Towards the Algorithmic Molecular Self-Assembly of Fractals by Cotranscriptional Folding*

Yusei Masuda, Shinnosuke Seki**, and Yuki Ubukata

Department of Computer and Network Engineering, The University of Electro-Communications, 1-5-1, Chofugaoka, Chofu, Tokyo, 1828585, Japan
s.seki@uec.ac.jp

Abstract. RNA cotranscriptional folding has been just experimentally proven capable of self-assembling a rectangular tile at nanoscale *in vivo* (RNA origami). We initiate the theoretical study on the algorithmic self-assembly of shapes by cotranscriptional folding using a novel computational model called the oritatami system. We propose an oritatami system that folds into an arbitrary finite portion of the Heighway dragon fractal, also-known as the paperfolding sequence $P = \mathrm{RRLRRLLR}\cdots$. The i-th element of P can be obtained by feeding i in binary to a 4-state DFA with output (DFAO). We implement this DFAO and a bit-sequence bifurcator as modules of oritatami system. Combining them with a known binary counter yields the proposed system.

1 Introduction

An RNA sequence, over nucleotides of four kinds `A`, `C`, `G`, `U`, is synthesized (*transcribed*) from its template DNA sequence over `A`, `C`, `G`, `T` nucleotide by nucleotide by an RNA polymerase (RNAP) enzyme according to the one-to-one mapping $\mathtt{A} \to \mathtt{U}$, $\mathtt{C} \to \mathtt{G}$, $\mathtt{G} \to \mathtt{C}$, and $\mathtt{T} \to \mathtt{A}$ (for details, see, e.g., [2]). The yield, called *transcript*, starts folding immediately after it emerges from RNAP. This is the *cotranscriptional folding* (see Fig. 1). Geary, Rothemund, and Andersen have recently demonstrated the capability of cotranscriptional folding to self-assemble an RNA molecule of an intended shape at nano-scale [7]. They actually proposed an architecture of a DNA sequence whose transcript folds cotranscriptionally into an RNA tile of specific rectangular shape highly likely *in vitro*.

Algorithms and computation are fundamental to molecular self-assembly as illustrated in an enormous success of their use in DNA tile self-assembly (see, e.g., [4, 12, 15] and references therein). The Sierpinski triangle fractal was algorithmically self-assembled even *in vitro* from coalescence of DNA tiles that compute XOR [13]. Cotranscriptional folding exhibits highly sophisticated computational

* This work is in part supported by JST Program to Disseminate Tenure Tracking System, MEXT, Japan, No. 6F36 and by JSPS KAKENHI Grant-in-Aid for Young Scientists (A) No. 16H05854 to S. S.

** Corresponding author

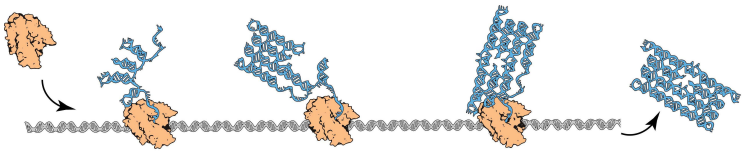

Fig. 1. RNA cotranscriptional folding. An RNA polymerase attaches to a template DNA sequence (gray spiral), scans it through, and synthesizes its RNA copy. The RNA sequence begins to fold upon itself immediately as it emerges from polymerase.

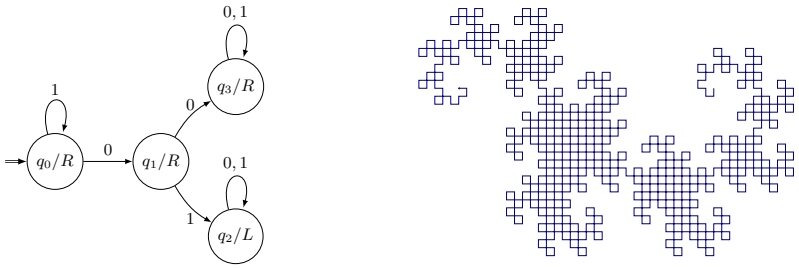

Fig. 2. (Left) DFAO to output the direction (L/R) of i-th turn of the Heighway dragon given $i \geq 0$ in binary from the LSB. (Right) The first $2^{10}-1$ turns of the dragon.

and algorithmic behaviors as well. Indeed, fluoride riboswitches in *Bacillus cereus* bacteria cotranscriptionally fold into a terminator stem or do not, in order to regulate gene expression [14]. This is just one example but should be enough to signify both the context-sensitivity of cotranscriptional folding and shapes thus self-assembled. Geary et al. have proved the capability of context-sensitivity to count in binary using a novel mathematical model of cotranscriptional folding called *oritatami system* (abbreviated as OS) [6].

We shall initiate theoretical study on algorithmic self-assembly of shapes by cotranscriptional folding using oritatami system. Sierpinski triangle would allow our study to borrow rich insights from the DNA tile self-assembly. However, in order to cut directly to the heart of algorithmic self-assembly by cotranscriptional folding, shapes of choice should be traversable somehow algorithmically. One such way is to feed a turtle program (see [1]) with an *automatic sequence* as commands (drawing a line segment, rotation, etc.), whose i-th bit can be obtained by giving i in binary from the least significant bit (LSB) to one DFA with output (DFAO) [3]. Shapes thus describable include the Heighway dragon [3] and von Koch curve [10]. A DFAO for the Heighway dragon is illustrated in Fig. 2. It outputs the following sequence, given $i = 0, 1, 2, \ldots$ in binary:

$$P = \mathrm{RRLRRLLRRRLLRLLRRRLRRLLLRRLLRLL}\cdots.$$

(The notation P is after its appellative *paperfolding sequence* [3].) For instance, given $i = 2$ in binary from the LSB as 01, the DFAO transitions as $q_0 \to q_1 \to q_2$ and hence $P[2] = \mathrm{L}$. A turtle should interpret an L (resp. R) as "move forward by

unit distance and turn left (resp. right) 90 degrees." Any portion of the dragon can be represented by a factor of P; for instance, Fig. 2 (Right) depicts the portion $P[0..1022]$, i.e., the first $2^{10} - 1$ turns of the dragon.

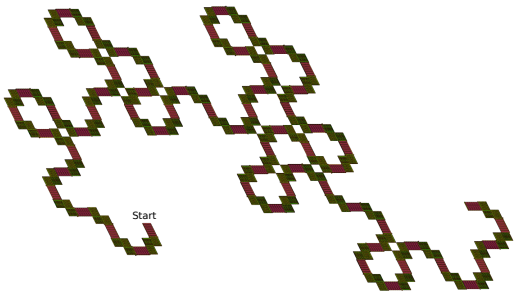

Fig. 3. The portion $P[0..62]$ of the Heighway dragon folded by the proposed oritatami system.

In this paper, we propose a generic design of oritatami system for the algorithmic cotranscriptional folding of an arbitrary finite portion of the Heighway dragon. Fig. 3 shows the portion $P[0..62]$ thus folded (the dragon is slanted but this is because the OS operates on the triangular grid rather than on the square grid). The OS transcribes three modules: counter module, DFAO module, and turning module, in this order repeatedly. The counter module is a technical modification of the binary counter proposed in [6] so that it increments a given count i exactly by 1 while folding into a (red) line segment. At the end of the segment comes a DFAO module, which computes the turn direction $P[i]$ and propagates it along with the count i to the next module for turn. A (green) L-shaped block is the turning module. It is a concatenation of three bit-sequence bifurcators, each of which folds into a rhombus, bifurcates i leftward as well as rightward, and guides further folding according to the turning direction. The next counter module then again increments one of the bifurcated i and folds into the next line segment, and so on.

The generic design proves the next theorem (for terminologies, see Sect. 2).

Theorem 1. *For any finite portion $P[i..j]$ of the Heighway dragon, there exist a scaling factor $c \in \mathbb{N}^+$ and a cyclic deterministic oritatami system of delay 3 and arity 3 that weakly folds into the c-rhombus scaling of $P[i..j]$.*

A JavaScript program to run this OS is available at `https://wolves13.github.io`.

2 Preliminaries

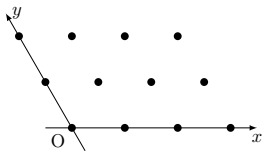

Fig. 4. Triangular grid graph $\mathbb{T}$ with the (x, y)-coordinate and the origin.

Let Σ be a set of types of abstract molecules, or *beads*, and Σ^* be the set of finite sequences of beads. A bead of type $a \in \Sigma$ is called an a-bead. Let $w = b_1 b_2 \cdots b_n \in \Sigma^*$ be a string of length n for some integer n and bead types $b_1, \ldots, b_n \in \Sigma$. The *length* of w is denoted by $|w|$, that is, $|w| = n$. For two indices i, j with $1 \leq i \leq j \leq n$, we let $w[i..j]$ refer to the subsequence $b_i b_{i+1} \cdots b_{j-1} b_j$; if $i = j$, then we simplify $w[i..i]$ as $w[i]$. For $k \geq 1$, $w[1..k]$ is called a *prefix* of w.

Oritatami systems fold their transcript, a sequence of beads, over the triangular grid graph $\mathbb{T} = (V, E)$ (see Figs. 4 and 5) cotranscriptionally based on hydrogen-bond-based interactions (*h-interactions* for short) which the system allow for between adjacent beads of particular types. When beads form an h-interaction, informally we say they are bound. For two points $p_1 = (x_1, y_1), p_2 = (x_2, y_2) \in V$, $\{p_1, p_2\} \in E$ if $|x_1 - x_2| = 1$ and $y_1 = y_2$ or $x_1 = x_2$ and $|y_1 - y_2| = 1$. A directed path $P = p_1 p_2 \cdots p_n$ in $\mathbb{T}$ is a sequence of *pairwise-distinct* points $p_1, p_2, \ldots, p_n \in V$ such that $\{p_i, p_{i+1}\} \in E$ for all $1 \leq i < n$. Its i-th point is referred to as $P[i]$. A *conformation* C is a triple (P, w, H) of a directed path P in $\mathbb{T}$, $w \in \Sigma^*$ of the same length as P, and a set of h-interactions $H \subseteq \{\{i, j\} \mid 1 \leq i, i+2 \leq j, \{P[i], P[j]\} \in E\}$. This is to be interpreted as the sequence w being folded in such a manner that its i-th bead $w[i]$ is placed on the i-th point $P[i]$ along the path and the i-th and j-th beads are bound if and only if $\{i, j\} \in H$. The condition $i + 2 \leq j$ represents the topological restriction that two consecutive beads along the path cannot be bound. A *rule set* $\mathcal{H} \subseteq \Sigma \times \Sigma$ is a symmetric relation over the set of pairs of bead types, that is, for all bead types $a, b \in \Sigma$, $(a, b) \in \mathcal{H}$ implies $(b, a) \in \mathcal{H}$. An h-interaction $\{i, j\} \in H$ is *valid with respect to* $\mathcal{H}$, or simply $\mathcal{H}$*-valid*, if $(w[i], w[j]) \in \mathcal{H}$. This conformation C is $\mathcal{H}$-valid if all of its h-interactions are $\mathcal{H}$-valid. For an integer $\alpha \geq 1$, C is *of arity* α if it contains a bead that forms α h-interactions and no bead of C forms more. By $\mathcal{C}_{\leq \alpha}$, we denote the set of all conformations of arity at most α.

