate the usability of our techniques by comparing two layouts for a given DNA walker circuit taken from Dannenberg et al. (2015). We conclude our paper in Sect. 5 with a brief summary and outlook on future work.

2 Modelling

2.1 DNA walker systems

We consider programmable DNA walker circuits introduced in Yin et al. (2004), Bath et al. (2005), Wickham et al. (2011) and Wickham et al. (2012), which are known to exhibit an inherently probabilistic behaviour. DNA walker circuits have been modelled and analysed with the PRISM tool (Dannenberg et al. 2015; Dannenberg 2016), and later by help of stochastic Petri nets (Barbot and Kwiatkowska 2015). To be self-contained we recall the basic facts required to understand our modelling approach deploying coloured stochastic Petri nets.

The DNA walker circuits under consideration are supposed to compute a Boolean function over n input variables, i.e., $\mathcal{B}^n \rightarrow \mathcal{B}$. Formally, a DNA walker circuit defines a planar *undirected graph*, in the following called DSD graph; see Fig. 1 for an example. Vertices stand for anchorages and undirected edges for possible walker steps. Vertices with two adjacent edges form linear tracks of the walker circuit, while vertices with three adjacent edges represent gates, i.e., either forking or joining junction

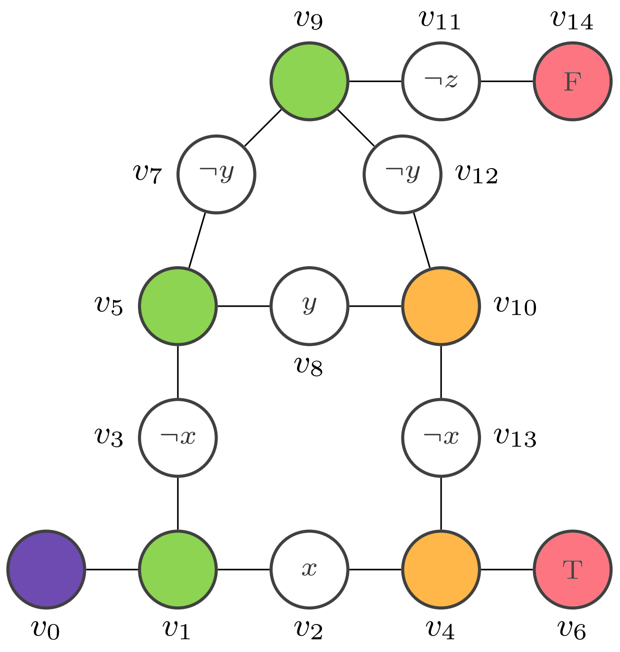


Fig. 1 DSD graph representing the boolean function $x \vee y \vee z$. Colour code: blue—INIT, green—FORK, orange—JOIN, red—FINAL; uncoloured—NORM; ϵ label not shown. (Color figure online)

points. There are no vertices with more than three adjacent edges, because there is a lack of experimental evidence (Wickham et al. 2011, 2012).

Anchorages may be labelled with literals over the domain of the Boolean function to be evaluated. The programming of the circuit is achieved by blocking those anchorages whose labels are evaluated to false for the given input values. These observations are summarised in the following definition which builds on the one given in Dannenberg (2016).

Definition 1 (*DSD graph, syntax*) A DSD graph is a tuple $\mathcal{G} = (V, E, In, L, Out)$, where

- V is the set of vertices, with $V = V_{INIT} \cup V_{NORM} \cup V_{FORK} \cup V_{JOIN} \cup V_{FINAL}$, and
  - $V_{INIT} = \{v_0\}$—the unique initial anchorage,
  - V_{NORM}—the vertex set of normal anchorages,
  - V_{FORK}—the vertex set of fork anchorages,
  - V_{JOIN}—the vertex set of join anchorages,
  - V_{FINAL}—the vertex set of final anchorages,

  and all vertex sets pairwise disjunctive.
- E is the set of undirected edges with $E \subset (V \times V)$. The initial vertex v_0 and final vertices have one adjacent edge, normal vertices have two adjacent edges, and junction vertices (either fork or join) have three adjacent edges.
- In is a set of Boolean variables, and $literals(In)$ yields the set of all value assignments over In.
- L is a labelling function with $L{:}V \setminus V_{FINAL} \rightarrow literals(In) \cup \{\epsilon\}$.
- Out is an output function with $Out : V_{FINAL} \rightarrow \{T, F\}$, assigning a truth value to each final vertex. □

Remarks

- All undirected edges have exactly one direction which corresponds to a step directed from the init vertex to one of the final vertices.
- By definition, the walker can never leave a final node; however this cannot be deduced from the undirected graph.
- Binary decision trees do not require join anchorages.
- There has to be at least one final vertex. Usually the graph will contain at least two final vertices, with each truth value occurring at least once.
- DSD graphs with exactly one final vertex labelled with *true* allow for composability, e.g., exploiting origami tiles.
- The empty label ϵ permits unblockable anchorages and is typically not displayed. The unique initial anchorage v_0 should be labelled with ϵ.
- The definition given in Dannenberg (2016) does not distinguish between fork and join, and it assigns $literals(in)$ to edges; we assign them to vertices.

A DSD graph may be seen as a finite automaton. It describes a map with all possible steps a DNA walker may take to execute the computation encoded in the underlying (binary decision) DAG for any input values. Accordingly, for a given set of input values, a walker is supposed to go only to anchorages, where the evaluation of the label yields *true*. The anchorages where the evaluation yields *false* are considered to be blocked; thus can not be visited by the walker. Assuming a consistent labelling, for each possible set of input values, there exists a path from the initial to a final vertex delivering the result. If there exists exactly one path, we call the DSD graph *deterministic* (Dannenberg 2016).

The DNA walker starts its journey at the unique initial vertex, and then follows one of the adjacent edges to reach a neighbouring unblocked vertex, which is repeated until reaching a final vertex, which by definition can not be left again. The final vertex reached indicates the result computed by the walker's journey through the DNA circuit.

Undirected edges can be read as a shorthand notation for two opposite, directed edges; these two directed edges stand for possible walker steps in opposite directions. Thus, a DNA walker does not go on a target-oriented journey; it can not distinguish between fork and join anchorages, and all anchorages reachable in one step have the same probability to be visited next. For example, assuming $x = true$ in Fig. 1, a walker may repeatedly move along $v_0 - v_1 - v_2 - v_4$ (in both directions), before accidentally finding the final vertex v_6. To put it differently, the challenge consists in realising an algorithm working on a directed graph (the DAG) by use of an undirected graph (the DSD graph).

A DNA walker's life becomes slightly easier in "burnt-bridges" circuits, where each position can only be visited once. As already visited positions are not among the possible choices of target positions for the next step, the walker will generally be driven in the direction of a final vertex.

This execution semantics goes beyond standard graph-based reasoning and is not covered by Definition 1. To formalise the execution semantics, we convert the DSD graph into a Petri net—first into a plain Petri net, inspired by the approach introduced in Barbot and Kwiatkowska (2015), and afterwards into a coloured Petri nets, which will yield a concise template for DNA walker circuit specifications.

Stepping distance It has been observed that a walker may move in one step to all unblocked anchorages within a certain radius, but with different probabilities (Wickham et al. 2011). We employ the approximation reported in Dannenberg et al. (2015) and assume that the walker stepping rate k is a piecewise function of the distance d over the maximum interaction distance d_M, the average distance between anchorages d_a, and the base rate k_s, given by:

$$k = \begin{cases} k_s & \text{if} \quad d \le 1.5 \cdot d_a \\ k_s/50 & \text{if} \quad 1.5 \cdot d_a < d \le 2.5 \cdot d_a \\ k_s/100 & \text{if} \quad 2.5 \cdot d_a < d \le d_M \\ 0 & \text{else,} \end{cases} \tag{1}$$

with $d_M = 24\,\text{nm}$, $d_a = 6.2\,\text{nm}$, $k_s = 0.009\,\text{s}^{-1}$. This will generally add further unintentional undirected edges to the DSD graph; how many and which ones depends on the topology.

2.2 Petri nets

To be self-contained we briefly recall basic Petri net concepts, which will allow us to formally treat the execution semantics of DSD graphs; for more details see Heiner et al. (2008).

Definition 2 (*Petri net, syntax*) A Petri net is a tuple $\mathcal{N} = (P, T, f, m_0)$, where

- P and T are finite, non-empty, and disjoint sets. P is the set of *places*, and T is the set of *transitions*.
- $f : ((P \times T) \cup (T \times P)) \to N_0$ defines the set of directed *arcs*, weighted by non-negative integer values.
- $m_0 : P \to N_0$ gives the *initial marking*. □

The pre-set of a node $x \in P \cup T$ is defined as ${}^\bullet x := \{y \in P \cup T | f(y,x) \neq 0\}$, and its post-set as $x^\bullet := \{y \in P \cup T | f(x,y) \neq 0\}$. We extend both notions to a set of nodes $X \subseteq P \cup T$ and define the set of all pre-nodes ${}^\bullet X := \bigcup_{x \in X} {}^\bullet x$, and the set of all post-nodes $X^\bullet := \bigcup_{x \in X} x^\bullet$.

Definition 3 (*Petri net, semantics*) Let $\mathcal{N} = (P, T, f, m_0)$ be a Petri net.

- A transition t is *enabled* in a marking m, written as $m[t\rangle$, if $\forall p \in {}^\bullet t : m(p) \ge f(p,t)$, else *disabled*.
- A transition t, which is enabled in m, may fire.
- When t in m *fires*, a new marking m' is reached, written as $m \xrightarrow{t} m'$, with

$$\forall p \in P : m'(p) = m(p) - f(p,t) + f(t,p).$$

- The firing happens atomically. □

In qualitative (time-free) Petri nets, the firing does not consume any time, while in stochastic Petri nets, transitions are associated with generally state-dependent firing rates. The repeated firing of enabled transitions (the game) yields the behaviour of a Petri net. Generally, there are more than one transition enabled in a given marking. Then the decision of the transition to fire next is taken non-deterministically in time-free Petri nets, and in accordance with the stochastic firing rates in stochastic Petri nets.

Transforming a DSD graph into a Petri net is straightforward: the vertices are turned into Petri net places and directed edges into Petri net transitions, such that the source and sink vertex of a given edge become the pre- and post-place of the corresponding transition, see Fig. 2. We keep the terminology introduced for DSD graphs and speak of init/norm/fork/join/final places. Finally, we model the DNA walker by a token which we set on the init place. Now, playing the token game will produce all possible paths (of arbitrary length) a walker can take for any input values, which will sooner or later end in a final place. The system behaviour has reached an intended dead state (no transition is enabled).

To control, how often a place can be visited, we adopt the modelling idea introduced in Barbot and Kwiatkowska (2015), compare Fig. 3. Initially all unblocked places hold one token, indicating that the place can be visited, and the init place holds additionally a token representing the walker. Then, a directed edge of the DSD graph going from vertex A to vertex B is modelled by a Petri net transition,

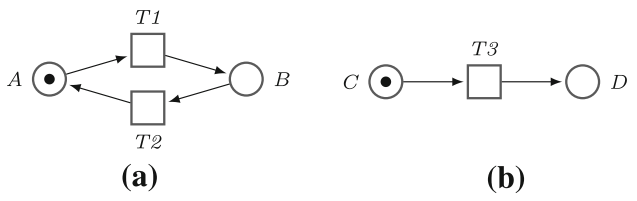


Fig. 2 DNA walker basic stepping scenarios, with A, B, C non-final vertices, and D final vertex. **a** Standard step, **b** final step

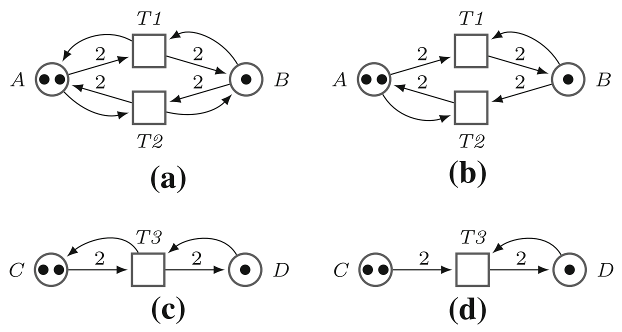


Fig. 3 DNA walker stepping scenarios, with A, B, C non-final vertices and D final vertex. In this paper, we focus on the "burnt-bridges" setting. **a** Unguided step, **b** "burnt-bridges" step, **c** unguided final step, **d** "burnt-bridges" final step

which checks if the target place B can be visited (is not blocked). When a walker moves from place A to place B, then in total for

- *Unguided scenario*: one token is removed from A, because the walker leaves A, and one token is kept, because A can be re-visited,
- *"Burnt-bridges" scenario*: two tokens are removed from A, because the walker leaves A, and A can not be re-visited.

