The Nodal Language and Compiler

Casey Grun, Justin Werfel, and Peng Yin
Wyss Institute for Biologically Inspired Engineering
Harvard University

April 7, 2014

1 Introduction

Yin et al. [1] described a method of representing dynamic systems of interacting nucleic acid species using a simple, graphical "complementarity graph" abstraction. In this abstraction, "nodes" are behavioral units that represent underlying molecular implementations. "Ports" on a node represent domains, which can be programmed to interact with ports on other nodes to trigger changes in secondary structure. This abstraction is interesting to us for several reasons:

Abstraction — nodes can represent single species or arbitrarily complex systems, as long as their input-output behavior can be well-specified. This opens the fascinating possibility of writing "functions" whose basic design can be re-used within a single mechanism or across different projects.

Implementation agnostic — nodal programs encode the abstract semantics of a molecular program — the complementarities required for interaction, the input-output behavior of each species, etc. Beyond this, details of the implementation, such as the length of specific strands, can be easily reprogrammed and/or optimized separately from the logical description of the system. Although the Nodal programs describe herein all use the builtin node types described in Fig. 5, Nodal programs could just as easily describe various other underlying implementations, such as triggered origami opening and aggregation or entropy-driven branch migrations [2].

Geometry — nodal programs allow a concise representation of the structural relationship between individual elements (unlike, for instance, chemical reaction networks)

Here we formalize the conventions of the graphical representation of systems using this Nodal abstraction; we describe a Javascript Object Notation (JSON) [3]-based textual representation of these systems, called **DyNAML** (**Dy**namic **N**ucleic **A**cid **M**arkup **L**anguage); we propose a standard 'library' of useful hairpin node types based on these conventions; and we present a compiler for translating systems designed in the Nodal language into segment-level designs. This compiler was originally developed by Justin Werfel at Harvard, and this description has benefited tremendously from conversations with him.

1.1 Previous work

Previous DNA compilers have fallen into two categories. First, some compilers—such as Phillips and Cardelli's Visual DSD [4] and Ligocki's Pepper Compiler [5]—have focused on describing and representing systems at a segment level, providing only minimal abstractions on top of this representation. DSD provides integrated reaction enumeration and simulation facilities like DyNAMiC Workbench, but places major restrictions on the types of structural motifs and reactions which can be represented—in particular, no structures containing loops are allowed; this precludes design of the types of hairpin and junction structures discussed in Sec. **??**. The second set of compilers, such as Qian's Seesaw gate compiler [6] or Soloveichik's Chemical Reaction Network compiler [7] provide a high-level, but inflexible abstraction—like DSD, they allow only a limited subset of possible structural motifs to be represented within their abstraction. Our approach by contrast provides a rich and flexible abstraction and places no restrictions on the types of structural motifs that can be represented.

2 Concepts and terminology

We begin by precisely defining several concepts introduced in Chapter **??**. These terms describe the structural and behavioral features of Nodes. These terms are summarized in Fig. 1

Segment — a segment $x = (\sigma, r)$ [1] is a contiguous sequence σ of nucleotides that acts as a unit in hybridization, along with a role r. Segments are conventionally named with a number or lowercase letter(s). Each segment x has a "complementary" segment x^* whose sequence is the Watson-Crick complement of x. For mathematical convenience, we will sometimes represent a segment as an integer, such that segment a is complementary to segment $-a$. $|x|$ denotes the length (in nucleotides) of a segment. Segments perform different 'roles', which dictate features such as the length of the segment. Roles include: "toehold"—segments that serve as nucleation sites for intramolecular binding (and branch migration) events, "clamp"—short segments that block exposure of sequestered toeholds (due to breathing at the ends of a DNA duplex), and "recognition"—long, information-carrying segments.

Domain — a domain $d = (\langle s_1, s_2, \ldots \rangle, r, \phi)$ is a sequence of 1 or more segments, along with an assigned role r and a relative polarity ϕ. The role of a domain corresponds to the role of its port on a node (see 'Port' below); the polarity is discussed below. $|d|$ denotes the number of segments in a domain. Domains may be either exposed or sequestered:

Exposed — a domain is exposed if it has a toehold which is accessible to initiate toehold-mediated branch migration reactions with other domains.

Sequestered — a domain is sequestered if it contains no toeholds which are accessible to participate in toehold-mediated branch migration reactions with other domains.