Oritatami systems grow conformations by elongating them under their own rule set. Given a rule set $\mathcal{H}$ and an $\mathcal{H}$-valid finite conformation $C_1 = (P, w, H)$, we say that another conformation C_2 is an *elongation of* C_1 *by a bead* $b \in \Sigma$, written as $C_1 \xrightarrow{\mathcal{H}}_b C_2$, if $C_2 = (Pp, wb, H \cup H')$ for some point p not along the path P and set of h-interactions $H' \subseteq \left\{\{i, |w|+1\} \;\middle|\; 1 \leq i < |w|, \{P[i], p\} \in E, (w[i], b) \in \mathcal{H}\right\}$, which can be empty. Note that C_2 is also $\mathcal{H}$-valid. This operation is recursively extended to the elongation by a finite sequence of beads as: for any conformation C, $C \xrightarrow{\mathcal{H}}{}^*_\lambda C$; and for a finite sequence of beads $w \in \Sigma^*$ and a bead $b \in \Sigma$, a conformation C_1 is elongated to a conformation C_2 by wb, written as $C_1 \xrightarrow{\mathcal{H}}{}^*_{wb} C_2$, if there is a conformation C' that satisfies $C_1 \xrightarrow{\mathcal{H}}{}^*_w C'$ and $C' \xrightarrow{\mathcal{H}}_b C_2$.

A finite *oritatami system* (OS) is a 5-tuple $\Xi = (\mathcal{H}, \alpha, \delta, \sigma, w)$, where $\mathcal{H}$ is a rule set, α is an arity, $\delta \geq 1$ is a parameter called the *delay*, σ is an initial $\mathcal{H}$-valid conformation of arity α called the *seed*, upon which its finite *transcript* $w \in \Sigma^*$ is to be folded by stabilizing beads of w one at a time so as to minimize energy collaboratively with the succeeding $\delta - 1$ nascent beads. The energy of a conformation $C = (P, w, H)$, denoted by $\Delta G(C)$, is defined to be $-|H|$; the more h-interactions a conformation has, the more stable it gets. The set $\mathcal{F}(\Xi)$ of conformations *foldable* by this system is recursively defined as: the seed σ is in $\mathcal{F}(\Xi)$; and provided that an elongation C_i of σ by the prefix $w[1..i]$ be foldable (i.e., $C_0 = \sigma$), its further elongation C_{i+1} by the next bead $w[i+1]$ is foldable if

$$C_{i+1} \in \operatorname*{arg\,min}_{\substack{C \in \mathcal{C}_{\leq \alpha}\, s.t. \\ C_i \xrightarrow{\mathcal{H}}_{w[i+1]} C}} \min \left\{ \Delta G(C') \mid C \xrightarrow{\mathcal{H}}{}^*_{w[i+2 \ldots i+k]} C', k \leq \delta, C' \in \mathcal{C}_{\leq \alpha} \right\}. \quad (1)$$

We say that the bead $w[i+1]$ and the h-interactions it forms are *stabilized* according to C_{i+1}. Note that an arity-α OS cannot fold any conformation of arity larger than α. A conformation foldable by Ξ is *terminal* if none of its elongations is foldable by Ξ. The OS Ξ is *deterministic* if for all $i \geq 0$, there exists at most one C_{i+1} that satisfies (1). The deterministic OS is abbreviated as DOS. Thus, a DOS folds into a unique terminal conformation. An OS is *cyclic* if its transcript is of the form $u^i u_p$ for some $i \geq 2$ and a prefix u_p of u. The cyclic OS is considered to be one of the practical classes of OS because a periodic RNA transcript is likely to be transcribed out of a circular DNA sequence [5].

Let us provide an example of cyclic DOS that folds into a useful *glider* motif. Consider a delay-3 OS whose transcript w is a repetition of $a \bullet b'b \bullet a'$ and whose rule set is $\mathcal{H} = \{(a, a'), (b, b')\}$, making $\bullet$-beads inert. Its seed, colored in red in Fig. 5, can be elongated by the first three beads $w[1..3] = a \bullet b'$ in various ways, only three of which are shown in Fig. 5 (left). Only $w[3] = b'$ can form a new h-interaction, with the b in the seed (according to $\mathcal{H}$, the first a is also capable of binding, with a', but the sole a' around is just "too close"). For the b–b' binding, $w[1..3]$ must be folded as bolded in Fig. 5 (left). According to this most stable "bolded" elongation, the bead $w[1] = a$ is stabilized to the east of the previous bead. Then $w[4] = b$ is transcribed. We can easily check that no matter how $w[2..4]$ is folded, b'-beads around are either too far or too close for $w[4]$ to bind to. Hence, $w[4]$ cannot override the previous "decision" so that $w[2]$ is stabilized as bolded. $w[5]$ cannot override it, either, simply because it is inert. It is easily induced inductively that gliders of arbitrary "flight distance" can be folded.

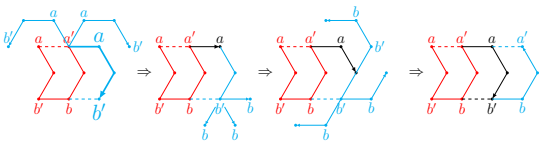

Fig. 5. Progression of a glider by distance 1.

Gliders also provide a medium to propagate 1-bit arbitrarily far as the position of their last beads, which is determined by the height (top or bottom) of the first bead and a propagating distance. For instance, the glider in Fig. 5 launches top and thus its last bead (the a') also comes top after traveling the distance 2. The OS we shall propose exploits this information-carrying capability.

Assume the (x, y)-coordinate over $\mathbb{T} = (V, E)$ as shown in Fig. 4. A *shape* S is a set of points in V. For an integer $c \geq 1$, let $Rhomb_c = \{(x, y) \in V \mid x, y \leq c\}$. Let $S' = \{(cx, cy) \mid (x, y) \in S\}$. The *c-rhombus scaling* of S, denoted by $\diamondsuit_c(S)$, is the union over all $\boldsymbol{p} \in S'$ of sets of points $Rhomb_c + \boldsymbol{p} = \{\boldsymbol{v} \in V \mid \boldsymbol{v} = \boldsymbol{r} + \boldsymbol{p}$ for some $\boldsymbol{r} \in Rhomb_c\}$. We say that an OS Ξ *weakly folds* (or "self-assembles") $\diamondsuit_c(S)$ if every terminal assembly of Ξ puts at least one bead in $Rhomb_c + \boldsymbol{p}$ for all $\boldsymbol{p} \in S'$ and no bead in $Rhomb_c + \boldsymbol{q}$ for all $\boldsymbol{q} \notin S'$.

3 Folding the n-bit Heighway dragon

We propose a generic design of DOS that allows us to fold an arbitrary finite portion $P[j_1..j_2]$ of the slanted Heighway dragon. Independently of j_1, j_2, both

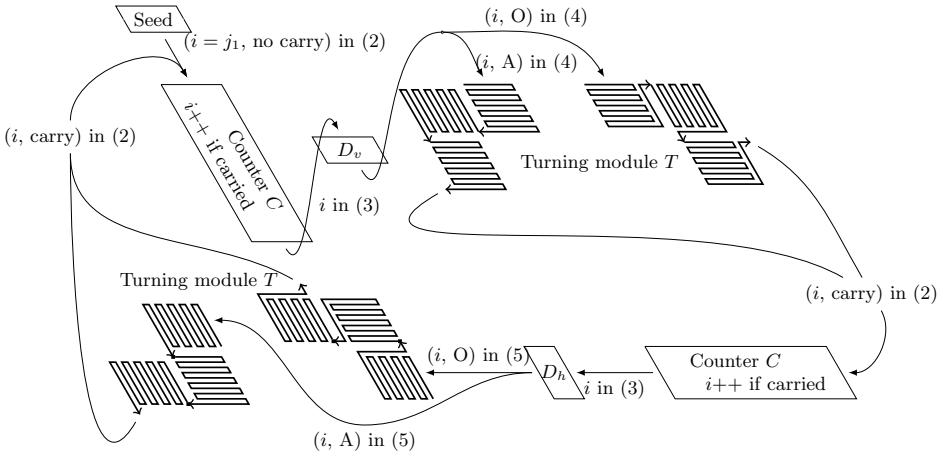

Fig. 6. Module automaton for the Heighway dragon $P[j_1..j_2]$. Transitions are labeled with the information propagated with its format.

delay and arity are set to 3 and 567 bead types[1] `1`, `2`, ..., `567` with a fixed rule set $\mathcal{H}$ are employed. The design also challenges to make the resulting DOS cyclic. Otherwise, one could simply implement left and right-turn modules and concatenate their copies according to the (non-periodic) sequence P. However, it is highly unlikely that an OS for the infinite Heighway dragon, if any, could adopt this approach in order to be describable by a finite mean. Such a "hardcoding" also runs counter to the spirit of algorithmic self-assembly.

Modularization is a semantic factorization of transcript into functional units. Functions of modules and transitions from one module to another are described in the form of module automaton (MA). The generic design employs the MA in Fig. 6. This MA yields the periodic transcript which repeats six modules of the following four types as CD_vTCD_hT:

- C is a *counter module*; it increments the count i (index of P), which is "initialized" to j_1 on the seed, by 1 and propagates it;
- D_v and D_h are a *DFAO module*; they compute $P[i]$ and interpret it properly (this issue of interpretation shall be discussed shortly);
- T is a *turning module*; it makes a turn according to the interpretation.

The first C and D_v modules fold into a vertical line segment, while the second C and D_h fold into the next line segment, which is guaranteed to be horizontal since vertical and horizontal segments alternate on the Heighway dragon. The DFAO modules D_v, D_h differ from each other only in their way to interpret their

[1] Some of the bead types might be saved but not easily due to the NP-hardness of minimizing the number of bead types [8].