In both cases, a second token is added to B, because the walker is now on B. In these scenarios, playing the token game will produce a path the walker takes, which typically goes straight to a final place (assuming a consistent labelling). This path will be unique for deterministic DSD graphs. By repeatedly re-initialising the Petri net we can explore all possible paths for any input values, which will be systematically done in the next section.

In summary, the conversion of a DSD graph into a Petri net is rather flexible and can be conveniently adjusted to the particular execution semantics on hand. In the following we focus on the "burnt-bridges" scenario, which is also easier to implement in DNA than in the "unguided" approach (Bath et al. 2005).

2.2.1 Stepping distance

So far, our Petri net model only contains those steps that a walker may take to follow a path in a given DSD graph. If the DSD graph were to be directed, these steps would all be intentional.

The walker stepping rate k (see Eq. 1) introduces three categories of steps: short distance, medium distance and long distance steps. Ideally a spatial layout of a walker circuit should ensure that all short distance steps correspond to edges in the DSD graph. The number of additional medium and long distance steps obviously depends on the topology. We discuss the influence of the topology on the number of steps in each category in Sect. 3.

These additional steps may introduce unwanted behaviours. Now, a walker can take shortcuts along a path, jump backwards in the path just taken, or jump to another branch, known as leakage steps (transitions). As a result, a walker can get lost in a non-final vertex without any neighbouring vertex free to be visited; technically speaking—the system behaviour may reach an unwanted dead state. To be able to distinguish between wanted and unwanted dead states, we add a loop (i.e.,a transition having the same pre- and post-place) to all places modelling final vertices. The number of leakage transitions and dead states will have an influence on a circuit's reliability, which we quantify in Sects. 3 and 4.

2.2.2 Fault model

The programming of a walker circuit according to the given input values of the Boolean function to be computed is realised by the blocking of the correspondingly labelled anchorages. This blocking mechanism may fail. To reflect this we follow the approach introduced in Barbot and Kwiatkowska (2015) and add a fault model to the Petri net we obtained so far, modelling the failure of the blocking mechanism; see Fig. 4.

In order to simplify the modelling, we now assume that all places hold initially a token. For the anchorages to be blocked, this token is removed by additionally added blocking transitions. However, the firing of blocking transitions can be prevented by transitions in conflict, representing the occurrence of a failure. If a failure transition fires, an anchorage to be blocked remains unblocked, and thus a walker can move to this anchorage and follow an incorrect path.

As the programming has to happen before the walker reaches a junction, transitions representing the blocking mechanism and its potential failure are modelled as *immediate transitions*, i.e., transitions which fire without any time delay and highest priority, thus before any stochastic transition will fire; see Heiner et al. (2009) for details. We assume a uniform failure of the blocking mechanism; thus all pairs of immediate transitions are equally weighted with a probability of f = 0.7 for the blocking transitions, and a probability of f = 0.3 for the failure transitions.

2.3 Coloured Petri nets

Colouring yields a form of high-level Petri nets which permit the description of similar network structures in a concise way using colours grouped in colour sets—to be understood as a synonym for discrete data types as known from programming languages. The colouring principle can be equally applied to qualitative and quantitative Petri nets (Blätke et al. 2015), and we use it in this paper to obtain coloured stochastic Petri nets.

Coloured Petri nets can be constructed from uncoloured Petri nets by folding, when partitions of places and transitions are given. These partitions define the colour sets of the coloured net. Vice versa, coloured Petri nets with finite colour sets can be automatically unfolded into uncoloured Petri nets, which then allows the application of all analysis

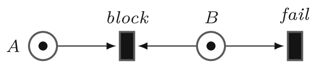


Fig. 4 Fault model added for every anchorage (place) A to be blocked. The success rate of *block* is assumed to be $f = 0.7$ and of *fail* $f = 0.3$

techniques available for the corresponding unfolded net class.

Coloured Petri nets consist, like standard Petri nets, of places, transitions and arcs. Additionally, a coloured Petri net is characterised by a set of colour sets, and related net inscriptions, which together permit to distinguish tokens by their colours. Defining coloured Petri nets formally would exceed the given space limit; see Liu et al. (2012) for details. Here, we confine ourselves to introduce the essential concepts by means of our application scenario. To illustrate our modelling ideas we use the toy example shown in Fig. 5.

Petri nets can be specified graphically or textually; our tools support both, the latter by use of the Coloured Abstract Net Description Language (CANDL) format (Schwarick et al. 2016). The following description is a combination of both.

2.3.1 Encoding the vertices

DNA walkers perform spatially localised computation (Barbot and Kwiatkowska 2015). Colour permits to encode locality, as we have shown in Gilbert et al. (2013). We start with defining a regular rectangular grid; we need seven rows and eight columns for our toy example. The 2D Cartesian coordinates are represented by pairs of colours (integers).

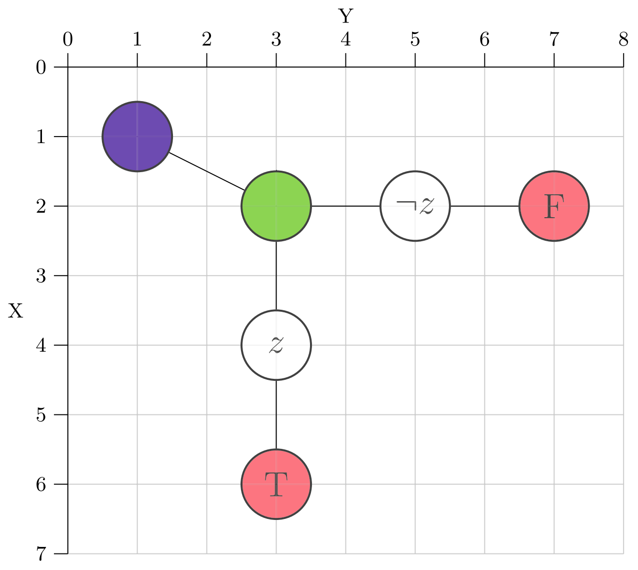


Fig. 5 Toy example to illustrate the use of coloured Petri nets. Colour code: blue—INIT, green—FORK, red—FINAL; uncoloured—NORM. (Color figure online)

```
constants:
  int D1 = 7;
  int D2 = 8;

colorsets:
  CD1 = {1..D1};           // row indices
  CD2 = {1..D2};           // column indices
  Grid2D = PROD(CD1,CD2); // rectangular grid

variables:
  CD1 : x; CD2 : y;
```

Now, the tuple (x, y) permits to address the grid element in the x-th row and the y-th column, compare Fig. 5. To encode all attributes of the vertices in the DSD graph, we define two further colour sets of enumeration type.

```
colorsets:
enum Type = {INIT,FINAL,NORM,FORK,JOIN};
enum Label= {E,T,F,Z,NZ};
            // Epsilon, True, False, Z, NotZ
variables:
  Type : z; Label : w;
```

The colour set *Label* has to be adjusted to *literals*(*In*) of a given DSD graph; see Definition 1. Now we have all ingredients to introduce the data type for the vertices, which is a product type over the colour sets *CD*1, *CD*2, *Type*, and *Label*.

```
colorsets:
Circuit = PROD(CD1,CD2,Type,Label);
```

We need two coloured places of this type, *A*—for the anchorages, and *B*—for the blocking mechanism.

The existing vertices with their attributes are defined by the Boolean function *Positions* by enumerating all tuples. Each tuple has to be of the type *Circuit*. Obviously, the definition of this function needs to be adjusted to the given DSD graph. Our toy example has six vertices, so we have six tuples here.

```
colorfunctions:
bool Positions(CD1 x, CD2 y, Type z, Label w)
{
    (x=1 & y=1 & z=INIT  & w=E) |
    (x=2 & y=3 & z=FORK  & w=E) |
    (x=4 & y=3 & z=NORM  & w=Z) |
    (x=2 & y=5 & z=NORM  & w=NZ)|
    (x=6 & y=3 & z=FINAL & w=T) |
    (x=2 & y=7 & z=FINAL & w=F)
};
```

2.3.2 Distance metrics

In our modelling approach we are bound to apply a discrete metric, which prevents the use of the popular Euclidean distance. The generalized distance $L_m = \|\cdot\|_m$ between two points p_1 and p_2 in a plane is defined as

$$\|p_1 - p_2\|_m = (|x_1 - x_2|^m + |y_1 - y_2|^m)^{\frac{1}{m}}, \quad (2)$$

also known as Minkowski distance (Cormen et al. 2001). The rectilinear or Manhattan distance is the L_1 distance and is the sum of the absolute differences of the points' coordinates

$$\|p_1 - p_2\|_1 = |x_1 - x_2| + |y_1 - y_2|. \quad (3)$$

The Euclidean distance is the L_2 distance and gives the length of the straight line between two points in Euclidean space

$$\|p_1 - p_2\|_2 = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \quad (4)$$

The L_∞ distance, also known as Chebyshev or chessboard distance, is the limit of L_m distances for $m \to \infty$; it is defined as

$$\|p_1 - p_2\|_\infty = max(|x_1 - x_2|, |y_1 - y_2|), \quad (5)$$

see Cormen et al. (2001) for details. Equations (3) and (5) yield discrete results for a discrete grid. The combination of the Manhattan distance (L_1) and the chessboard distance (L_∞) provides the required results, see Fig. 6, and we define corresponding colour functions; see "Appendix" section for details.

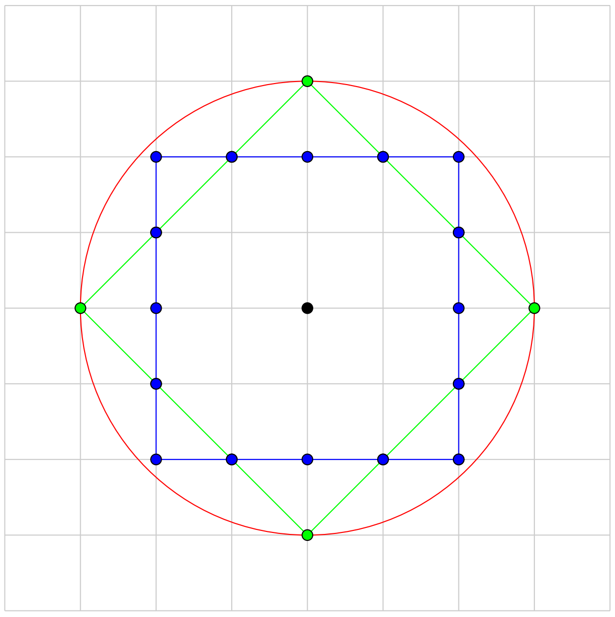


Fig. 6 Discrete distance function; $L_1 = 3$ (green) combined with $L_\infty = 2$ (blue) yield together all discrete points within $L_2 = 3$ (red). (Color figure online)

2.3.3 Encoding the walker steps

In the following we use the discretised version of Eq. (1):

$$k = \begin{cases} k_s & \text{if} \quad d \le dS \\ k_s/50 & \text{if} \quad dS < d \le dM \\ k_s/100 & \text{if} \quad dM < d \le dL \\ 0 & \text{else,} \end{cases} \quad (6)$$

with $dS = 3, dM = 5, dL = 8$. It is obvious how to adjust the resolution of the discretisation to the required precision.