Strand — a strand is a sequence domains. Strands have an intrinsic "directionality"—one end of the strand is the 5' end, and one is the 3' end (referring to the position of the carbon on the nucleotide sugar backbone). When drawn schematically, the 3' end of a strand is often indicated with an arrowhead.

Pole — the positive pole of a strand is the first indexed segment of a strand; the negative pole is the last. The pole of a domain is the toehold segment, while the negative pole is the segment furthest from the toehold.

Polarity — describes the relative directionality of components of the system. We denote the polarity of some object x by $\phi(x)$, where $\phi(x) \in \{-1, 1\}$. These conventions are summarized and motivated in Fig. 2.

- Strand polarity indicates whether segment indexing begins at the 3' or 5' end of the strand (that is, whether the positive pole is at the 3' or the 5' end of the strand). $\phi = 1$ means the *positive* pole is the 5' end, while $\phi = -1$ means the *negative* pole is at the 5' end.
- Domain polarity describes the location of that domain's toehold relative to the pole of the strand. A 'positive' domain ($\phi = 1$) has a toehold on the end of the domain that is closer to the strand's positive pole; a 'negative' ($\phi = -1$) domain's toehold is on the end closer to the negative pole.
- Segment polarity is used to indicate Watson-Crick complementation—e.g. segment `1` is complementary to `1*`.
- Node polarity refers to the relative directionality of its strands; any node can in principle be "flipped" by reversing the polarity of all of its strands, then reversing the order of the strands.

This complicated set of definitions allows a facile description of which binding interactions are permitted between domains. We define the "absolute polarity" $\Phi(x)$ of some object x to be the product of the polarity of each of its parents—for instance, the absolute polarity of a domain d on strand s within node n is $\Phi(d) = \phi(d) * \phi(s) * \phi(n)$. Two domains d_1 and d_2 can be bound only if their absolute polarities have opposite signs—that is $\Phi(d_1) = -\Phi(d_2)$. This corresponds to the notion that the two domains will be "anti-parallel" in the final structure, and therefore the *sequences* corresponding to these domains will be Watson-Crick complementary.

[1] In this chapter, we generally will use italic typeface to denote a variable, but typewriter font to denote a literal name. Therefore x is some arbitrary segment, while `s` refers to a particular segment with the name `s`.