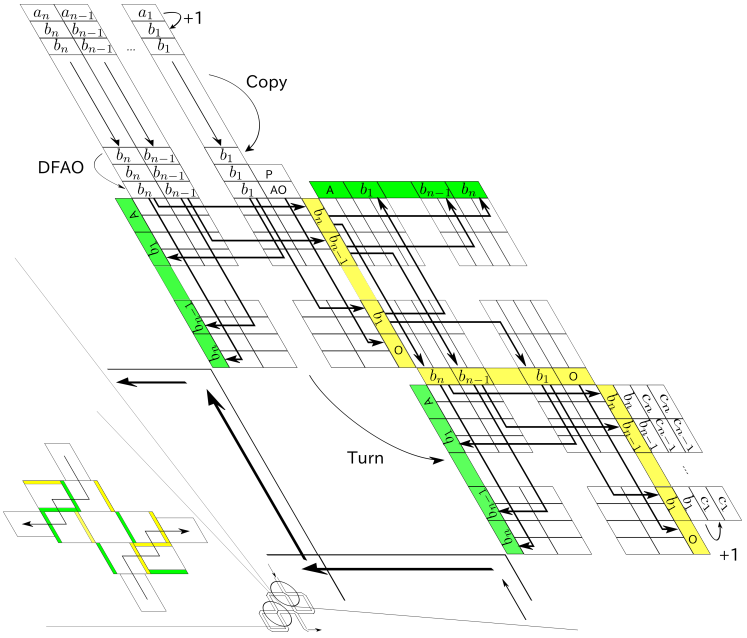

Fig. 7. Folding of one segment plus turn of the Heighway dragon, flow of information through it, and the two ways of collision avoidance between two turns.

intermediate outcome $P[i]$. The slanted Heighway dragon involves two types of left turn as well as two types of right turn: acute and obtuse. Observe that after (slanted) vertical line segments, the dragon turns left obtusely and right acutely, whereas after horizontal ones, it turns left acutely and right obtusely. Therefore, it suffices for D_v and D_h to compute $P[i] \in \{\mathrm{L}, \mathrm{R}\}$ in the same way and D_v interprets L as O and R as A, while D_h interprets L as A and R as O.

One issue intrinsic to the folding by OS rises when the dragon makes a turn where it has already turned before. The OS is, by definition, not allowed to put a bead anywhere occupied by another bead. Hence, the dragon must be scaled-up somehow. We employ the c-rhombus scaling for c so large that a rhombus corresponding to a point affords two turning modules, which otherwise collide, as long as they fold into an L-shape (see Figs. 3, 6, and 7). The turning module consists of three copies of a functional subunit that bifurcates i while folding into a $c/3 \times c/3$ rhombus and guides the further folding towards the direction specified by the A/O signal fed by the previous DFAO module; towards green by A or yellow by O as shown in Fig. 7 (also refer Figs. 11 and 14).

Having outlined the generic design, now we explain how the design implements an OS for a specific target portion $P[j_1..j_2]$, or more precisely, how the modules C, D_v, D_h, T and their submodules are implemented, interlocked with each other, and collaborate. Let $n = \min\{m \mid j_2 < 2^m\}$. Each of the modules consists of submodules which are small, say, with several dozens of beads.

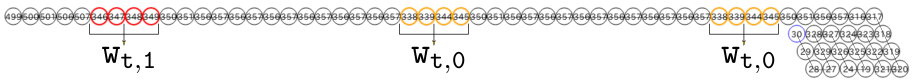

Fig. 8. The seed for the 3-bit Heighway dragon that starts at $j_1 = 100_2$.

Submodules implement various "functions" each of which is "called" in proper *environments*, i.e., the beads already placed around the tip of the transcript acting as the memory in the computation. The conformation that a submodule folds deterministically in a "valid" environment corresponds to the function to be called then. These "functional" conformations are called *bricks*, upon which the whole folding is built. *Brick automata* (BA) describe the OS' behavior at submodule level, which can be found in Sects. A.1, B.1, and C.1. They enumerate all the pairs of an environment to be encountered and the brick to be folded there as well as transitions among them. Once verified, all the BAs for C (resp. D_v, D_h, and T) guarantee that the module outputs in the expected format (3) (resp. (4), (5), and (2)), and hence we can say that the DOS behaves as described in the module automaton and folds into $P[j_1..j_2]$. Using a simulator developed for [9], we verified all the BAs. This amounts to the proof of Theorem 1.

Seed (Fig. 8) encodes the initial count $i = j_1$ in its binary representation $b_n b_{n-1} \cdots b_1$ as the following sequence of bead types:

$$\mathtt{499}{\to}\mathtt{500}{\to}\mathtt{501}{\to}\mathtt{506}{\to}\mathtt{507}{\to}\bigodot_{k=n}^{2}(\mathtt{w}_{t,b_k}{\to}\mathtt{350}{\to}\mathtt{351}{\to}(\mathtt{356}{\to}\mathtt{357}{\to})^{6})\mathtt{w}_{t,b_1} \tag{2}$$

where $\mathtt{w}_{t,0} = \mathtt{338}{\to}\mathtt{339}{\to}\mathtt{344}{\to}\mathtt{345}$ and $\mathtt{w}_{t,1} = \mathtt{346}{\to}\mathtt{347}{\to}\mathtt{348}{\to}\mathtt{349}$.

Counter module C is borrowed from [6] with technical modification to let it operate in the dynamics (1), which is more prevailing [8, 9, 11] though less tractable. We hence leave all its details to the Appendix (Sect. A), and just describe its input and output. It takes the current count i formatted as (2), which is fed by the seed or by the previous turning module, increments the count by 1 unless it is preceded by the seed, and outputs the resulting count in its binary representation $a_n a_{n-1} \cdots a_1$ in the following format:

$$\mathtt{44}{\to}\mathtt{45}{\to}\mathtt{46}{\to}\mathtt{51}{\to}\mathtt{52}{\to}\bigodot_{k=n}^{2}(\mathtt{w}_{c,a_k}{\to}(\mathtt{75}{\to}\mathtt{76}{\to})^{5}\mathtt{51}{\to}\mathtt{52}{\to})\mathtt{w}_{c,a_1} \tag{3}$$

where $\mathtt{w}_{c,0} = \mathtt{57}{\to}\mathtt{58}{\to}\mathtt{63}{\to}\mathtt{64}{\to}\mathtt{69}{\to}\mathtt{70}$ and $\mathtt{w}_{c,1} = \mathtt{65}{\to}\mathtt{66}{\to}\mathtt{67}{\to}\mathtt{68}{\to}\mathtt{69}{\to}\mathtt{70}$.

DFAO modules D_v, D_h receive the current count i in the format (3) from the previous counter module, compute $P[i]$, and interpret it as A or O properly. D_v and D_h then output the interpretation along with the count i in the following formats, respectively:

$$\bigodot_{k=n}^{2}(\mathtt{w}_{d,a_k}{\to}(\mathtt{52}{\to}\mathtt{51}{\to})^{7})\mathtt{w}_{d,a_1}{\to}\mathtt{52}{\to}\mathtt{51}{\to}\mathtt{200}{\to}\mathtt{199}{\to}\mathtt{w}_{dv,P[i]} \tag{4}$$

$$\bigodot_{k=n}^{2}(\mathtt{w}_{d,a_k}{\to}(\mathtt{52}{\to}\mathtt{51}{\to})^{7})\mathtt{w}_{d,a_1}{\to}\mathtt{52}{\to}\mathtt{51}{\to}\mathtt{311}{\to}\mathtt{310}{\to}\mathtt{w}_{dh,P[i]} \tag{5}$$

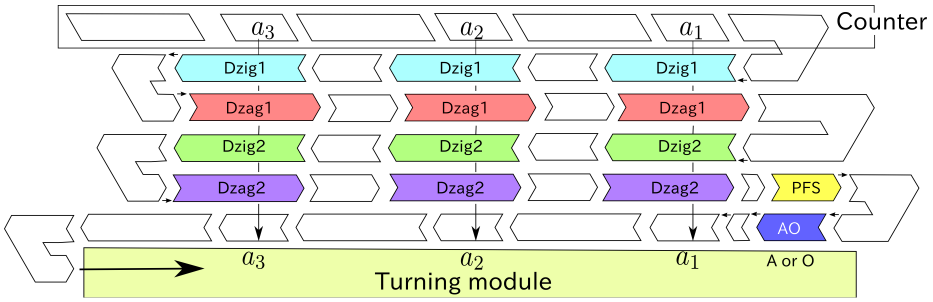

Fig. 9. Submodule-level abstraction of the folding of DFAO module.

where $\mathtt{w}_{\mathtt{dv,L}} = 198 \to 197$, $\mathtt{w}_{\mathtt{dv,R}} = 194 \to 193$, $\mathtt{w}_{\mathtt{dh,L}} = 305 \to 304$, and $\mathtt{w}_{\mathtt{dh,R}} = 309 \to 308$.

What the DFAO in Fig. 2 really does for computing $P[i]$ is to search for the first 0 from the LSB and check if it is followed by 0 ($P[i] = \mathrm{R}$) or by 1 ($P[i] = \mathrm{L}$). See Fig. 9. D_v employs the six submodules: `Dzig1`, `Dzag1`, `Dzig2`, `Dzag2`, `PFS`, and $\mathtt{AO}_v$ (resp. $\mathtt{AO}_h$), which are interleaved by spacers, as well as those that guide the transcript into two zigzags and one more zig (throughout the paper, zigs are to go leftward while zags are to go rightward). The first zigzag is for the search, the second is for the check and computation of $P[i]$, and the third zig is for the interpretation of $P[i]$ as A/O. While performing these tasks, these zigs and zags also propagate the count i to the next turning module.

In the first zig, n copies of `Dzig1` detect the first 0 collaboratively in two phases. See Fig. 10 for all the bricks of `Dzig1` with the corresponding environments. Phase 1 is to copy all the 1's prior to the first 0 and Phase 2 is to copy all the bits after the 0. `Dzig1` knows which phase it is in by the relative position where it starts folding to the input above (top in Phase1, bottom in Phase 2). In Phase 1, `Dzig1`s certainly fold into the brick Dzig1-1. At the first 0, a `Dzig1` rather folds into Dzig1-f0 brick, ending at the top in order to transition to Phase 2. Each of the remaining `Dzig1` folds into either Dzig1-20 or Dzig1-21, copying all the remaining bits. Interleaving spacers are implemented as a glider (see Sect. 2), hence capable of propagating 1bit (top/bottom) on which phase the system is in. In the first zag, n copies of `Dzag1` reformat and propagate 0's, 1's, and the first 0 using the three bricks in Fig. 27.

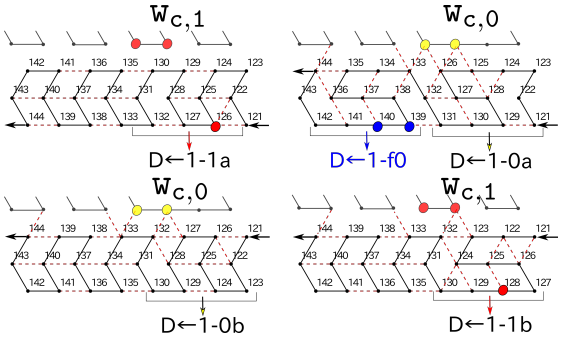

Fig. 10. The four bricks of `Dzig1`: (top) Dzig1-1 and Dzig1-f0; (bottom) Dzig1-20 and Dzig1-21.