We define for each step category

–[short|medium|long] distance [standard|final] steps—a coloured transition and illustrate it here for the short distance standard steps, see Fig. 7. In coloured Petri nets, arcs are weighted with formal sums of tuples: the transition *stepShort* requires two tokens with values bound to ($x1$, $y1$, $z1$, $w1$) and one token with values bound to ($x2$, $y2$, $z2$, $w2$), with the constraint that these two tuples relate to short distance neighbours and none of them is a final vertex. This constraint is expressed as transition guard (given in brackets), which is technically a Boolean expression. A coloured transition can fire for specific values bound to all variables occurring at its adjacent arcs, if its guard is evaluated to true. We introduce the following functions for the transition guard.

```
colorfunctions:
bool ShortDistance (CD1 x1,CD2 y1,CD1 x2,CD2 y2)
     {RectilinearDistance(x1,y1,x2,y2) <= dS
       || ChessboardDistance(x1,y1,x2,y2) < dS};

bool NoSelfLoop    (CD1 x1,CD2 y1,CD1 x2,CD2 y2)
     {(x1 != x2 | y1 != y2)};

// to be used as transition guard
bool IsNeighbourShortD(CD1 x1,CD2 y1,Type z1,Label w1,
                       CD1 x2,CD2 y2,Type z2,Label w2)
     {ShortDistance(x1,y1,x2,y2) &&
      NoSelfLoop(x1,y1,x2,y2)  &&
      Positions(x1,y1,z1,w1) &&
      Positions(x2,y2,z2,w2)};
```

We proceed likewise for the other step categories. Finally, we introduce a coloured transition *loop* which keeps the walker technically alive when having reached a final vertex; see Fig. 12 in "Appendix" section.

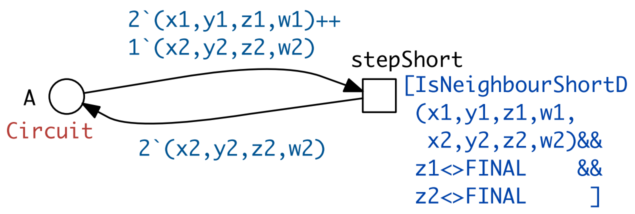


Fig. 7 Coloured transition encoding all short distance (regular) steps according to Fig. 3b

2.3.4 Blocking of anchorages and fault model

Similarly, we introduce coloured transitions encoding the blocking and its potential failure for all vertices labelled with an element of $Literals(In)$. For this purpose, we need a further function:

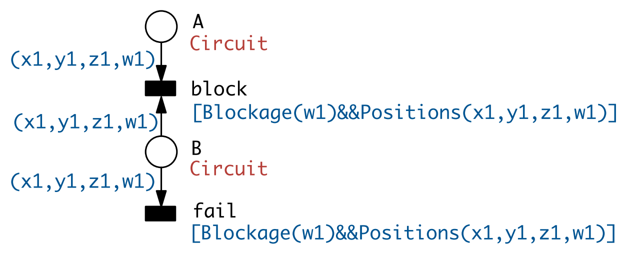
```
bool Blockage(Label w) {
    w!=E & w!=T & w!=F
};
```

to guard the firing of the two coloured immediate transitions *block* and *fail*; see Fig. 8. Which anchorages have to be blocked for a given set of input values is controlled by the initial marking of the place B.

In the appendix, we provide the complete CANDL specification for this toy example. It can be used as template to specify any DNA walker circuit according to Definition 1 as coloured SPN; compare workflow, first step in "Appendix" section. Unfolding these coloured Petri nets generates Petri nets as described in Sect. 2.2. The adaptation of the template to any of the execution semantics discussed there is straightforward.

2.3.5 Comparing both Petri net modelling approaches

There is no explicit notion of space in the uncoloured Petri net model, which encodes in a undifferentiated manner the intensional transitions of the DSD graph structure together with medium and long distance relationships. However the functionality of the DNA circuit is also influenced by the 2D topology of the net, for example leakage, and colour permits the construction of multi-dimensional models (Heiner and Gilbert 2013), Thus the coloured Petri net model explicitly contains all locality data, which are exploited in the automated unfolding to generate all transitions according to the defined neighbourhood relations. Using coloured Petri nets, adjusting the model to different distance notions does not require programming skills, and the extension of the approach to the 3-dimensional scenario is straightforward.



A
Circuit
(x1,y1,z1,w1)
block
[Blockage(w1)&&Positions(x1,y1,z1,w1)]
(x1,y1,z1,w1)
B
Circuit
(x1,y1,z1,w1)
fail
[Blockage(w1)&&Positions(x1,y1,z1,w1)]

Fig. 8 Coloured immediate transitions encoding the blocking and its potential failure for all vertices labelled with $Literals(In)$; compare Fig. 4

More importantly, the use of colour enables the grid layout of the graph and inter-anchorage distance to be directly encoded in the model, and thus leakage can be directly extracted from the model as we will see in the next section.

3 Identification of leakage transitions

One of the major issues with existing modelling approaches for DNA walker circuits is the inability to automatically identify leakage transitions. We present an algorithm that investigates the structure of the unfolded Petri net in order to identify leakage transitions.

3.1 Place indexing

The underlying idea of the algorithm is to follow the short distance sub-graph (i.e.,the sub-graph comprising only short distance steps), which should unambiguously correspond to the DAG of the intended computation. Any further short distance transitions make the computational DAG ambiguous; these additional transitions may be shortcuts or leakage transitions.

To identify the computational DAG in a given short distance sub-graph we borrow a simple labelling principle widely used to efficiently organise the nodes of a left-complete binary tree in an array data structure (Cormen et al. 2001). Node labels are defined over $\mathbb{N}^+$ and serve as indices in the array, and simple operations over the array indices give direct access to a node's parent or children nodes; see Fig. 9. We extend this idea to index DAGs, which however makes everything a bit more complicated, as we now have to deal with join vertices as well.

We employ a breadth first search (BFS) over all places of the net, starting with the INIT place that is indexed with 1. Each place that we have to examine is added to a queue, i.e.,a first-in–first-out (FIFO) data structure ensuring a BFS. By indexing the visited places we follow automatically shortest paths to the final places, but with one exception that needs special care, see JOIN places below. To identify the successors of a given place x, we introduce a new notation x°, providing the set of post-places y of all post-transitions of x satisfying

$$\forall y \in (x^\bullet)^\bullet : index(y) \neq index(x) \wedge [index(y) \\ = 0 \vee index(y) \neq index(x)/2].$$

While indexing the places, we collect transition types, which are characterised by triples (pre-node index, post-node index, [TRACK|FORK|JOIN|LEAK]).

Let's consider the following cases which may occur when indexing an *unambiguous short distance sub-graph*.

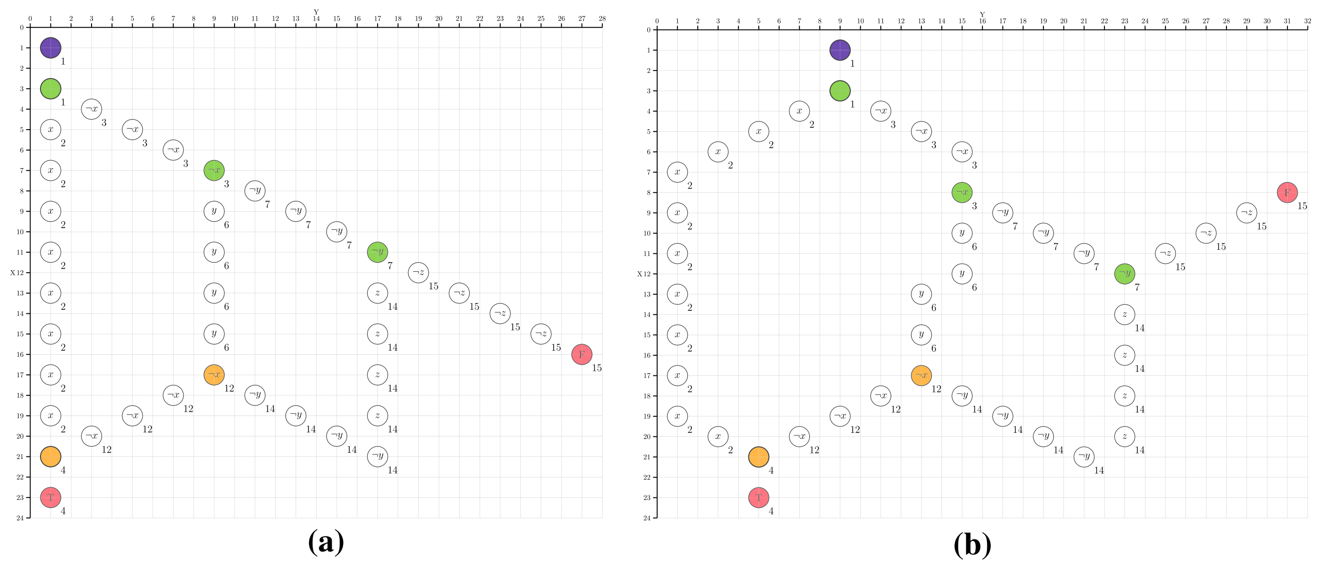


Fig. 9 Two layouts for the DSD graph given in Fig. 1 for the Boolean function $x \vee y \vee z$. Both layouts are adapted versions from Dannenberg (2016). The numbers shown next to the vertices are the indices generated by Algorithm 1 for the transition classification. **a** Naive layout, **b** optimized layout

- The unique place of type INIT has to have exactly one successor by definition. The successor of INIT gets the same index, i.e.,1, and the successor is added to the queue. The transition from INIT to its successor is part of a (linear) track, thus we add a tuple of $(1, 1, TRACK)$ to the known transition types.
- For binary forward branching, a node c of type FORK has 3 short distance neighbours, one corresponds to the predecessor and the others to the two branches the computational path may take. The first of the two successors c_0° gets an index that is two times the index of c. Then c_0° is added to the queue. The second successor c_1° gets an index that is two times the index of c plus one. Then c_1° is added to the queue. The transitions from c to its successors are FORK transitions, so we add two triples $(c, c_i^\circ, \text{FORK})$ to the known transition types.
- Binary backward branching takes place on a node c of type JOIN and as such c has 3 short distance neighbours, two predecessors and one successor. When the algorithm reaches c, it may happen that two of the neighbours are not indexed yet. In this case we are not able to decide which one is the predecessor and which one is the successor, so we postpone that decision and treat both places as if they each would be a successor. The successor gets the same index as c and is added to the queue. The transitions between c and its successor are of type TRACK, i.e.,the tuple of both indices and type TRACK is added to the known transition types.
- A node of type NORM in a linear track has exactly two short distance neighbours, one corresponds to the predecessor, and one to the successor node. But there are several cases to deal with.

  1. The successor is not indexed yet and is of type JOIN. So it gets an index two times of c, and the transition must be of type JOIN too. The successor is added to the queue.
  2. The successor is not indexed yet and is not of type JOIN. So it gets the same index as c, because it is on the same track. Thus the transition is of type TRACK. The successor is added to the queue.
  3. The index of the successor is smaller than the index of c and the successor is of type JOIN. So we have reached an already visited backward branch and add the tuple of indices and type JOIN to the known transition types.
  4. The index of the successor is smaller than the index of c and the successor is not of type JOIN. So we override the successor's index and add it to the queue, because we are backtracking to a previously visited track.

  In each of the previous cases, the transition between c and its successor is of type JOIN or TRACK, i.e., a triple of both indices and type JOIN or TRACK is added to the known transition types.

  A normal node may have 3 short distance neighbours in the case of leak transitions. In this situation we have to take care of several cases.