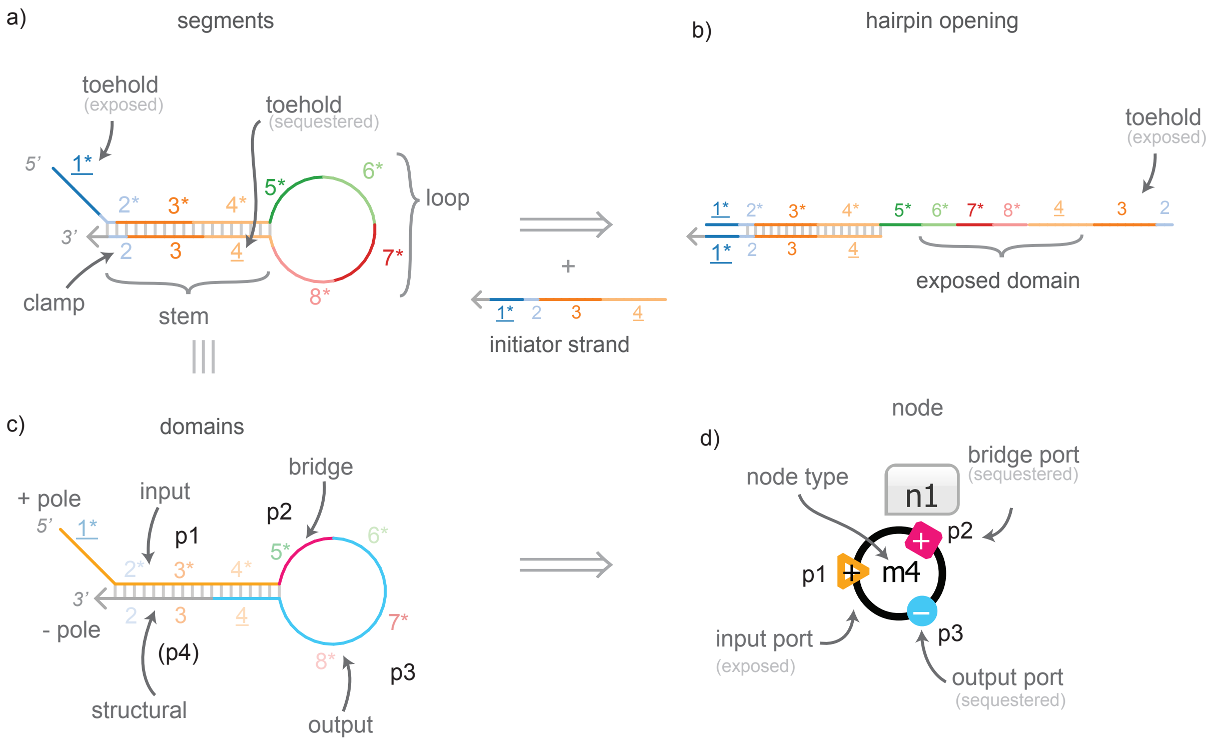


Figure 1: Features of a node and its structural implementation. **(a)** A complex of DNA molecule(s) whose behavior we wish to summarize with a node. The strand(s) of the complex are divided into "segments"—contiguous regions of nucleotides. Certain "toehold" segments (underlined) act as nucleation sites that can bind to strands on other complexes—these may be "exposed" for binding, or "sequestered" within a "stem" or "loop" of a structure, such as this hairpin. Other, short "clamp" segments act to prevent exposure of segments (due to spontaneous breathing of the DNA duplex forming the stem). For notational convenience, we designate the first-indexed segment to be the "positive pole" of the molecule, while the "negative pole" is the last-indexed segment. **(b)** Binding of a long strand to the exposed toehold (1*) could trigger branch-migration, displacing 2 and 3 and exposing the sequestered toehold 4; **(c)** The segments of this complex may be grouped into "domains" indicating the behaviorally-relevant parts of the molecule. We can describe the four segments 1*–4* as an "input" domain (p1), while 6*–4* are an "output" domain (p3). The segment 5* could act as binding region on its own, binding to another exposed domain without triggering branch migration—we call this a "bridge" domain. Finally, segments 2–3 are not intended to interact with other molecules, simply to protect segments in p1—we therefore call (p4) a "structural" domain, and we parenthesize its name to emphasize that it will not appear as a port in the node depicted to the right. **(d)** A node is a behavioral unit with several "ports", summarizing behaviorally relevant parts of the node's underlying molecular implementation—each port on the node represents a non-structural domain on the underlying implementation. The "polarity" of each "port" reflects whether its toehold is nearer to the positive or negative pole (positive domains are nearer the *positive* pole, while negative domains are nearer the negative pole). This notion of domain polarity allows nodes to be re-oriented in order to ensure strands which must bind in the final molecular implementation of the system are anti-parallel in their orientation.