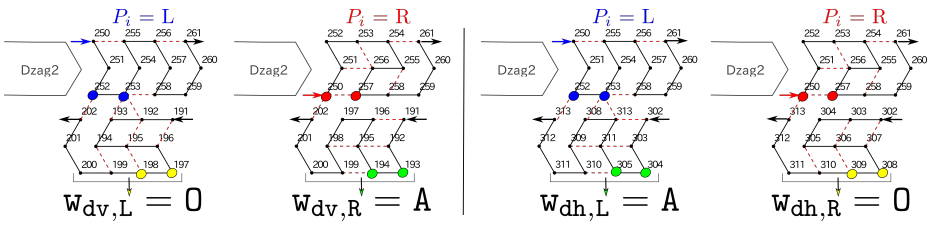

Fig. 11. The two bricks of PFS above and the corresponding two bricks of (left) $\mathtt{AO}_v$ and (right) those of $\mathtt{AO}_h$.

In the second zig, n copies of Dzig2 check whether the first 0 is followed (being read from LSB) by 0 or 1, in a similar manner to the search in the first zig. They usually take one of the two bricks Dzig2-0 and Dzig2-1 to copy 1's and 0's, which start and end at the bottom (see Fig. 28). At the encounter to the first 0, a Dzig2 folds into a special brick Dzig2-f0 and ends rather at the top. The next Dzig2, if any, starts folding at the top so that it takes the special brick Dzig2-0f0 if the first 0 is followed by 0 or Dzig2-1f0 if it is followed by 1. Recall the reading 1 here is a necessary and sufficient condition for the DFAO in Fig. 2 to transition to q_2, that is, $P[i] = \mathrm{L}$. Dzig2-1f0 exposes a marker q_2 downward. These bricks end at the bottom so that the remaining bits are copied by the ordinary bricks Dzig2-0 and -1. The second zag starts at the bottom and copy 0's and 1's by the two bricks of Dzag2 (top left and center in Fig. 29) until a Dzag2 encounters the 1 marked by q_2, if any. At the encounter, the Dzag2 folds into the special brick Dzag2-T1 and changes the ending position to the top, letting the remaining Dzag2 rather fold into the bricks Dzag2-R0 and -R1 for copying, which end at the top. As such, the second zag can feed $P[i]$ to PFS as the position of its first bead.

At the beginning of the third zig, D_v employs $\mathtt{AO}_v$ to convert $P[i]$ into $\mathtt{w_{dv,P[i]}}$ while D_h employs rather $\mathtt{AO}_h$ to convert $P[i]$ into $\mathtt{w_{dh,P[i]}}$. The turning module interprets $\mathtt{w_{dv,L}}$ and $\mathtt{w_{dh,R}}$ as turning obtusely while $\mathtt{w_{dv,R}}$ and $\mathtt{w_{dh,L}}$ as turning acutely. As a part of effort to save bead types, the submodule $\mathtt{AO}_v$ is also diverted in order for both D_v and D_h to propagate the count i in the rest of this zig.

Turning module T consists of 3 copies of the pair of two functional units called *bit-bifurcator* and *steering arm*. See Fig. 12. The bit-bifurcator forks the count $i = a_n a_{n-1} \cdots a_1$ left and rightward while folding into zigzags. It consists of 10 submodules, which handle the following tasks:

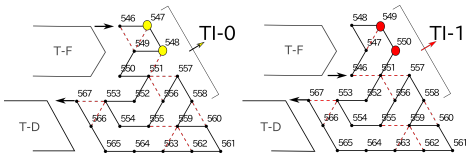

Fig. 13. The two bricks of turn-rgp.

1. Propagate 1-bit vertically: body-rpx1, body-rpx2, body-lpx1, body-lpx2;
2. Let 1-bit propagating vertically cross another 1-bit propagating horizontally: body-gx1, body-gx2;

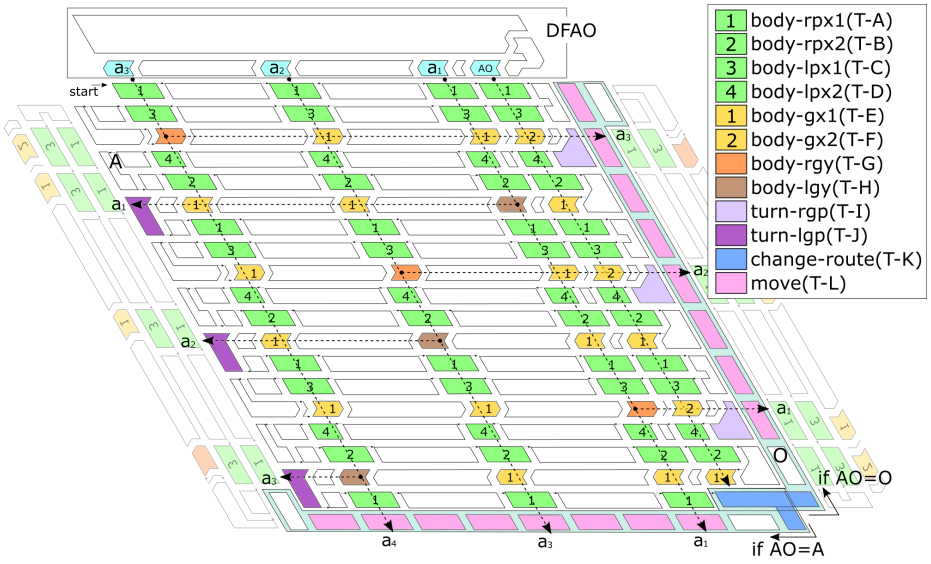

Fig. 12. Submodule-level abstraction of the whole folding of the pair of a bifurcator and steering arm. All the white submodules are spacers, some of which are implemented in the shape of parallelogram instead of glider.

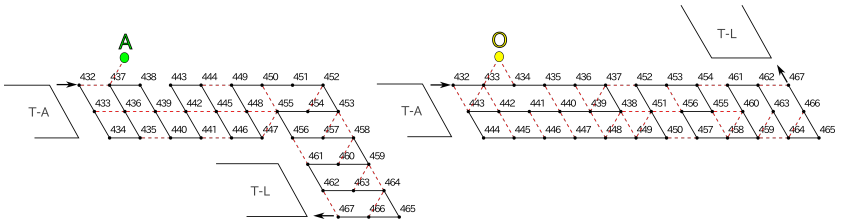

Fig. 14. The two bricks of `change-route`.

3. Fork 1-bit vertically and horizontally: `body-rgy`, `body-lgy`;
4. Undergo transition between a zig and a zag and exposes 1-bit outside: `turn-rgp`, `turn-lgp`.

Submodules to handle the first two types of tasks have already been implemented (see, e.g., [9]). The submodule `body-rgy` is implemented by recycling the first half of `Dzag2` (Fig. 29). Starting from the bottom, it can take two conformations that end at different heights and expose sequences bead types distinct enough downward. The 1-bit thus forked transfers till the end of a zag and is converted into a sequence of bead types by `turn-rgp` (Fig. 13). The `body-lgy` and `turn-lgp` are the zig-counterparts of them (Figs. 42 and 43).

The bifurcator also propagates the 1-bit A/O signal, output by the previous DFAO module, to let the steering arm know which way to go. Specifically, the signal has the `change-route` submodule of the steering arm take one of the

two bricks shown in Fig. 14, guiding the rest of the arm towards the specified direction. The rest of the arm is a catenation of `move` submodules (Fig. 44), which is capable of letting the bifurcated bit sequence through. Note that the turning module does not have to bifurcate the A/O signal. Indeed, the second and third pairs of bifurcator and steering arm are supposed to turn in the same manner as the first. It hence suffices to append A and O to the bifurcated bit sequences on the acute and obtuse sides, respectively.

Acknowledgements We would like to thank Hwee Kim for valuable discussions.

References

1. Abelson, H., diSessa, A.: Turtle Geometry The Computer as a Medium for Exploring Mathematics. MIT Press Series in Artificial Intelligence, The MIT Press (1981)
2. Alberts, B., Johnson, A., Lewis, J., Morgan, D., Raff, M., Roberts, K., Walter, P.: Molecular Biology of the Cell. Garland Science, 6th edn. (2014)
3. Allouche, J.P., Shallit, J.: Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press (2003)
4. Doty, D.: Theory of algorithmic self-assembly. Commun. ACM 55(12), 78–88 (2012)
5. Geary, C., Andersen, E.S.: Design principles for single-stranded RNA origami structures. In: Proc. DNA20. pp. 1–19. LNCS 8727, Springer (2014)
6. Geary, C., Meunier, P.E., Schabanel, N., Seki, S.: Programming biomolecules that fold greedily during transcription. In: Proc. MFCS2016. pp. 43:1–43:14. LIPIcs 58 (2016)
7. Geary, C., Rothemund, P.W.K., Andersen, E.S.: A single-stranded architecture for cotranscriptional folding of RNA nanostructures. Science 345(6198), 799–804 (2014)
8. Han, Y.S., Kim, H.: Ruleset optimization on isomorphic oritatami systems. In: Proc. DNA23. pp. 33–45. LNCS 10467, Springer (2017)
9. Han, Y.S., Kim, H., Ota, M., Seki, S.: Nondeterministic seedless oritatami systems and hardness of testing their equivalence. In: Proc. DNA22. pp. 19–34. LNCS 9818, Springer (2016)
10. Ma, J., Holdener, J.: When Thue-Morse meets Koch. Fractals 13, 191–206 (2005)
11. Ota, M., Seki, S.: Ruleset design problems for oritatami systems. Theor. Comput. Sci. 671, 26–35 (2017)
12. Patitz, M.J.: Self-assembly of fractals. In: Encyclopedia of Algorithms, pp. 1918–1922. Springer (2016)
13. Rothemund, P.W.K., Papadakis, N., Winfree, E.: Algorithmic self-assembly of DNA Sierpinski triangle. PLoS Biology 2(12), e424 (2004)
14. Watters, K.E., Strobel, E.J., Yu, A.M., Lis, J.T., Lucks, J.B.: Cotranscriptional folding of a riboswitch at nucleotide resolution. Nat. Struct. Mol. Biol. 23(12), 1124–1133 (2016)
15. Winfree, E.: Algorithmic Self-Assembly of DNA. Ph.D. thesis, California Institute of Technology (June 1998)

A Counter module

As described in the module automaton in Fig. 6, the counter module receives the count i either from the seed or from the previous turning module, increments it by 1 unless being preceded by the seed, and propagates it at some distance for the succeeding DFAO module.

The transcript of the n-bit counter module is periodic. Its period is the concatenation of the following submodules:

- n submodules `Half-adder` (abbreviated as `HA`); the half-adder;
- Spacers interleaved between `HA`s;
- Submodule `Left-turner` (`CLT`); the left-turner;
- n submodules `Formatter` (`CF`), the formatter;
- Spacers interleaved between `CF`s;
- Submodule `Right-turner` (`CRT`); the right-turner.