1. The successor is not indexed yet and is of type JOIN, so it gets an index two times of c and the transitions must be of type JOIN too, the successor is added to the queue and is additionally marked.
2. The successor is not indexed yet and is not of type JOIN, so it gets the same index as c, because they are on the same track. Thus the transitions are of type TRACK and the successor is added to the queue and is additionally marked.
3. The successor is marked and is of type JOIN, so we have reached an already visited backward branch and add a tuple of the indices and type JOIN to the known transition types.
4. The successor is marked and is not of type JOIN, so we overwrite the successor's index, if it is smaller, because we reached a corner with shortcut transitions.
5. The successor is not marked; thus the transition must be a leak transition. The transitions from c to its successors are LEAK transitions, so we add two triples (c, c_i°, LEAK) to the known transition types.

In the first four cases, the transitions between c and its successor are of type JOIN or TRACK, i.e., tuples of both indices and type JOIN or TRACK are added to the known transition types.

- A node of type FINAL has only one short distance neighbour, which is then a predecessor, so there are no more nodes to investigate.

The algorithm terminates, when the queue is empty and all places are indexed. Furthermore, we obtain a set of tuples defining the known transition types between pairs of indices. In all other cases, the algorithm sends a warning and terminates; see Algorithm 1 for its pseudo code.

3.2 Transition classification

Having indexed the short distance sub-graph, we classify the steps deploying the set of known transition types, i.e., the set M computed by Algorithm 1. There are four types of steps: TRACK, FORK, JOIN and LEAK.

- The pre-place and post-place of a TRACK transition t have the same index, forming a linear track, i.e.,$index(t^\bullet) = index(^\bullet t)$.
- For the indices of the pre- and post-place of a FORK transition t, it holds either $index(t^\bullet) = 2 \cdot index(^\bullet t)$ or $index(t^\bullet) = 2 \cdot index(^\bullet t) + 1$.
- For the indices of the pre- and post-place of a JOIN transition t, it holds either $index(t^\bullet) = 2 \cdot index(^\bullet t)$ or $index(t^\bullet) < index(^\bullet t)$.
- There is one precisely defined case of LEAK transitions—a leak that follows directly after a fork. Thus, any transition satisfying $index(t^\bullet) = index(^\bullet t) + 1$ or $index(^\bullet t) = index(t^\bullet) - 1$ is a LEAK transition. However, depending on the layout, a leak can occur anywhere between two places having different indices.

The classification of steps into TRACK, FORK, JOIN and LEAK takes place by looking up the set M of known transition types. If there is a tuple in M of the indices of the pre- and post-place (or vice-versa) of the step, then the associated type is allocated to the step. All remaining, non-classified steps have to be leaks and are classified accordingly. In combination with the three step distances (short, medium, long), we are able to provide a concise classification of all transitions into 12 categories. Short distance leakage transitions clearly indicate potential for layout improvement.

```
Algorithm 1 leak detection algorithm
Require: short-distance graph as $\mathcal{SPN}$,
Ensure: indexed short-distance graph, transition classification
 1: $\forall p \in P : \mathrm{IDX}(p) \leftarrow 0$
 2: $\forall p \in P : \mathrm{MARK}(p) \leftarrow FALSE$
 3: $Q.enqueue(INIT)$
 4: $\mathrm{IDX}(INIT) \leftarrow 1$
 5: $M \leftarrow \emptyset$                ▷ contains tuples $(IDX, IDX, TRANS)$
 6: while $Q \neq \emptyset$ do
 7:     $c \leftarrow Q.dequeue()$
 8:     switch $\mathrm{TYPE}(c)$
 9:         case $INIT$ :
10:             if $|c^{\circ}| = 1$ then
11:                 $\mathrm{IDX}(c^{\circ}_0) \leftarrow \mathrm{IDX}(c)$
12:                 $M \leftarrow M \cup (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_0), TRACK)$
13:                 $Q.enqueue(c^{\circ}_0)$
14:             end if
15:         end case
16:         case $FORK$ :
17:             if $|c^{\circ}| = 2$ then
18:                 for $i \leftarrow 0, 1$ do
19:                     $\mathrm{IDX}(c^{\circ}_i) \leftarrow \mathrm{IDX}(c) * 2 + i$
20:                     $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_i), FORK)$
21:                     $Q.enqueue(c^{\circ}_i)$
22:                 end for
23:             end if
24:         end case
25:         case $JOIN$ :
26:             if $|c^{\circ}| = 1 \vee |c^{\circ}| = 2$ then
27:                 for $i \leftarrow 0, |c^{\circ}| - 1$ do
28:                     if $\mathrm{IDX}(c^{\circ}_i) = 0$ then
29:                         $\mathrm{IDX}(c^{\circ}_i) \leftarrow \mathrm{IDX}(c)$
30:                         $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_i), TRACK)$
31:                         $Q.enqueue(c^{\circ}_i)$
32:                     end if
33:                 end for
34:             end if
35:         end case
36:         case $NORM$ :
37:             if $|c^{\circ}| = 1$ then
38:                 if $\mathrm{IDX}(c^{\circ}_0) = 0$ then
39:                     if $\mathrm{TYPE}(c^{\circ}_0) = JOIN$ then
40:                         $\mathrm{IDX}(c^{\circ}_0) \leftarrow \mathrm{IDX}(c) * 2$
41:                         $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_0), JOIN)$
42:                     else
43:                         $\mathrm{IDX}(c^{\circ}_0) \leftarrow \mathrm{IDX}(c)$
44:                         $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_0), TRACK)$
45:                     end if
46:                     $Q.enqueue(c^{\circ}_0)$
47:                 else if $\mathrm{IDX}(c^{\circ}_0) < \mathrm{IDX}(c)$ then
48:                     if $\mathrm{TYPE}(c^{\circ}_0) = JOIN$ then
49:                         $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_0), JOIN)$
50:                     else
51:                         $\mathrm{IDX}(c^{\circ}_0) \leftarrow \mathrm{IDX}(c)$
52:                         $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_0), TRACK)$
53:                         $Q.enqueue(c^{\circ}_0)$
54:                     end if
55:                 end if
56:             else if $|c^{\circ}| = 2$ then
57:                 for $i \leftarrow 0, 1$ do
58:                     if $\mathrm{IDX}(c^{\circ}_i) = 0$ then
59:                         if $\mathrm{TYPE}(c^{\circ}_i) = JOIN$ then
60:                             $\mathrm{IDX}(c^{\circ}_i) \leftarrow \mathrm{IDX}(c) * 2$
61:                             $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_i), JOIN)$
62:                         else
63:                             $\mathrm{IDX}(c^{\circ}_i) \leftarrow \mathrm{IDX}(c)$
64:                             $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_i), TRACK)$
65:                         end if
66:                         $\mathrm{MARK}(c^{\circ}_i) \leftarrow TRUE$
67:                         $Q.enqueue(c^{\circ}_i)$
68:                     else if $\mathrm{MARK}(c^{\circ}_i) = TRUE$ then
69:                         if $\mathrm{TYPE}(c^{\circ}_i) = JOIN$ then
70:                             $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_i), JOIN)$
71:                         else if $\mathrm{IDX}(c^{\circ}_i) < \mathrm{IDX}(c)$ then
72:                             $\mathrm{IDX}(c^{\circ}_i) \leftarrow \mathrm{IDX}(c)$
73:                             $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_i), TRACK)$
74:                         end if
75:                     else
76:                         $M \leftarrow (\mathrm{IDX}(c), \mathrm{IDX}(c^{\circ}_i), LEAK)$
77:                     end if
78:                 end for
79:             end if
80:         end case
81:     end switch
82: end while
```

3.3 Case study

For demonstration purposes we use two layouts for the Boolean function $x \vee y \vee z$, which are inspired by Dannenberg (2016). The first layout can be regarded as rather naive and incorporates some flaws resulting in leakage transitions, see Fig. 9a. The second layout is optimised in the sense that the number of leakage transitions is reduced, see Fig. 9b. Tables 1 and 2 show the results for the two layouts of Fig. 9. They confirm that the layout in Fig. 9b is better than the layout in Fig. 9a with respect to leakage transitions. As expected, the number of short distance FORK and short distance JOIN transitions are the same for both layouts.

Table 1 Transition classification for the naive layout in Fig. 9a

	Short	Medium	Long	Σ
Track	64	51	51	166
Fork	12	30	54	96
Join	8	17	33	58
Leak	8	36	98	142

Table 2 Transition classification for the optimised layout in Fig. 9b

	Short	Medium	Long	Σ
Track	66	57	57	180
Fork	12	34	62	108
Join	8	20	30	58
Leak	2	20	58	80

3.4 Implementation

The algorithm presented for indexing an unambiguous short distance sub-graph satisfying Definition 1 enables us to classify step transitions and thereby to identify leakage transitions. It is implemented in our advanced Petri net analysis tool MARCIEand is available under http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Marcie.

4 Analysis techniques

4.1 Qualitative analysis

We obtain the Petri nets to be analysed by automatic unfolding of coloured SPNs following our template for DNA walker circuit specification. Thus, by construction, we always obtain a very special net class of Petri nets, which correspond—from a behavioural point of view—to finite automata. At any point of time, the walker can be at exactly one anchorage, the walker cannot multiply itself and can never disappear.

To increase our confidence in the template and to deepen our understanding of its behaviour, we apply a popular analysis technique relying on an exhaustive description of all possible behaviour. For this, we compute all markings (system states) reachable from the initial marking m_0 by any firing sequence of arbitrary length, written as $[m_0\rangle$, forming the *state space* of a given Petri net. The reachability relation over the state space is known as the reachability graph.

Definition 4 (*Reachability graph*) Let $\mathcal{N} = (P, T, f, m_0)$ be a Petri net. The reachability graph of $\mathcal{N}$ is the graph $\mathcal{RG}(\mathcal{N}) = (V_{\mathcal{N}}, E_{\mathcal{N}})$, where

- $V_{\mathcal{N}} := [m_0\rangle$ is the set of nodes,
- $E_{\mathcal{N}} := \{ (m, t, m') \mid m, m' \in [m_0\rangle, t \in T : m \xrightarrow{t} m' \}$ is the set of arcs. $\square$

The nodes of a reachability graph represent all possible markings of the net. The arcs in between are labelled by single transitions, the firing of which causes the related state change. The reachability graph gives us a finite automaton representation of all possible single step firing sequences. Consequently, concurrent behaviour is described by enumerating all interleaving firing sequences; so the reachability graph reflects the behaviour of the net according to the interleaving semantics.

Generally, reachability graphs tend to be huge. In the worst-case the state space grows faster than any primitive recursive function (Priese and Wimmel 2003). In our case, the size of the state space depends on the total number of vertices and the number of vertices to be blocked. The state space may explode for DNA walker models because all paths have to be generated which a walker can take. Moreover, the blocking and its failure introduces concurrency, which is analysed by considering all interleaving sequences of the transitions generated by unfolding the coloured transitions *block* and *fail*.

If we succeed in constructing the complete reachability graph, we are able to decide behavioural Petri net properties. We recall the most important ones, which include the three orthogonal behavioural properties—boundedness, reversibility, and liveness.

- A Petri net is *k-bounded* iff there is no node in the reachability graph with a token number larger than k in any place.
- A Petri net is *reversible* iff the reachability graph is strongly connected.
- A Petri net is *free of dead states* iff the reachability graph does not contain terminal nodes, i.e., nodes without outgoing arcs.
- In order to decide *liveness*, the reachability graph has to be partitioned into strongly connected components (SCC), i.e., maximal sets of strongly connected nodes. A SCC is called terminal if no other SCC is reachable in the partitioned graph. A transition is *live* iff it is included in all terminal SCCs of the partitioned reachability graph. A Petri net is live iff this holds for all transitions.