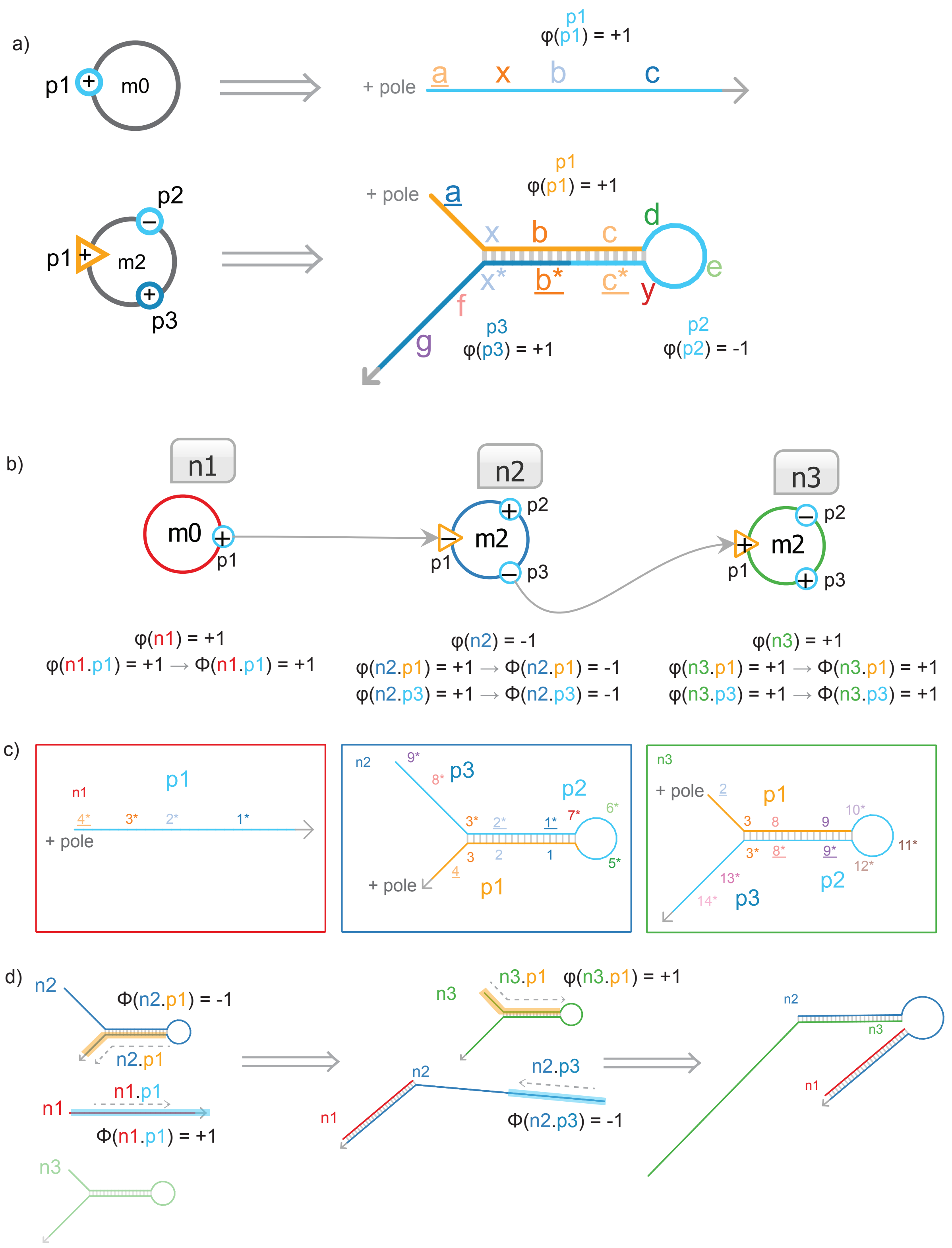


Figure 2: Demonstration of polarity. **(a)** Two node types: `m0` (single-stranded initiator) and `m2` (hairpin with two output domains). In `m2`, because the toeholds for both output domains need to be sequestered within the stem of the hairpin, `p2` must have its toehold farther from the positive pole (thus $\phi(\texttt{p1}) = -1$). **(b)** Simple Nodal system constructed using these node types. The connection between `n1.p1` and `n2.p1` requires `n2` to be "flipped" ($\phi(\texttt{n2}) = -1$) such that $\Phi(n1.p1) = -\Phi(n2.p1)$. **(c)** Molecular implementation of this system. Notice how `n2` and `n3` are essentially mirror images (because $\phi(\texttt{n2}) = -\phi(\texttt{n3})$). **(d)** Reaction mechanism executing this system, demonstrating how anti-parallelism is necessary for binding interactions between complementary domains to be possible.

Structural properties	**Behavioral properties**
Strand composition S (Segments, Domains/roles, Strands)	Available input, output, and bridge ports P
Complementarities $\mathbf{\Gamma}$ Structure T	Internal logic Λ

Table 1: Various underlying ('structural') properties of a particular molecule type yield interesting ('behavioral') properties in the node type

Node type — a 'class' of nodes; a node type maps a structural implementation (a "molecule type") to a behavioral description. The structure of the node type must define: the sequence of strands S comprising the node type, including a matrix of complementarity requirements between segments $\mathbf{\Gamma}$, and an initial structure T. These 'structural' properties map to higher-level, 'behavioral' properties, which describe features of the node type that are of interest to a nodal programmer; specifically, the composition of strands implies an available set of ports P, while the complementarity matrix and the initial structure imply an "input-output" or "internal logic" function Λ. A list of 'structural' and corresponding 'behavioral' properties for hairpin-based node types is shown in Figure 1. Each node type may also have a series of implementation 'parameters', which are left to be specified by the node.

Nodal species (or simply **Node**) — an instance of a node type which also encodes values for various implementation parameters. In particular, nodes specify the complementarity relationships between each port of the node type and ports on other nodes in the system.

Port — a port represents a domain which can interact with ports on other nodes via Watson-Crick base pairing to either trigger 'input-output behavior' of other nodes (e.g. the opening of hairpins), or to form structures with other nodes without otherwise altering the secondary structures of those nodes' underlying species. Ports may be exposed (open) or sequestered (closed), denoted by an open or a filled port symbol, respectively. Ports may take on several 'roles':

Input Port — when a strand binds to a node's input port, the secondary structure of that node is changed. This is known as the 'input-output behavior', or 'internal logic' of the node. In hairpins, this almost always results in the opening of one or more output or bridge ports. Input ports may also have combinatorial function — for instance, a node may have several input ports, all of which must be bound in order to trigger the input-output behavior.