The length of one period is hence $O(n)$.

The n `Half-adder`s and the spacers between them fold into a zig ($\leftarrow$), the `Left-turner` reverses the folding direction ($\hookrightarrow$), the n `Formatter`s and the spacers between them fold into a zag ($\rightarrow$), and the `Right-turner` reverses the folding direction back ($\hookleftarrow$) for the next zigzag folded by the next period. The zig is to fold below a sequence of beads that encodes the current count i in the format (2) if this is the first zig or in the format (3) otherwise, and to increment i to $i + 1$ if being fed with carry. Its output (i or $i + 1$) is encoded in an internal format, different from (3) in which 1 is encoded either as `HA-1a` = `10-9-8-7` or `HA-1b` = `10-9-8-1` and 0 as `HA-0a` = `12-7-6-1` or `HA-0b` = `10-9-4-3`. The next zag folded by the n formatters and the spacers between them is to reformat the output of the zig in (3) so that the next zig can read it.

The whole DOS is designed so as to feed carry only by the turning module to the first zig of the counter module. That is, any zig but the first is never fed with carry. As a result, the first counter module just propagates the initial count $i = j_1$ encoded on the seed to the succeeding DFAO module, while any other counter module increments the current count i given by the previous turning module by 1 and propagates it to the succeeding DFAO module.

A.1 Brick automata for Counter module

The behavior of the counter module mentioned above is described at the submodule level in the four brick automata shown in Figs. 15, 16, 17, and 18. The first, second, and third BAs are for the zig. The first BA (Fig. 15) is for the first zig of the first counter module, which folds just below the seed. It is not fed with carry so that all the submodules are to start folding at the top (no carry). The second BA (Fig. 16) is for the first zig of other counter modules, which folds just below the preceding turning module. As described in the module automaton for this DOS (Fig. 6), it is fed with carry. Note that only this second BA among these three involves transitions labeled with B (bottom, i.e., with carry). The third BA (Fig. 17) is for all the other zigs, which are preceded by a zag. The zag is always preceded by a zig so that we need only one BA for the zag (Fig. 18).

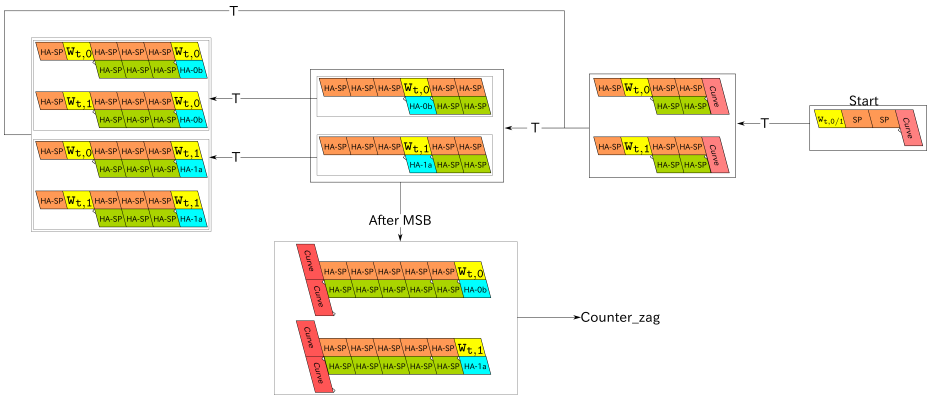

Fig. 15. The brick automaton for the first zig of the first counter module, which folds just below the seed. The seed does not feed carry.

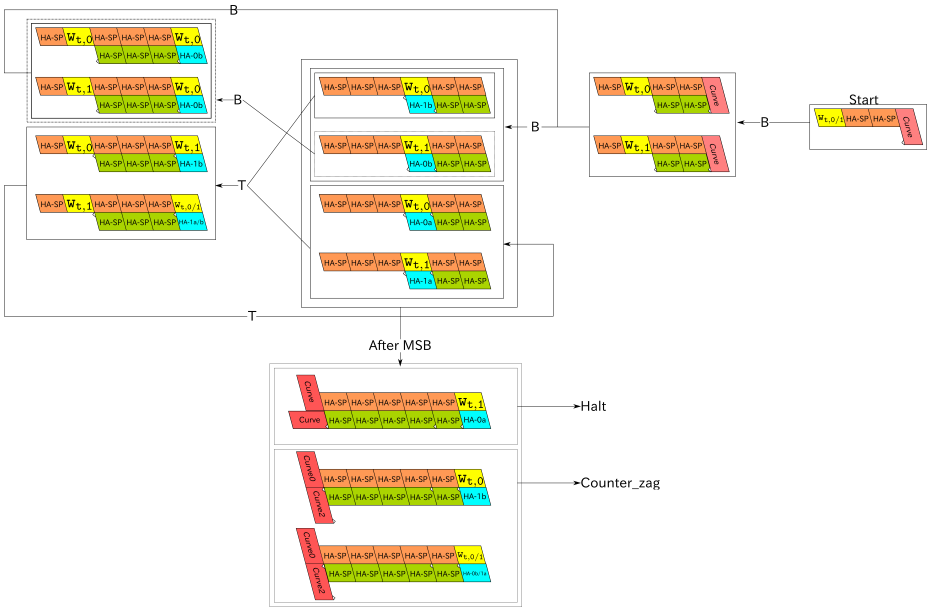

Fig. 16. The brick automaton for the first zig of all the counter modules but the first one, which are preceded by a turning module. The turning module feeds carry.

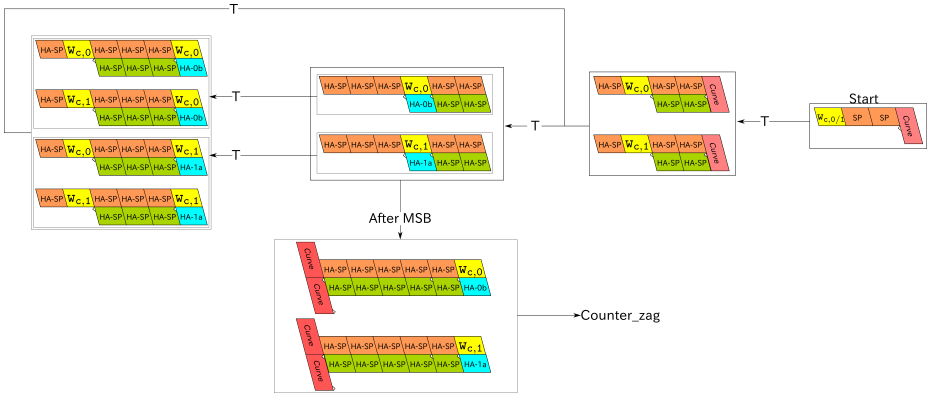

Fig. 17. The brick automaton for every zig except the first, which is never fed with carry by the previous right turner.

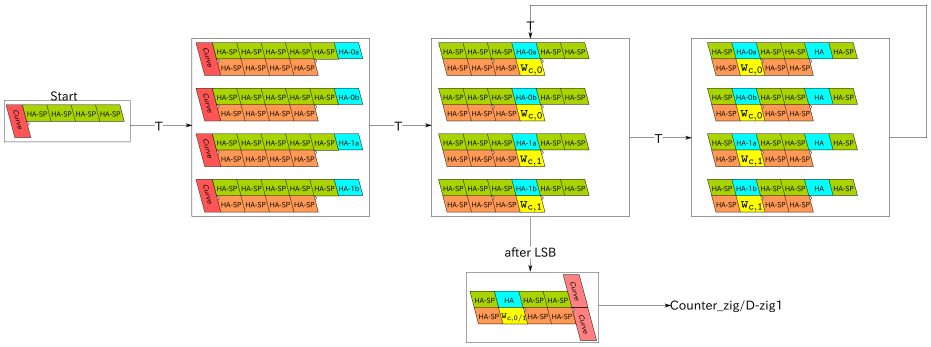

Fig. 18. The brick automaton for all the zags of the counter module.

A.2 Bricks for Counter module

According to the brick automata in Sect. A.1, we design the four submodules: `Half-adder`, `Left-turner`, `Formatter`, and `Right-turner` as follows.

Half-adder (Fig. 19) folds into the four different conformations (bricks) in eight expected environments which are characterized by the four sequences of bead types above: $\mathtt{w}_{\mathtt{t},1}$, $\mathtt{w}_{\mathtt{t},0}$, $\mathtt{w}_{\mathtt{c},1}$, and $\mathtt{w}_{\mathtt{c},0}$ and the position to start folding: top or bottom. The sequences $\mathtt{w}_{\mathtt{t},1}$ and $\mathtt{w}_{\mathtt{c},1}$ encode the 1-bit input 1 to this half-adder and $\mathtt{w}_{\mathtt{t},0}$ and $\mathtt{w}_{\mathtt{c},0}$ encode 0. Starting at the bottom and top is interpreted as being fed with carry and no-carry, respectively. Among these eight environments, those two with $\mathtt{w}_{\mathtt{c},0}/\mathtt{w}_{\mathtt{c},1}$ and carry are never encountered because only the first zig can be fed with carry and the count i they read is given by the seed or the previous turning module, and hence, in the format (2). The brick automata in Figs. 15, 16, and 17 guarantee that `HA` never encounter any other environment, either.

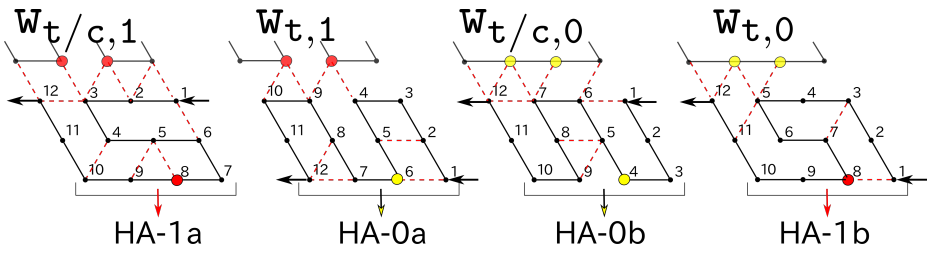

Fig. 19. The four bricks of `Half-adder` (`HA`). The first and third are diverted to implement the `body-lpx2` (T-D) submodule of the turning module.

Left- and Right-turner always fold into one brick so that we omit their figures. Moreover, their bricks end at the top. This means, in particular, that `Right-turner` never feeds carry to the succeeding zig, that is, no zig but the first one of each counter module is capable of incrementing the count i.

Formatter (Fig. 20) encounters the four environments depending on the input above: `HA-0a`, `HA-0b`, `HA-1a`, and `HA-1b`. Since `Left-turner` always ends at the top and spacers always start and end at the top, `Formatter` never encounters any other environment (see the brick automaton in Fig. 18). `Formatter` folds as in Fig. 20 (left) and exposes $\mathtt{w}_{\mathtt{c},0}$ below if `HA-0a` or `HA-0b` is given, or it folds as in Fig. 20 (right) and exposes $\mathtt{w}_{\mathtt{c},1}$ below if `HA-1a` or `HA-1b` is given. As a result, the count i is now encoded in the format (3), which can be read by the succeeding zig.