Our Petri nets are by construction:

- *2-bounded*: an anchorage can be unblocked (1 token), and host the walker (1 token) at the same time; no more moving tokens do exist;
- *not reversible*: which is an immediate consequence of the "burnt-bridges" scenario, causing acyclic reachability graphs;
- generally, *not free of dead states*: a walker can be trapped in a non-final vertex without any neighbouring vertex free to be visited;
- *not live*: while all transition can occur once in some behaviour, none of them will ever have a chance to fire twice in the "burnt-bridges" scenario.

These behavioural properties obviously depend on the applied execution semantics for the given DSD graph, but they are shared by all instances following the same template. For our execution semantics on hand, they coincide with our expectations.

For a given net, we determine these properties by help of Charlie (Heiner et al. 2015) or Marcie (Heiner et al. 2013). Charlie provides a traditional implementation, which works fine up to about 500,000 states (on current computer technique), while Marcie applies symbolic data structures, which substantially postpone the situation where the size of the state space exceeds the available memory.

Having validated our qualitative Petri nets, we are ready for the next step—the stochastic analysis. Standard stochastic Petri nets fulfilling the Markov property share the reachability graph with its underlying qualitative Petri net. Thus, all qualitative properties are still valid, but their interpretation can be refined by taking probabilities into consideration.

Our application scenario requires immediate transitions, which brings us to Generalised Stochastic Petri Nets (GSPN). Immediate transitions always fire with highest priority. With other words, if an immediate transition and a stochastic transition are concurrently enabled, then in the stochastic setting, only one firing sequence is considered (with the immediate transition firing first), while in the qualitative, time-free setting two firing sequences are considered (immediate—stochastic, stochastic—immediate). Consequently, the reachability graph induced by a GSPN is generally a proper subgraph of its underlying qualitative Petri net, which in turn means that a property relying on a given path in the reachability graph may not hold anymore in a specific sub-graph. For example, a dead state, reachable in the qualitative Petri net, is not necessarily reachable in the stochastic setting. In contrast, if the qualitative Petri net is free of dead states, then this holds for the GSPN as well.

To clarify the situation, we need to deploy stochastic analysis techniques, discussed in the next section.

4.2 Stochastic analysis

We analyse the probabilistic behaviour of DNA walker by means of simulative model checking (Rohr 2017). We start with recalling some properties defined in Dannenberg (2016), before extending the analysis by additional properties.

First, we investigate the transient behaviour of the walker circuits, e.g., how likely it is to have reached some state at a certain time point. This can be achieved by probabilistic model checking using the Continuous Stochastic Logic (CSL) (Baier et al. 2000). It is a stochastic adaptation of the Computation Tree Logic (CTL) (Clarke et al. 2001) to formulate properties over Continuous-time Markov Chains (CTMCs).

In the second part of our stochastic analysis we want to observe derived measures, also called reward, cost, observer, gain or bonus. Hence, we add an extra dimension to the CTMC and while moving on in time, it accumulates an output. In order to realise this, a reward structure $(\underline{\rho}, \boldsymbol{\iota})$ is added to the CTMC. The state reward function $\underline{\rho} : \mathcal{R} \to \mathbb{R}_0^+$ defines the rate at which reward $\underline{\rho}(s)$ is obtained in state s. That means a reward of $\tau \cdot \underline{\rho}(s)$ is earned, if the CTMC stays in state s for τ time units. The impulse reward function $\boldsymbol{\iota} : \mathcal{R} \times \mathcal{R} \to \mathbb{R}_0^+$ assigns to each transition t from state s to s' a reward $\boldsymbol{\iota}(s, s')$, i.e., a reward $\boldsymbol{\iota}(s, s')$ is acquired, if transition t fires. Having this, we can perform reward analysis by applying the CSL reward extensions (Kwiatkowska et al. 2007), e.g., what is the expected accumulated reward after some time. For example, rewards can be used to analyse the behaviour of transitions in terms of firing occurrences, which can be accumulated by a class of transitions.

Last but not least, we conduct performability analysis by use of the Continuous Stochastic Reward Logic (CSRL) (Haverkort et al. 2002), which is a superset of CSL. It combines the temporal logic formulas of CSL with a reward function, and the temporal logic operators have an additional reward interval. Now it is possible to reason about the probability to have reached some state at a certain time point and with respect to an interval on the accumulated reward.

4.2.1 Transient analysis

We consider four properties for transient analysis and use the same time bound $\tau = 12{,}000\,\mathrm{s}$ for all properties, this corresponds to 200 min.

The first property to check is the probability of the walker to have reached any of the FINAL anchorages at time point τ.

$$\mathcal{P}_{=?}[\mathrm{F}^{\tau,\tau}\,\mathtt{FINAL}] \tag{7}$$

The second property is the probability of the walker to have reached the CORRECT FINAL anchorage according to its input values at time point τ.

$$\mathcal{P}_{=?}[\mathrm{F}^{\tau,\tau}\,\mathtt{CORRECT}] \tag{8}$$

The third property is the probability of the walker to get stuck on its way in a dead state. The atomic proposition DEADLOCK describes the set of dead states.

$$\mathcal{P}_{=?}[\mathrm{F}^{\tau,\tau}\,\mathtt{DEADLOCK}] \tag{9}$$

The forth property is the conditional probability CONDITION of the walker to have reached the CORRECT FINAL anchorage according to its input values given that it has reached any of the FINAL anchorages at time point τ.

$$\begin{aligned}\mathcal{P}_{=?}(\mathtt{CONDITION}) &= \mathcal{P}_{=?}(\mathtt{CORRECT} \mid \mathtt{FINAL}) \\ &= \frac{\mathcal{P}_{=?}[\mathrm{F}^{\tau,\tau}\mathtt{CORRECT}]}{\mathcal{P}_{=?}[\mathrm{F}^{\tau,\tau}\mathtt{FINAL}]}\end{aligned} \tag{10}$$

4.2.2 Reward analysis

Observing derived measures requires the definition of a reward function that extends the CTMC by another dimension. One such measure is the accumulated number of steps n taken by a DNA walker on its way from the initial anchorage to a final anchorage. This is a discrete random variable, because the DNA walker randomly chooses steps of different length at each anchorage, except for the final anchorage. The number of steps a walker takes on its path can be computed by the impulse reward function $\boldsymbol{\iota}_{steps}$ defined by the following reward structure:

```
rewards[steps]{
  [stepShort] true : 1;
  [stepShortFinal] true : 1;
  [stepMedium] true : 1;
  [stepMediumFinal] true : 1;
  [stepLong] true : 1;
  [stepLongFinal] true : 1;
}
```

The impulse reward function $\boldsymbol{\iota}_{steps}$ is increased by one each time a stepping transition is fired and thus computes the number of steps a walker takes. The expected (average) number of steps of the DNA walker within $\tau = 12{,}000\,\mathrm{s}$ can be computed with the CSL formula

$$\mathcal{R}\{steps\}_{=?}\left[\mathrm{C}^{\leq\tau}\right]. \tag{11}$$

The classification of transitions reveals 12 different classes in the walker circuit under study, e.g., *track-short*, *fork-medium*, *leak-long*, etc.; and we are able to observe them directly. The number of steps of a certain class a DNA walker takes on its path can be computed by the impulse reward function $\boldsymbol{\iota}_{\ll\text{class}\gg}$ defined by the following reward structure:

```
rewards[<<class>>]{
  [transition_1] true : 1;
  ...
  [transition_n] true : 1;
}
```

We define one transition reward structure entry per classified transition identified with Algorithm 1. Thus the expected (average) number of classified steps the walker takes within $\tau = 12{,}000\,\mathrm{s}$ can be computed with the CSL formula

$$\mathcal{R}\{\ll\text{class}\gg\}_{=?}\left[\mathrm{C}^{\leq\tau}\right]. \tag{12}$$

4.2.3 Performability analysis

The reward analysis reveals the expected number of steps or leakage steps, but we are interested in the probabilities for different numbers of steps or leakage steps, too. Such probabilities are known as performability. We are able to compute the probability distribution of the related random variable deploying simulative model checking of CSRL in combination with impulse rewards (Rohr 2017). We compute the probability to reach a FINAL anchorage within $\tau = 12{,}000\,\mathrm{s}$ by taking exactly n steps with the following CSRL formula

$$\mathcal{P}\{steps\}_{=?}\left[\mathrm{F}^{\tau,\tau}_{n,n}\mathtt{FINAL}\right]. \tag{13}$$

Equation (13) defines the probability mass function (PMF) for the discrete probability distribution of the discrete random variable n.

The probability to reach a FINAL anchorage within $\tau = 12{,}000\,\mathrm{s}$ and by taking at most n steps can be computed by the CSRL formula

$$\mathcal{P}\{steps\}_{=?}\left[\mathrm{F}^{\tau,\tau}_{0,n}\mathtt{FINAL}\right]. \tag{14}$$

Equation (14) is the cumulative distribution function (CDF) for the discrete probability distribution of the discrete random variable n.

Exchanging the used reward function from $\boldsymbol{\iota}_{steps}$ to $\boldsymbol{\iota}_{\ll\text{class}\gg}$ allows us to compute the PDF

$$\mathcal{P}\{\ll\text{class}\gg\}_{=?}\left[\mathrm{F}^{\tau,\tau}_{n,n}\mathtt{FINAL}\right] \tag{15}$$

and the CDF

$$\mathcal{P}\{\ll\text{class}\gg\}_{=?}\left[\mathrm{F}^{\tau,\tau}_{0,n}\mathtt{FINAL}\right] \tag{16}$$

of the discrete random variable n for each class of steps, i.e., the number of steps of each class a DNA walker takes up to time point $\tau = 12{,}000\,\mathrm{s}$.

4.2.4 Case study

At first we compute the transient probabilities of Eqs. (7)–(10) for the two layouts. The results of the transient analysis are shown in Table 3. It turned out that both layouts perform equally well in the transient analysis. This leads to the supposition that achieving the reduced number of leak transitions is bought at the cost of a higher probability of reaching a dead state, due to the longer tracks.

Second we compute the expected number of steps according to Eq. (11) for the two layouts in Fig. 9. For the naive layout, the expected number of steps of one path

Table 3 Transient probabilities averaged over all possible input values for the naive layout in Fig. 9a and the optimised layout in Fig. 9b

	Final (%)	Correct (%)	Deadlock (%)	Condition (%)
Naive	68.66	62.93	9.64	91.66
Optimised	59.48	54.53	13.11	91.68

Table 4 Expected reward averaged over all possible input values for the naive layout in Fig. 9a

	Short	Medium	Long	Σ
Track	9.897	0.282	0.202	10.354
Fork	1.694	0.163	0.172	2.029
Join	1.208	0.139	0.168	1.515
Leak	0.483	0.189	0.311	0.983

Table 5 Expected reward averaged over all possible input values for the optimised layout in Fig. 9b

	Short	Medium	Long	Σ
Track	10.693	0.354	0.253	11.300
Fork	1.669	0.183	0.206	2.058
Join	1.286	0.170	0.170	1.626
Leak	0.114	0.135	0.195	0.444

averaged over all possible input values is 15.59, while it is for the optimised layout 16.02. The slightly higher value for the optimised layout can be explained by the longer tracks and the higher number of vertices.

Next we report the results for expectation analysis exploiting rewards for each class of steps according to Eq. (12). Tables 4 and 5 show the results, which are in-line with the static analysis in Tables 1 and 2. In the naive layout, a short distance leakage transition occurs in the average in every second run, while for the optimised layout, it occurs about in 1 run out of 10. When we look on leakage transitions in total, then we have in the naive layout in average about 1 leakage transition per run, which drops for the optimised layout below 1 leakage transition in every second run.

The last point in our stochastic analysis is the computation of the performability for the overall number of steps needed to reach a FINAL anchorage. Therefore, we check Eq. (13) for several values of n. Figures 10 and 11 show the probability distributions for both layouts. The second peak in Fig. 10 and its wider curve suggest a higher variability in the number of steps required to reach a FINAL anchorage in the naive layout, while the narrow, almost bell-shaped curve in Fig. 11 suggests a more constant

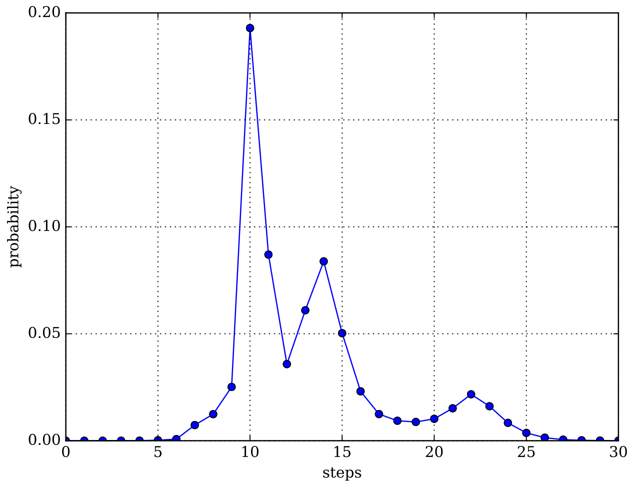


Fig. 10 Probability distribution of Eq. (13) computed for $n = (0, 30)$ and the naive layout

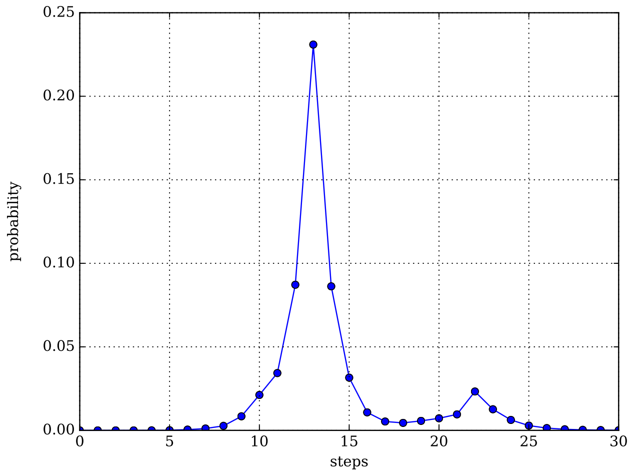


Fig. 11 Probability distribution of Eq. (13) computed for $n = (0, 30)$ and the optimised layout

number of steps in the optimised layout. This difference in the variability may be caused by the different numbers of leakage transitions.

4.2.5 Design trade-off

The results of structural and probabilistic analysis illustrate that we are facing an optimization problem (as discussed in terms of 'Design principles' in Dannenberg (2016)):

- *Objective 1*: reduce number of leakage transitions (as more leakage transitions increase the probability of getting the wrong result).
- *Objective 2*: reduce length of tracks (as longer tracks increase the deadlock probability).

Both objectives contradict each other: number of leakage transitions are reduced by increasing the length of tracks; and vice versa; so the challenge in circuit design is to find the right balance between both objectives. Compared with Dannenberg (2016), we are able to quantify objective 1 by a structural analysis. In future work, it would be interesting to explore if also objective 2 could be quantified by a structural analysis (e.g.,total length of shortest linear paths), to obtain a cheaper circuit design assessment.

5 Conclusions

In this paper we have reported a novel technique for the 2D modelling of DNA walker circuits using coloured stochastic Petri nets which enables functionality, topology and dimensionality all to be integrated in one two-dimensional model.

The move to coloured Petri nets not only brings a concise representation, but even more importantly a high degree in flexibility with respect to the topology of the anchorages and distance measures of the walker steps, both can be adjusted on the modelling level, no programming required. The anchorages to be blocked are automatically derived from the given input values (true or false) for all parameters of the given Boolean function, this is less error prone than setting them manually.

In terms of technology, the coloured approach enables construction of concise templated models which can be robustly expanded using standard mechanisms built into coloured Petri net tools. Other approaches require individually handcoded programs for translating circuit descriptions into SPN models (Barbot and Kwiatkowska 2015) or PRISM models (Dannenberg et al. 2015).

The concept of immediate transitions as in Petri nets does not exist in the PRISM language, thus it has to be approximated by very high transition rates, which increase the stiffness of the system and may cause numerical issues.

Automatic identification of leakage transitions is a relevant problem, which has remained unsolved until this research. We classify transitions between anchorages into short, medium and long distance categories, which enables a fine-grained analysis of the behaviour of the model. We present an algorithm for the automatic identification of leakage transitions, exploiting the unfolding of the coloured Petri net model. Leakage transitions are classified according to the used distance measure. Our algorithm is innovative, flexible and works for any kind of topologies and distance measures. The identification/classification of leakage transitions is merely a qualitative analysis technique and thus less expensive than CTMC based analysis. We show how advanced stochastic analysis including impulse rewards and performability analysis based on simulative CSRL model checking can be deployed to explore the stochastic behaviour of DNA circuit models. To the best of our knowledge this technique is not supported by other tools so far.

We illustrate the application of these techniques to compare the performance of two alternative layouts for an example DNA walker circuit. The results confirm that leakage can be reduced by employing a circuit layout topology that increases the distance between any two anchorages potentially permitting leakage transitions. An implication of this is that leakage reduction involves increasing the area of the circuit for a given number of anchorages. Since one goal of DNA circuit design is to minimise circuit area, the ability to identify leakage transitions is an important step in the process of optimising DNA circuit layouts, taking into account minimisation of both the computational error and area of circuits. Moreover, the use of multi-dimensional models opens the way to multi-dimensional model checking along the lines of Pârvu and Gilbert (2014).

Acknowledgements We would like to thank Benoît Barbot for providing the Petri net files for all case studies and the program source to generate them as used in Barbot and Kwiatkowska (2015).

Appendix

This appendix provides a full documentation of the toy example, see Fig. 5, employed in Sect. 2.3 to explain the use of coloured Petri nets for the specification of DNA walker circuits. All files can be found at http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Examples, and the software tools required at http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Software.

Workflow

Our workflow deploys the following tools:

- MARCIE (Heiner et al. 2013)—leakage detection, qualitative and stochastic analysis;
- Snoopy (Heiner et al. 2012)—graphical design, visualisation and simulation;
- Charlie (Heiner et al. 2015)—structural and qualitative analysis.

If you want to reproduce one of our examples or try your own ones, please follow these steps.

1. Write with your preferred text editor a CANDL specification. Start from the template provided, and adjust
   - the definition of constants in the group *block* according to the *literals*(*In*) of your DSD graph, see Definition 1, and the input values;
   - the colour set *Label* to include *literals*(*In*);
   - the colour function *Positions* to define all vertices of your DSD graph.

   That's it!
2. This CANDL specification can be processed by Marcie
   - to determine the leak transitions, categorised into short/medium/long distance transitions:

     ```
     marcie --net=file.candl --detect
     ```
   - to compute the state space, if possible, to determine dead states;
   - for any stochastic analysis as described in Sect. 4.2.
3. Alternatively, unfold the CANDL specification with

   ```
   andl_converter -f file.candl -u -o file.andl
   ```

   to obtain an ANDL file. This step is only mandatory if one wants to employ Charlie.
4. This ANDL file can be processed by our tools. Use
   - Sooopy—to obtain a graphical representation of the unfolded net: → file → import
   - Marcie—to determine the leak transitions, categorised into short/medium/long distance transitions:

     ```
     marcie --net=file.andl --detect
     ```

     Remark The Petri net provided as andl file has to be obtained by unfolding a coloured SPN following our template.
   - Charlie—for structural analysis;
   - Marcie—to compute the state space, if possible, to determine the dead states;
   - Marcie—for any stochastic analysis as described in Sect. 4.2.

Please see Marcie's Manual for more details (Schwarick et al. 2016).

Coloured SPN: CANDL specification

For reasons of completeness we provide a graphical representation of the coloured SPN template, made by reading the CANDL code into Snoopy; see Fig. 12. Many details are hidden in the graphics; the following CANDL code provides all details. See the Marcie manual (Schwarick et al. 2016) for a formal definition of the CANDL syntax.

```
colspn [toy]
{
constants:
all:
  int D1 = 7;
  int D2 = 8;
  int dS = 3; // short distance
  int dM = 5; // medium distance
  int dL = 8; // long distance
parameters:
  double rateShort = 0.009;
  double rateShortInit = 0.003;
  double rateShortFinal = 0.0009;
  double rateMedium = rateShort/50;
  double rateMediumInit = rateShortInit/50;
  double rateMediumFinal = rateShortFinal/50;
  double rateLong = rateShort/100;
  double rateLongInit = rateShortInit/100;
  double rateLongFinal = rateShortFinal/100;
  double rateLoop = 1e-09;
  double weightBlock = 0.7;
block:
  int m_X = 0;
  int m_NX = 0;

colorsets:
  Dot = {dot};
  CD1 = {1..D1};
  CD2 = {1..D2};
  enum Type = {INIT,FINAL,NORM,FORK,JOIN};
  enum Label= {E,T,F,X,NX};
  Circuit = PROD(CD1,CD2,Type,Label);

variables:
  CD1 : x1;
  CD2 : y1;
  CD1 : x2;
  CD2 : y2;
  Type : z1;
  Type : z2;
  Label : w1;
  Label : w2;

colorfunctions:

bool Positions(CD1 x, CD2 y, Type z, Label w) {
    (x=1 & y=1 & z=INIT & w=E) | (x=2 & y=3 & z=FORK & w=E)|
    (x=4 & y=3 & z=NORM & w=X) | (x=2 & y=5 & z=NORM & w=NX)|
    (x=6 & y=3 & z=FINAL & w=T)| (x=2 & y=7 & z=FINAL & w=F)
};

bool Blockage(Label w) {
    w!=E & w!=T & w!=F
};

// Rectilinear distance, Manhattan distance, L1 norm
CD1 RectilinearDistance(CD1 x1,CD2 y1,CD1 x2,CD2 y2) {abs(x1-x2) + abs(y1-y2)};

// Chessboard distance,  Chebyshev distance, Loo norm
CD1 ChessboardDistance(CD1 x1,CD2 y1,CD1 x2,CD2 y2) {max(abs(x1-x2), abs(y1-y2))};

bool NoSelfLoop (CD1 x1,CD2 y1,CD1 x2,CD2 y2)
    {(x1 != x2 | y1 != y2)};

bool ShortDistance (CD1 x1,CD2 y1,CD1 x2,CD2 y2)
    {RectilinearDistance(x1,y1,x2,y2) <= dS || ChessboardDistance(x1,y1,x2,y2) < dS};

bool MediumDistance (CD1 x1,CD2 y1,CD1 x2,CD2 y2)
    {(RectilinearDistance(x1,y1,x2,y2) > dS && ChessboardDistance(x1,y1,x2,y2) >= dS)
      && (RectilinearDistance(x1,y1,x2,y2) <= dM || ChessboardDistance(x1,y1,x2,y2) < dM)};

bool LongDistance (CD1 x1,CD2 y1,CD1 x2,CD2 y2)
    {(RectilinearDistance(x1,y1,x2,y2) > dM && ChessboardDistance(x1,y1,x2,y2) >= dM)
      && (RectilinearDistance(x1,y1,x2,y2) <= dL || ChessboardDistance(x1,y1,x2,y2) < dL)};

bool IsNeighbourShortD (CD1 x1,CD2 y1,Type z1,Label w1,CD1 x2,CD2 y2,Type z2,Label w2)
    {ShortDistance(x1,y1,x2,y2) && NoSelfLoop(x1,y1,x2,y2)
      && Positions(x1,y1,z1,w1) && Positions(x2,y2,z2,w2)};

bool IsNeighbourMediumD (CD1 x1,CD2 y1,Type z1,Label w1,CD1 x2,CD2 y2,Type z2,Label w2)
    {MediumDistance(x1,y1,x2,y2) && NoSelfLoop(x1,y1,x2,y2) && Positions(x1,y1,z1,w1)
      && Positions(x2,y2,z2,w2)};

bool IsNeighbourLongD (CD1 x1,CD2 y1,Type z1,Label w1,CD1 x2,CD2 y2,Type z2,Label w2)
    {LongDistance  (x1,y1,x2,y2) && NoSelfLoop(x1,y1,x2,y2) && Positions(x1,y1,z1,w1)
      && Positions(x2,y2,z2,w2)};

places:
discrete:
  Circuit A = [z1 = INIT]2‘(x1,y1,z1,w1) ++ [z1 != INIT]1‘(x1,y1,z1,w1);
  Circuit B = m[w1]‘(x1,y1,z1,w1);

transitions:
  stepShort
 {[IsNeighbourShortD(x1,y1,z1,w1,x2,y2,z2,w2) && z1 != FINAL && z2 != FINAL]}
    :
    : [A - {2‘(x1,y1,z1,w1)++1‘(x2,y2,z2,w2)}] & [A + {2‘(x2,y2,z2,w2)}]
    : [z1 = INIT]rateShortInit ++ [z1 != INIT]rateShort
    ;
  stepShortFinal
 {[IsNeighbourShortD(x1,y1,z1,w1,x2,y2,z2,w2) && z1 != FINAL && z2 = FINAL]}
    :
    : [A - {2‘(x1,y1,z1,w1)++1‘(x2,y2,z2,w2)}] & [A + {2‘(x2,y2,z2,w2)}]
    : rateShortFinal
    ;
  stepMedium
 {[IsNeighbourMediumD(x1,y1,z1,w1,x2,y2,z2,w2) && z1 != FINAL && z2 != FINAL]}
    :
    : [A - {2‘(x1,y1,z1,w1)++1‘(x2,y2,z2,w2)}] & [A + {2‘(x2,y2,z2,w2)}]
    : [z1 = INIT]rateMediumInit ++ [z1 != INIT]rateMedium
    ;
  stepMediumFinal
 {[IsNeighbourMediumD(x1,y1,z1,w1,x2,y2,z2,w2) && z1 != FINAL && z2 = FINAL]}
    :
    : [A - {2‘(x1,y1,z1,w1)++1‘(x2,y2,z2,w2)}] & [A + {2‘(x2,y2,z2,w2)}]
    : rateMediumFinal
    ;
  stepLong
 {[IsNeighbourLongD(x1,y1,z1,w1,x2,y2,z2,w2) && z1 != FINAL && z2 != FINAL]}
    :
    : [A - {2‘(x1,y1,z1,w1)++1‘(x2,y2,z2,w2)}] & [A + {2‘(x2,y2,z2,w2)}]
    : [z1 = INIT]rateLongInit ++ [z1 != INIT]rateLong
    ;
  stepLongFinal
 {[IsNeighbourLongD(x1,y1,z1,w1,x2,y2,z2,w2) && z1 != FINAL && z2 = FINAL]}
    :
    : [A - {2‘(x1,y1,z1,w1)++1‘(x2,y2,z2,w2)}] & [A + {2‘(x2,y2,z2,w2)}]
    : rateLongFinal
    ;
  loop
 {[Positions(x1,y1,z1,w1) && z1 = FINAL]}
    :
    : [A - {2‘(x1,y1,z1,w1)}] & [A + {2‘(x1,y1,z1,w1)}]
    : rateLoop
    ;

immediate:
  block
  {[Blockage(w1) && Positions(x1,y1,z1,w1)]}
    :
    : [A - {(x1,y1,z1,w1)}] & [B - {(x1,y1,z1,w1)}]
    : weightBlock
    ;
  fail
  {[Blockage(w1) && Positions(x1,y1,z1,w1)]}
    :
    : [B - {(x1,y1,z1,w1)}]
    : 1-weightBlock
    ;
}
// end colspn [toy]
```