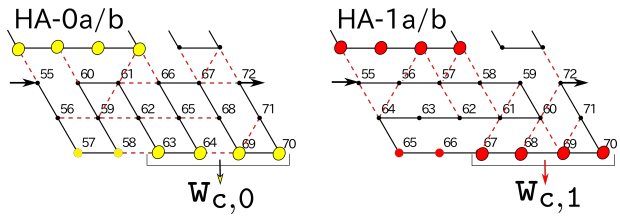

Fig. 20. The two bricks of `Formatter` (`CF`). This submodule `CF` is used as the submodule `body-rpx2` (T-B) of the turning module.

B DFAO module

B.1 Brick automata for DFAO module

The DFAO module is designed at the level of submodule based on the six brick automata shown in Figs. 21, 22, 23, 24, 25, and 26. The brick automaton in Fig. 25 is for D_v, while the one in Fig. 26 is for D_h. They differ only in the type of bricks that converts P_i into the A/O-signal; $\texttt{AO}_v$ (blue in Fig. 25) for D_v while $\texttt{AO}_h$ (red in Fig. 26). The labels B and T of transitions in these brick automata stand for that the previous submodule (glider) ends at the bottom and at the top, respectively.

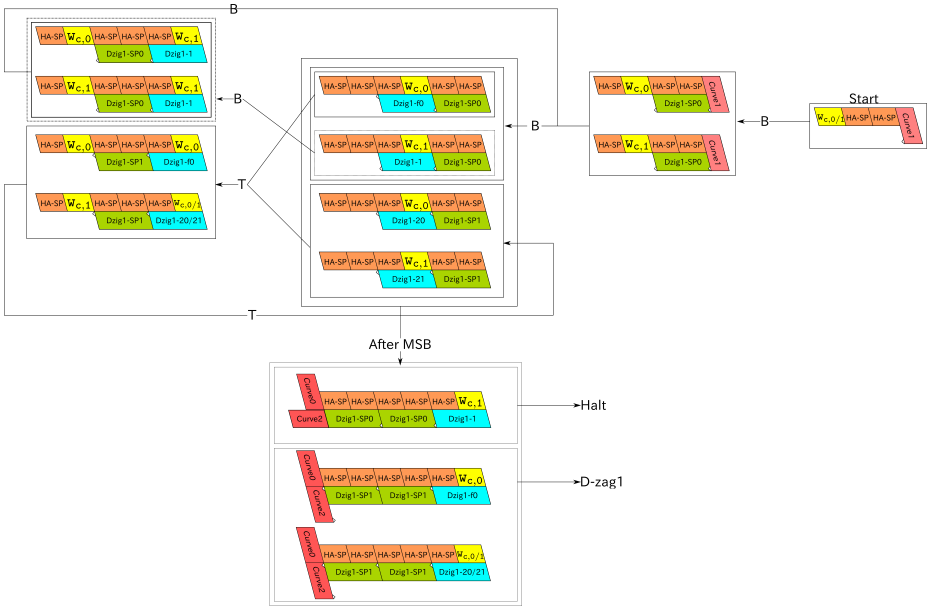

Fig. 21. The brick automaton for the first zig of DFAO module.

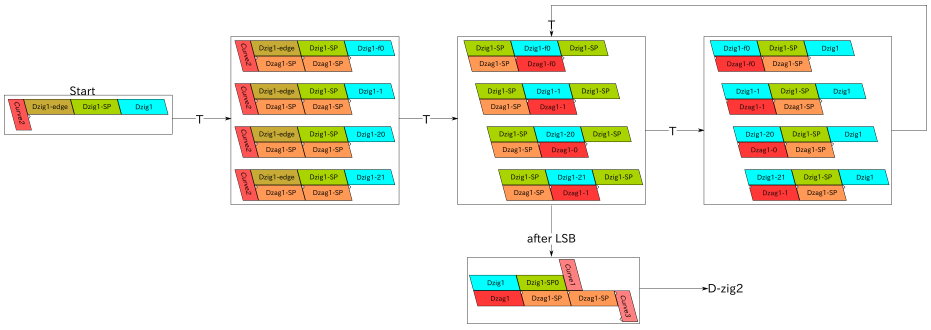

Fig. 22. The brick automaton for the first zag of DFAO module.

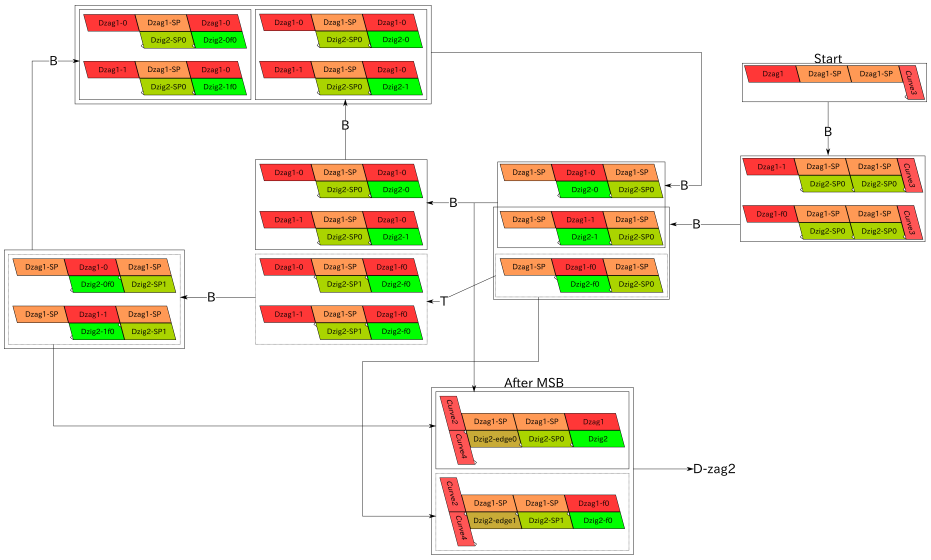

Fig. 23. The brick automaton for the second zig of DFAO module.

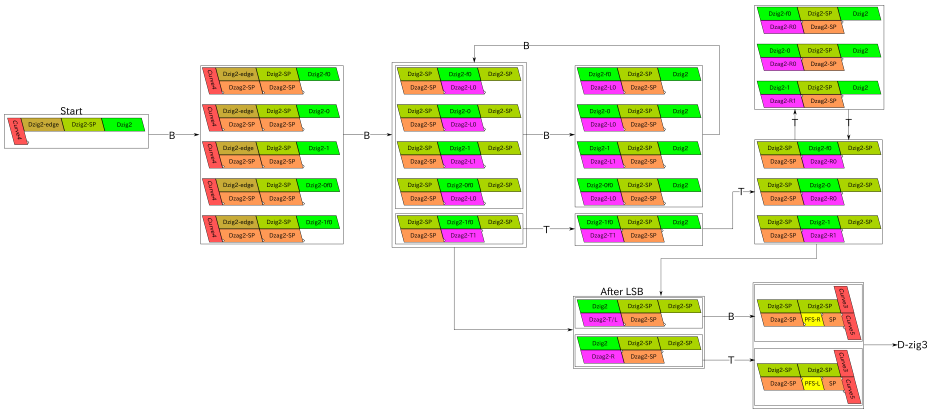

Fig. 24. The brick automaton for the second zag of DFAO module.

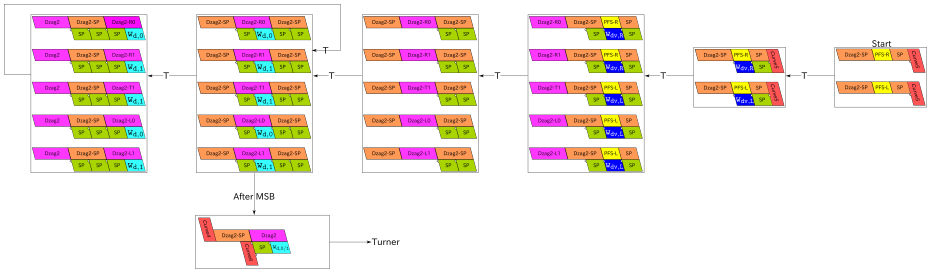

Fig. 25. The brick automaton for the third zig of DFAO module D_v, which is for a vertical segment.

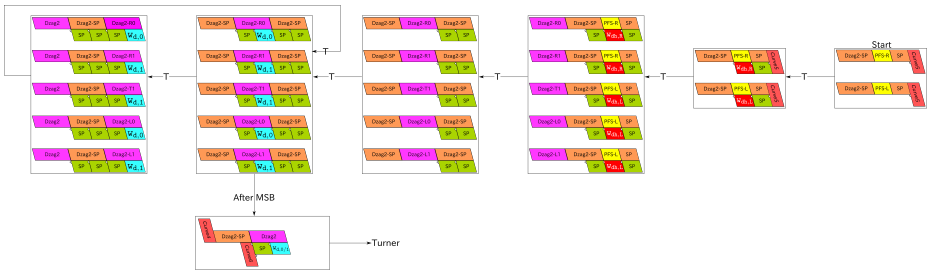

Fig. 26. The brick automaton for the third zig of DFAO module D_h, which is for a horizontal segment.

B.2 Bricks for DFAO module

All the bricks of the submodules of DFAO module omitted in the main text are shown in Figs. 27, 28, and 29.

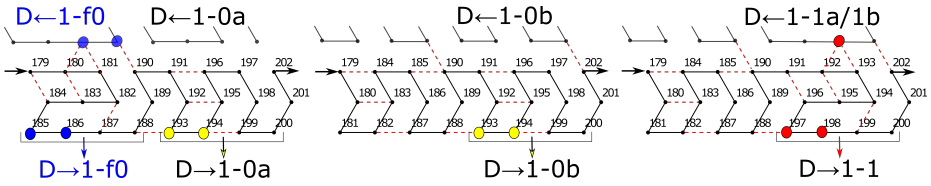

Fig. 27. The three bricks of `Dzag1`: Dzag1-f0, Dzag1-0, and Dzag1-1.

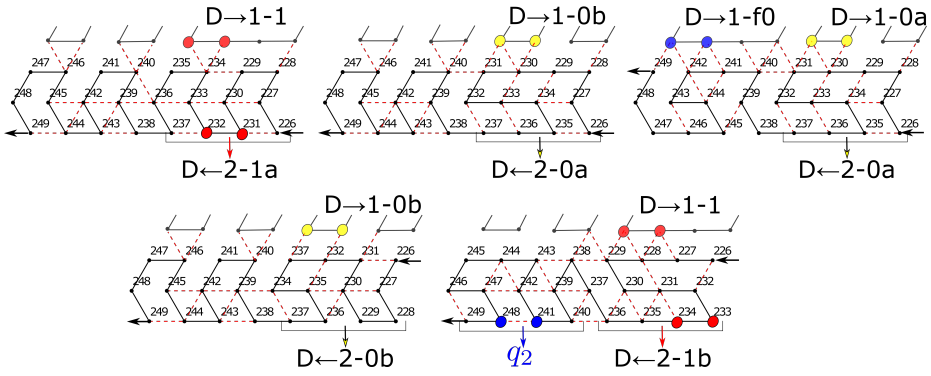

Fig. 28. The five bricks of `Dzig2`: (top) Dzig2-1, Dzig2-0, Dzig2-f0, (bottom) Dzig2-0f0 and Dzig2-1f0.

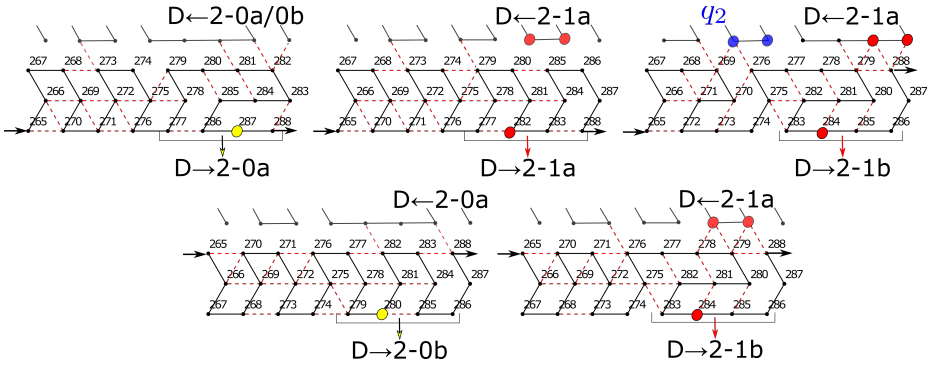

Fig. 29. The five bricks of `Dzag2`: (top) Dzag2-L0, Dzag2-L1, and Dzag2-T1; (bottom) Dzag2-R0 and Dzag2-R1. The first and second halves are diverted to implement the `body-gx1` (T-E) and `body-rgy` (T-G) submodules of the turning module, respectively.

C Turning module

C.1 Brick automata for Turning module

The turning module is designed at the level of submodule based on the ten brick automata shown in Figs. 30, 31, 32, 33, 34, 35, 36, 37, 38, and 39.

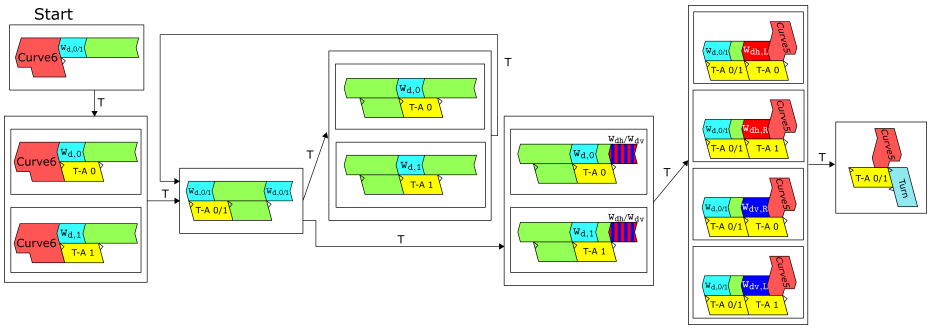

Fig. 30. The brick automaton for the first zag of Turning module.

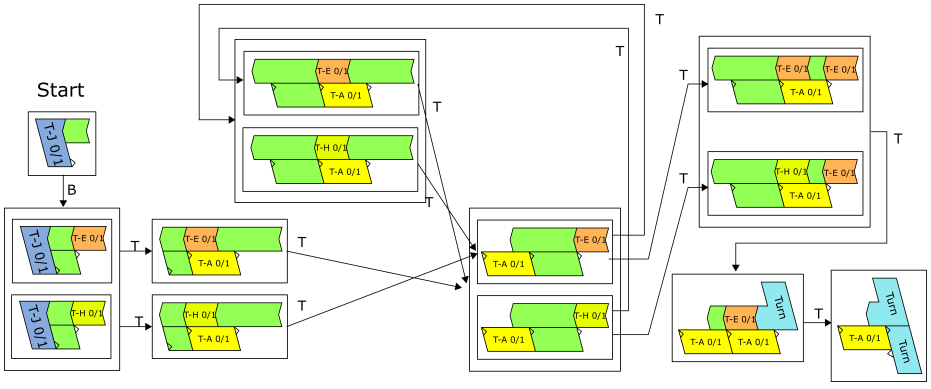

Fig. 31. The brick automaton for all zags that involve `body-rpx1` (T-A) below a zig that involves `body-gx1` (T-E) of Turning module.

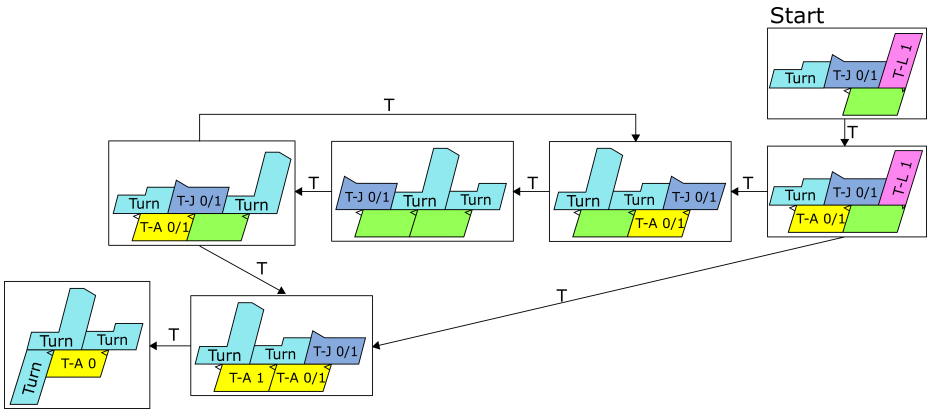

Fig. 32. The brick automaton for the first zag of the second and third bifurcators in case the first bifurcator received the A-signal.

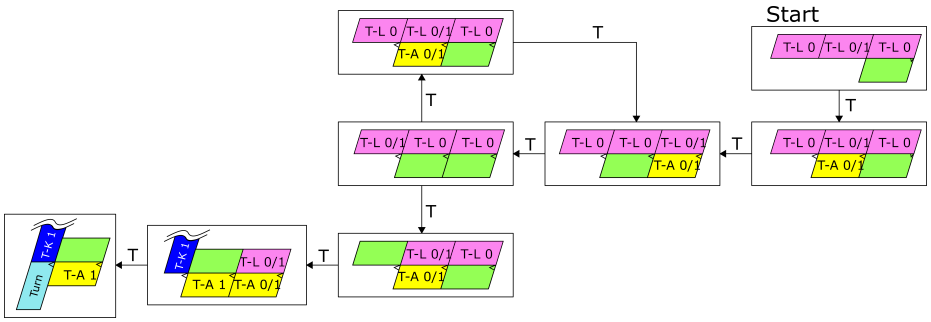

Fig. 33. The brick automaton for the first zag of the second and third bifurcators in case the first bifurcator received the O-signal.

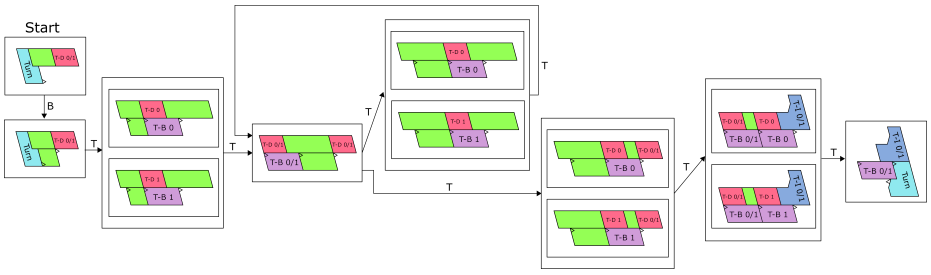

Fig. 34. The brick automaton for all zags that involve `body-rpx2` (T-B) of Turning module.

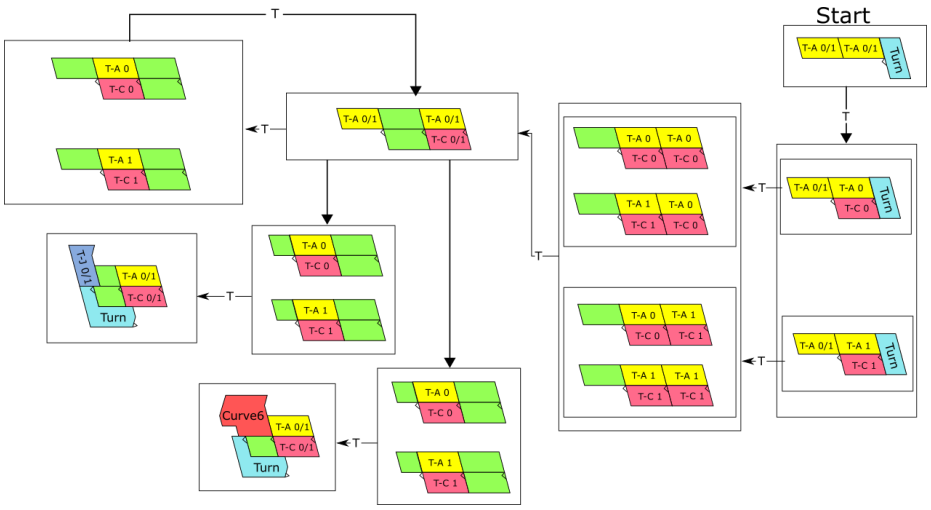

Fig. 35. The brick automaton for all zigs that involve `body-lpx1` (T-C) of Turning module.

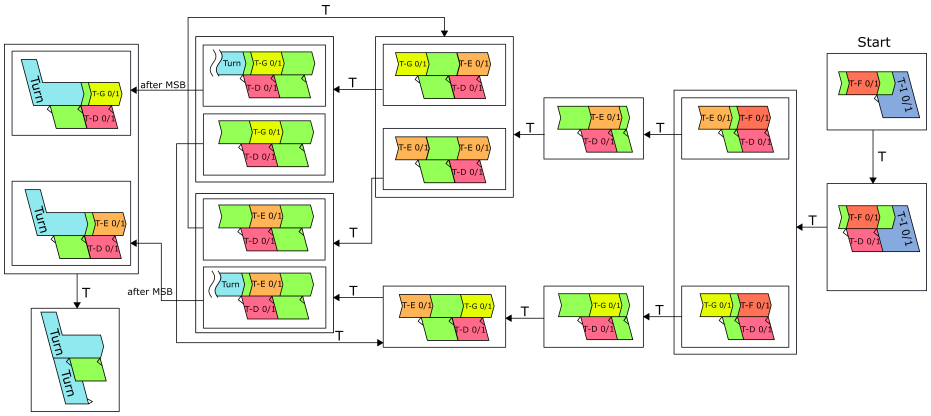

Fig. 36. The brick automaton for all zigs that involve `body-lpx2` (T-D) of Turning module.