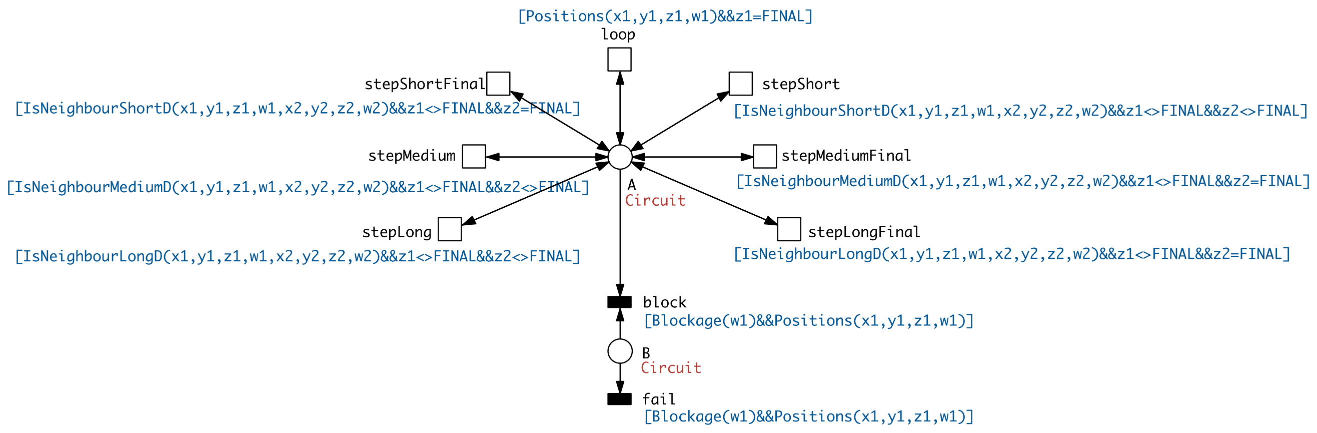

[Positions(x1,y1,z1,w1)&&z1=FINAL]
loop
stepShortFinal
[IsNeighbourShortD(x1,y1,z1,w1,x2,y2,z2,w2)&&z1<>FINAL&&z2=FINAL]
stepShort
[IsNeighbourShortD(x1,y1,z1,w1,x2,y2,z2,w2)&&z1<>FINAL&&z2<>FINAL]
stepMedium
[IsNeighbourMediumD(x1,y1,z1,w1,x2,y2,z2,w2)&&z1<>FINAL&&z2<>FINAL]
stepMediumFinal
[IsNeighbourMediumD(x1,y1,z1,w1,x2,y2,z2,w2)&&z1<>FINAL&&z2=FINAL]
A
Circuit
stepLong
[IsNeighbourLongD(x1,y1,z1,w1,x2,y2,z2,w2)&&z1<>FINAL&&z2<>FINAL]
stepLongFinal
[IsNeighbourLongD(x1,y1,z1,w1,x2,y2,z2,w2)&&z1<>FINAL&&z2=FINAL]
block
[Blockage(w1)&&Positions(x1,y1,z1,w1)]
B
Circuit
fail
[Blockage(w1)&&Positions(x1,y1,z1,w1)]

Fig. 12 Graphical representation of the coloured SPN; many details not shown, e.g., arc inscriptions, initial marking, transition rates; see CANDL code for all details

Generated SPN: leakage transitions

```
// marcie --net=toy.candl --detect

place labels:
A_1_1_INIT_E = 1
A_2_3_FORK_E = 1
A_2_5_NORM_NX = 2
A_2_7_FINAL_F = 2
A_4_3_NORM_X = 3
A_6_3_FINAL_T = 3
B_2_5_NORM_NX = 0
B_4_3_NORM_X = 0

transition types: // _a_b_c_d:  (d,b) -> (c,a)
stepShort_1_3_1_2_0_0_3_0 = track
stepShort_3_1_2_1_0_0_0_3 = track

stepShort_3_3_2_4_3_0_2_3 = fork
stepShort_3_3_4_2_0_3_3_2 = fork
stepShort_3_5_2_2_4_0_2_3 = fork
stepShort_5_3_2_2_0_4_3_2 = fork

stepShort_3_5_4_2_4_3_2_2 = leak
stepShort_5_3_2_4_3_4_2_2 = leak
stepShortFinal_3_3_6_4_3_1_2_1 = track
stepShortFinal_7_5_2_2_4_2_2_1 = track

stepMedium_1_3_1_4_3_0_2_0 = fork
stepMedium_3_1_4_1_0_3_0_2 = fork
stepMedium_1_5_1_2_4_0_2_0 = fork
stepMedium_5_1_2_1_0_4_0_2 = fork

stepMediumFinal_3_3_6_2_0_1_3_1 = fork
stepMediumFinal_7_3_2_2_0_2_3_1 = fork

stepMediumFinal_7_3_2_4_3_2_2_1 = leak
stepMediumFinal_3_5_6_2_4_1_2_1 = leak

stepLongFinal_3_1_6_1_0_1_0_1 = fork
stepLongFinal_7_1_2_1_0_2_0_1 = fork

loop_1_6_3_1 = final
loop_2_2_7_1 = final
block_3_4_3_2 = block
block_4_2_5_2 = block
fail_3_4_3_2 = block
fail_4_2_5_2 = block
place statistics:
label = count:
0 = 2
1 = 2
2 = 2
3 = 2

transition statistics:
type:        step(count)
track:        short(4)
fork:        short(4)        medium(6)        long(2)
leak:        short(2)        medium(2)

block:        unknown(4)
final:        unknown(2)
```