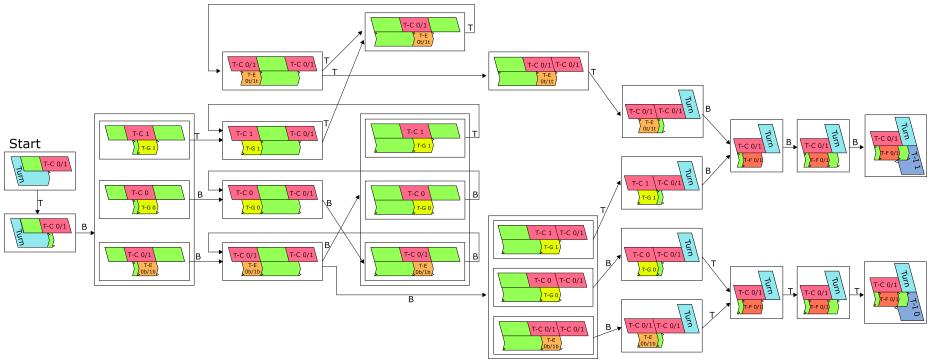

Fig. 37. The brick automaton for all zags that involve `body-gx1` (T-E) of Turning module, which also involve one `body-rgy` (T-G) and $n-1$ `body-gx2` (T-F).

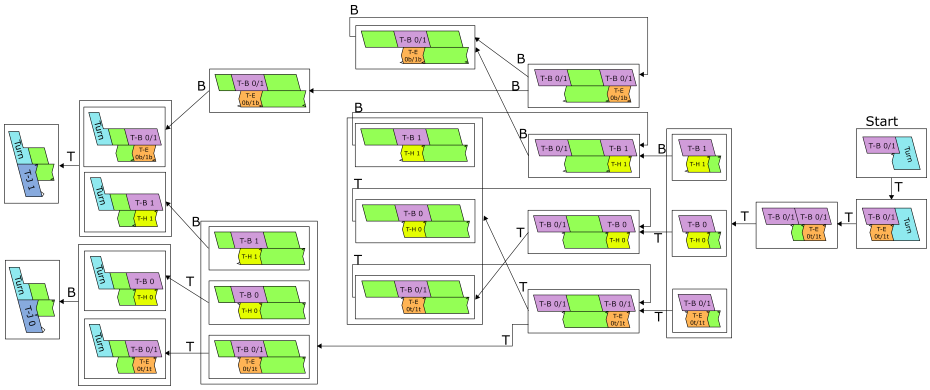

Fig. 38. The brick automaton for all zigs that involve `body-gx1` (T-E) of Turning module, which also involve one `body-lgy` (T-H).

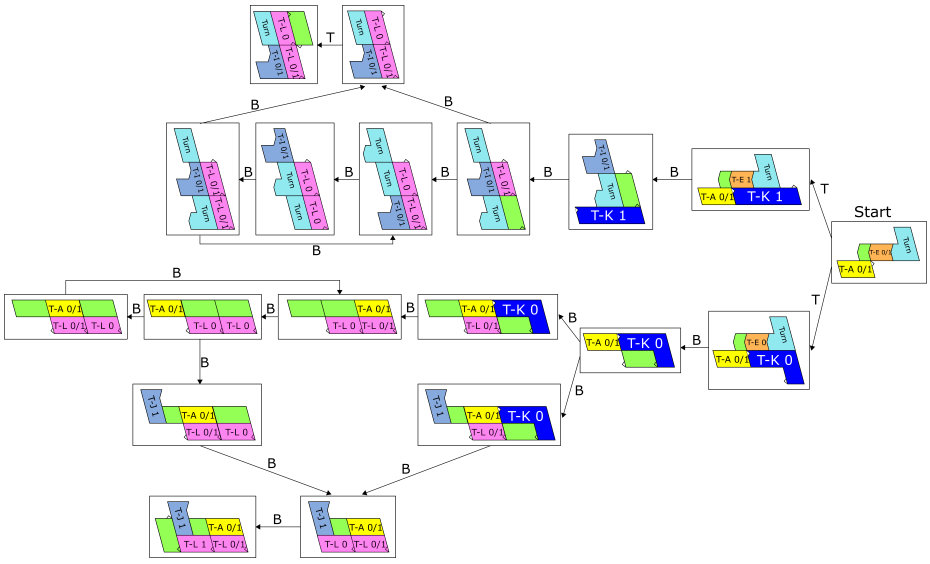

Fig. 39. The brick automaton for `change-route` (T-K) and `move` (T-L).

C.2 Bricks for Turning module

Inside the main text, we have already shown the bricks of the `turn-rgp` (T-I) and `change-route` (T-K) submodules of the turning module (Figs. 13 and 14). Below, we enumerate the bricks of the `body-rpx1` (T-A), `body-lpx1` (T-C), `body-lgy` (T-H), `turn-lgp` (T-J), and `move` (T-L) submodules in Fig.s 40, 41, 42, 43, and 44, respectively. The formatter `CF` and half-adder `HA` of the counter module are diverted respectively as the `body-rpx2` (T-B) and `body-lpx2` (T-D) submodules. The `Dzag2` submodule of the DFAO module is split into half, and the first and second halves are diverted respectively as the `body-rgy` (T-G) and `body-gx1` (T-E) submodules. Lastly, the `body-gx2` (T-F) submodule is obtained by implementing the `body-gx2` (T-E) using a pairwise-distinct transcript `414-415-...-425`; they behave exactly in the same way; this copy is necessary because if `body-gx1` were used in place for `body-gx2`, the previous `body-gx1` would prevent it from folding correctly into bricks.

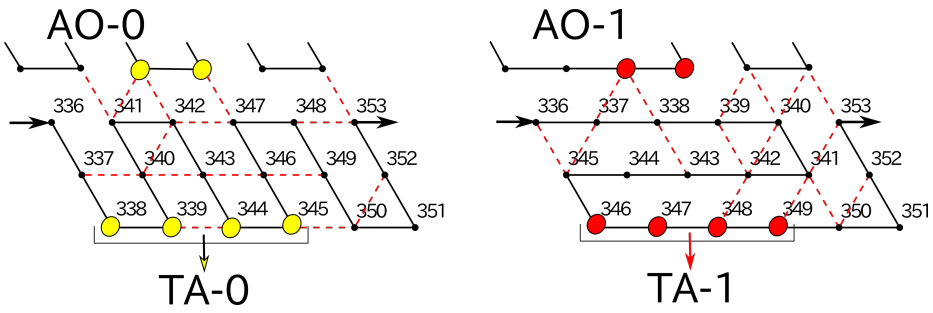

Fig. 40. The two bricks of `body-rpx1` (T-A) submodule

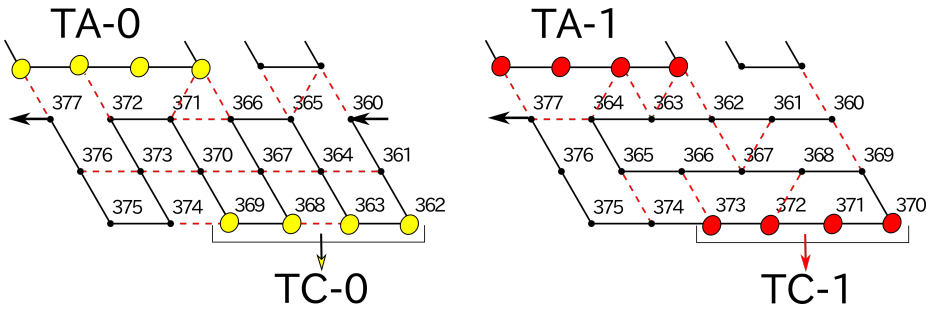

Fig. 41. The two bricks of `body-lpx1` (T-C) submodule

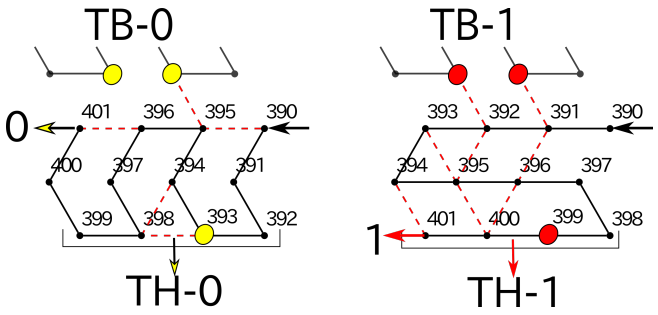

Fig. 42. The two bricks of `body-lgy` (T-H) submodule

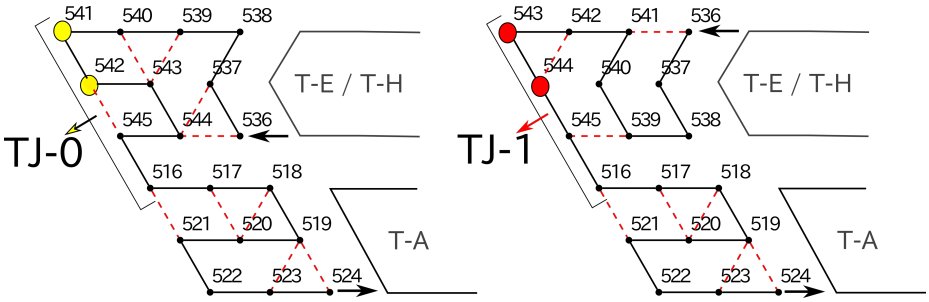

Fig. 43. The two bricks of `turn-lgp` (T-J) submodule

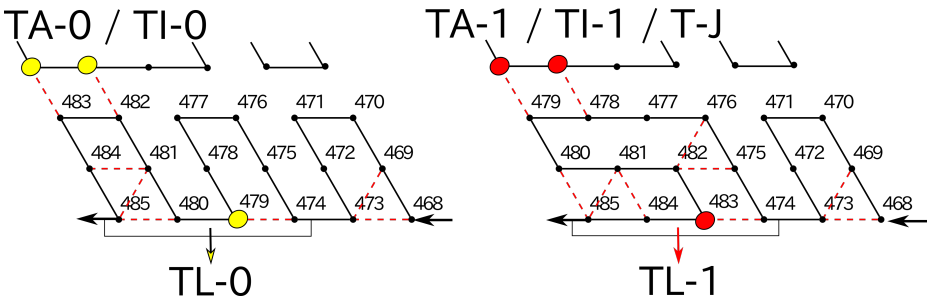

Fig. 44. The two bricks of `move` (T-L) submodule