Generated SPN: ANDL specification

Size of the generated file, as reported by Marcie: |P|=8, |T|=26, |A|=70.

```
spn  [toy]
{
constants:
all:
  int D1 = 7;
  int D2 = 8;
  int dS = 3;
  int dM = 5;
  int dL = 8;
parameters:
  double rateShort = 0.009;
  double rateShortInit = 0.003;
  double rateShortFinal = 0.0009;
  double rateMedium = rateShort/50;
  double rateMediumInit = rateShortInit/50;
  double rateMediumFinal = rateShortFinal/50;
  double rateLong = rateShort/100;
  double rateLongInit = rateShortInit/100;
  double rateLongFinal = rateShortFinal/100;
  double rateLoop = 1e-09;
  double weightBlock = 0.7;
block:
  int m_X = 0;
  int m_NX = 0;

places:
discrete:
  A_1_1_INIT_E = 2;
  A_2_3_FORK_E = 1;
  A_2_5_NORM_NX = 1;
  A_2_7_FINAL_F = 1;
  A_4_3_NORM_X = 1;
  A_6_3_FINAL_T = 1;
  B_2_5_NORM_NX = m_NX;
  B_4_3_NORM_X = m_X;

transitions:
stochastic:
  stepShort_1_3_1_2_0_0_3_0
    :
    : [A_2_3_FORK_E - 2] & [A_1_1_INIT_E - 1] & [A_1_1_INIT_E + 2]
    : rateShort
    ;
  stepShort_3_1_2_1_0_0_0_3
    :
    : [A_1_1_INIT_E - 2] & [A_2_3_FORK_E - 1] & [A_2_3_FORK_E + 2]
    : rateShortInit
    ;
  stepShort_3_3_2_4_3_0_2_3
    :
    : [A_4_3_NORM_X - 2] & [A_2_3_FORK_E - 1] & [A_2_3_FORK_E + 2]
    : rateShort
    ;
  stepShort_3_3_4_2_0_3_3_2
    :
    : [A_2_3_FORK_E - 2] & [A_4_3_NORM_X - 1] & [A_4_3_NORM_X + 2]
    : rateShort
    ;
  stepShort_3_5_2_2_4_0_2_3
    :
    : [A_2_5_NORM_NX - 2] & [A_2_3_FORK_E - 1] & [A_2_3_FORK_E + 2]
    : rateShort
    ;
  stepShort_3_5_4_2_4_3_2_2
    :
    : [A_2_5_NORM_NX - 2] & [A_4_3_NORM_X - 1] & [A_4_3_NORM_X + 2]
    : rateShort
    ;
  stepShort_5_3_2_2_0_4_3_2
    :
    : [A_2_3_FORK_E - 2] & [A_2_5_NORM_NX - 1] & [A_2_5_NORM_NX + 2]
    : rateShort
    ;
  stepShort_5_3_2_4_3_4_2_2
    :
    : [A_4_3_NORM_X - 2] & [A_2_5_NORM_NX - 1] & [A_2_5_NORM_NX + 2]
    : rateShort
    ;
  stepShortFinal_3_3_6_4_3_1_2_1
    :
    : [A_4_3_NORM_X - 2] & [A_6_3_FINAL_T - 1] & [A_6_3_FINAL_T + 2]
    : rateShortFinal
    ;
  stepShortFinal_7_5_2_2_4_2_2_1
    :
    : [A_2_5_NORM_NX - 2] & [A_2_7_FINAL_F - 1] & [A_2_7_FINAL_F + 2]
    : rateShortFinal
    ;
  stepMedium_1_3_1_4_3_0_2_0
    :
    : [A_4_3_NORM_X - 2] & [A_1_1_INIT_E - 1] & [A_1_1_INIT_E + 2]
    : rateMedium
    ;
  stepMedium_1_5_1_2_4_0_2_0
    :
    : [A_2_5_NORM_NX - 2] & [A_1_1_INIT_E - 1] & [A_1_1_INIT_E + 2]
    : rateMedium
    ;
  stepMedium_3_1_4_1_0_3_0_2
    :
    : [A_1_1_INIT_E - 2] & [A_4_3_NORM_X - 1] & [A_4_3_NORM_X + 2]
    : rateMediumInit
    ;
  stepMedium_5_1_2_1_0_4_0_2
    :
    : [A_1_1_INIT_E - 2] & [A_2_5_NORM_NX - 1] & [A_2_5_NORM_NX + 2]
    : rateMediumInit
    ;
  stepMediumFinal_3_3_6_2_0_1_3_1
    :
    : [A_2_3_FORK_E - 2] & [A_6_3_FINAL_T - 1] & [A_6_3_FINAL_T + 2]
    : rateMediumFinal
    ;
  stepMediumFinal_3_5_6_2_4_1_2_1
    :
    : [A_2_5_NORM_NX - 2] & [A_6_3_FINAL_T - 1] & [A_6_3_FINAL_T + 2]
    : rateMediumFinal
    ;
  stepMediumFinal_7_3_2_2_0_2_3_1
    :
    : [A_2_3_FORK_E - 2] & [A_2_7_FINAL_F - 1] & [A_2_7_FINAL_F + 2]
    : rateMediumFinal
    ;
  stepMediumFinal_7_3_2_4_3_2_2_1
    :
    : [A_4_3_NORM_X - 2] & [A_2_7_FINAL_F - 1] & [A_2_7_FINAL_F + 2]
    : rateMediumFinal
    ;
  stepLongFinal_3_1_6_1_0_1_0_1
    :
    : [A_1_1_INIT_E - 2] & [A_6_3_FINAL_T - 1] & [A_6_3_FINAL_T + 2]
    : rateLongFinal
    ;
  stepLongFinal_7_1_2_1_0_2_0_1
    :
    : [A_1_1_INIT_E - 2] & [A_2_7_FINAL_F - 1] & [A_2_7_FINAL_F + 2]
    : rateLongFinal
    ;
  loop_1_6_3_1
    :
    : [A_6_3_FINAL_T - 2] & [A_6_3_FINAL_T + 2]
    : rateLoop
    ;
  loop_2_2_7_1
    :
    : [A_2_7_FINAL_F - 2] & [A_2_7_FINAL_F + 2]
    : rateLoop
    ;
immediate:
  block_3_4_3_2
    :
    : [A_4_3_NORM_X - 1] & [B_4_3_NORM_X - 1]
    : weightBlock
    ;
  block_4_2_5_2
    :
    : [A_2_5_NORM_NX - 1] & [B_2_5_NORM_NX - 1]
    : weightBlock
    ;
  fail_3_4_3_2
    :
    : [B_4_3_NORM_X - 1]
    : 1-weightBlock
    ;
  fail_4_2_5_2
    :
    : [B_2_5_NORM_NX - 1]
    : 1-weightBlock
    ;

}// end spn [toy]
```

Generated SPN: graphics

The graphical representations shown in Figs. 13 and 14 take advantage of a special feature supported by Snoopy allowing for hierarchical Petri net design where *macro transitions* (represented by two centric squares) stand for subnets. Here we use these transitions to hide the details of the Petri net representation for an undirected step according to the "burnt-bridges" scenario, see Fig. 3b.

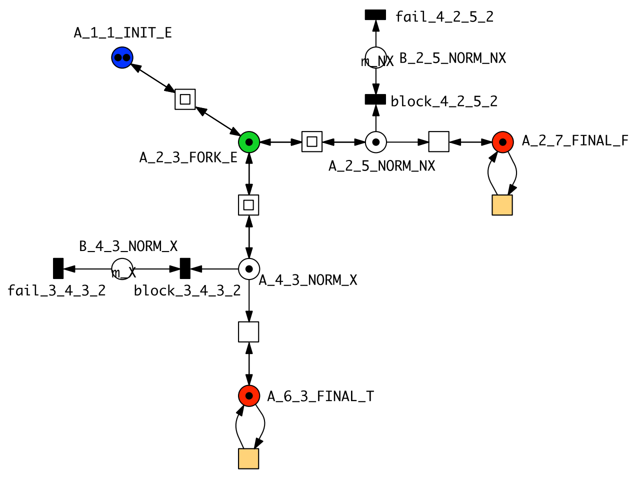


Fig. 13 Petri net generated by unfolding a CANDL specification containing only short distance steps; transition names and arc weights not shown. Colour code: places—as used in Fig. 5, yellow transitions—loop, added for technical reasons, see Sect. 4.1. (Color figure online)

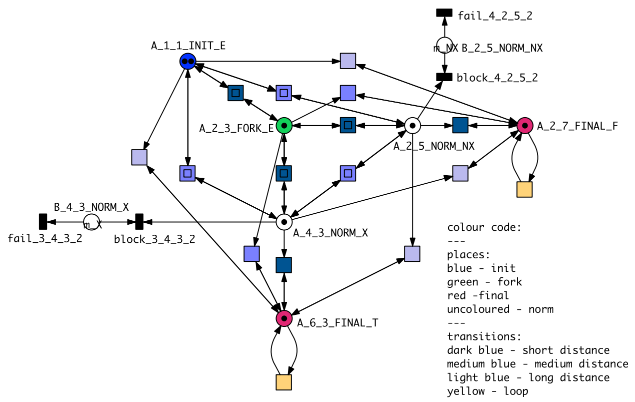


Fig. 14 Petri net generated by unfolding the CANDL specification; transition names and arc weights not shown

References

Baier C, Haverkort B, Hermanns H, Katoen JP (2000)Model checking continuous-time Markov chains by transient analysis. In: Proc. CAV 2000. LNCS 1855. Springer, pp 358–372

Barbot B, Kwiatkowska M (2015) On quantitative modelling and verification of DNA walker circuits using stochastic Petri nets. In: Proc. PETRI NETS 2015. LNCS, vol 9115. Springer, pp 1–32 (2015)

Bath J, Green SJ, Turberfield AJ (2005) A free-running DNA motor powered by a nicking enzyme. Angew Chem 117(28):4432–4435

Blätke M, Heiner M, Marwan W (2015) BioModel engineering with Petri nets. Elsevier, Amsterdam, pp 141–193

Boemo M, Lucas A, Turberfield A, Cardelli L (2016) The formal language and design principles of autonomous DNA walker circuits. ACS Synth Biol 5:878–884

Chen YJ, Dalchau N, Srinivas N, Phillips A, Cardelli L, Soloveichik D, Seelig G (2013) Programmable chemical controllers made from DNA. Nat Nanotechnol 8(10):755–762

Clarke EM, Grumberg O, Peled D (2001) Model checking. MIT Press, Cambridge

Cormen T, Leiserson C, Rivest R, Stein C (2001) Introduction to algorithms, 2nd edn. MIT Press, Cambridge

Dannenberg F (2016) Modelling and verification for DNA nanotechnology. PhD thesis, Oxford University, Balliol College

Dannenberg F, Kwiatkowska M, Thachuk C, Turberfield A (2015) DNA walker circuits: computational potential, design, and verification. Nat Comput 14(2):195–211

Gilbert D, Heiner M, Lehrack S (2007) A unifying framework for modelling and analysing biochemical pathways using Petri nets. In: Proc. CMSB 2007. LNCS/LNBI, vol 4695. Springer, pp 200–216

Gilbert D, Heiner M, Liu F, Saunders N (2013) Colouring space—a coloured framework for spatial modelling in systems biology. In: Colom J, Desel J (eds) Proc. PETRI NETS 2013, LNCS, vol 7927. Springer, pp 230–249

Haverkort B, Cloth L, Hermanns H, Katoen JP, Baier C (2002) Model checking performability properties. IEEE Computer Society Press, Washington, DC, pp 103–112

Heiner M, Gilbert D (2013) Biomodel engineering for multiscale systems biology. Prog Biophys Mol Biol 111(2–3):119128. https://doi.org/10.1016/j.pbiomolbio.2012.10.001

Heiner M, Gilbert D, Donaldson R (2008) Petri nets in systems and synthetic biology. In: SFM. LNCS, vol 5016. Springer, pp 215–264

Heiner M, Lehrack S, Gilbert D, Marwan W (2009) Extended stochastic Petri nets for model-based design of wetlab experiments. In: LNCS/LNBI, vol 5750. Springer, pp 138–163

Heiner M, Herajy M, Liu F, Rohr C, Schwarick M (2012) Snoopy—a unifying Petri net tool. In: Proc. PETRI NETS 2012, LNCS, vol 7347. Springer, pp 398–407

Heiner M, Rohr C, Schwarick M (2013) MARCIE—model checking and reachability analysis done efficiently. In: Colom J, Desel J (eds) Proc. PETRI NETS 2013, LNCS, vol 7927. Springer, pp 389–399

Heiner M, Schwarick M, Wegener J (2015) Charlie—an extensible Petri net analysis tool. In: Proc. Petri nets, LNCS, vol 9115. Springer, pp 200–211

Jung J, Hyun D, Shin Y (2015) Physical synthesis of DNA circuits with spatially localized gates. In: 2015 33rd IEEE international conference on computer design (ICCD). IEEE, pp 259–265

Kwiatkowska M, Norman G, Parker D (2007) Stochastic model checking. In: SFM. LNCS, vol 4486. Springer, pp 220–270

Lakin M, Youssef S, Polo F, Emmott S, Phillips A (2011) Visual DSD: a design and analysis tool for DNA strand displacement systems. Bioinformatics 27(22):3211–3213

Liu F, Heiner M, Rohr C (2012) Manual for colored Petri nets in Snoopy

Pârvu O, Gilbert D (2014) Automatic validation of computational models using pseudo-3D spatio-temporal model checking. BMC Syst Biol 8(124):1–24

Phillips A, Cardelli L (2009) A programming language for composable DNA circuits. J R Soc Interface 6(Suppl 4):S419–S436

Priese L, Wimmel H (2003) Theoretical informatics—Petri nets. Springer, Berlin (in German)

Rohr C (2017) Simulative analysis of coloured extended stochastic Petri nets. PhD thesis, BTU Cottbus, Department of Computer Science (submitted)

Schwarick M, Rohr C, Heiner M (2016) MARCIE manual. BTU Cottbus, CS Institute

Wickham S, Endo M, Katsuda Y, Hidaka K, Bath J, Sugiyama H, Turberfield A (2011) Direct observation of stepwise movement of a synthetic molecular transporter. Nat Nanotechnol 6(3):166–169

Wickham S, Bath J, Katsuda Y, Endo M, Hidaka K, Sugiyama H, Turberfield A (2012) A DNA-based molecular motor that can navigate a network of tracks. Nat Nanotechnol 7(3):169–173

Yin P, Yan H, Daniell XG, Turberfield AJ, Reif JH (2004) A unidirectional DNA walker that moves autonomously along a track. Angew Chem Int Ed 43(37):4906–4911