Output Port — output ports may bind to input ports on other nodes to trigger input-output behavior on these nodes. This generally means they need to be long enough to trigger toehold-mediated branch migrations on the other nodes.

Bridge Port — bridge ports may not trigger input-output behavior on other nodes; instead, they generally serve a structural function. When open, they can bind to other exposed, complementary bridge ports. However, they do not otherwise trigger a change in secondary structure of the target node. Bridge ports generally consist of shorter domains without toeholds.

Structural Domain — some domains cannot actually interact with other nodes; these domains are designated "structural domains" and are not depicted as ports on the node.

System — a DyNAMiC system consists of a set of nodes.

3 Basic Semantics and Graphical representations

Yin et al. described a graphical language for programming dynamic, self-assembling nucleic acid systems. Here we extend and formalize that notation system to reduce ambiguity and allow for more diverse types of node types and interactions. Specifically, we distinguish between the 'complementarity graph' and the 'reaction mechanism,' and introduce specific conventions for depicting species not based on hairpins.

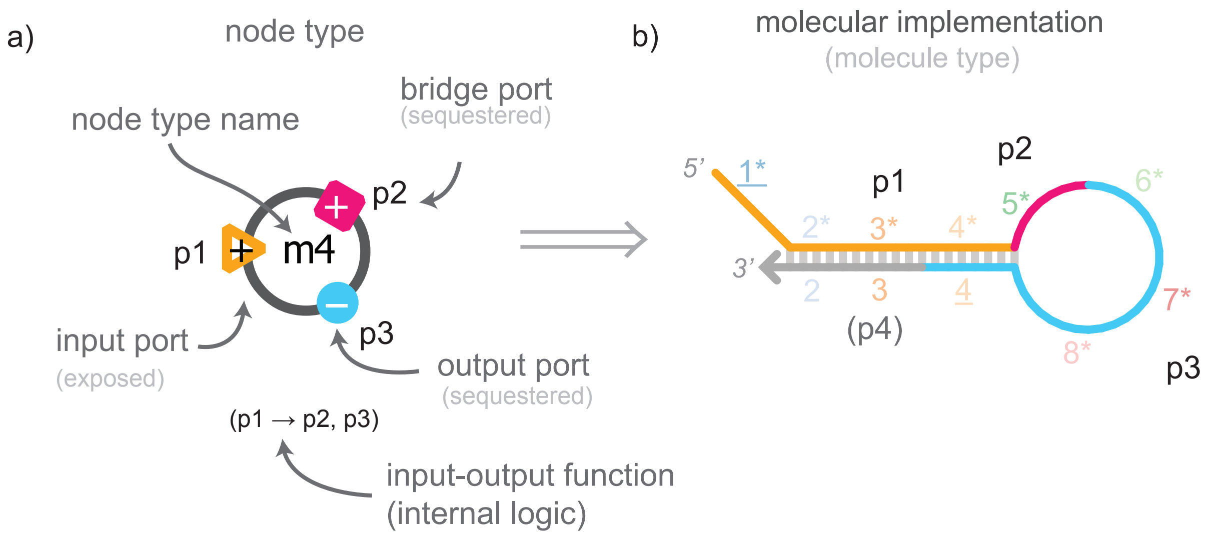


Figure 3: Explicit mapping of a graphical representation ('depiction') of a node type to a hairpin implementation. The graphic represents the arrangement of domains, the segment composition of those domains, and the mapping of domains onto ports. Underlined segments represent toeholds segments. Generally node types are drawn as grey circles, while nodes are drawn as black or colored circles.

3.1 Node types

To build a well-specified system of nodes, each node must have a defined molecular implementation. One way to do this is to explicitly specify the structural implementation of each node. More useful is to first describe a set of "node types" to serve as *templates* or *classes* for the nodes, then to create nodes by instantiating these node types. Defining a node type constitutes mapping a structural implementation to a behavioral unit. Specifically, we must specify the set of strands S (each of which is a collection of domains, including information about the role of each domain) comprising the node type, the initial structure T, and the matrix of intra-node type complementarities $\mathbf{\Gamma}$. The combination of these structural features implies the behavioral properties of the node type—its available ports, the 'input-output' or 'internal logic' function of the node type, and the initial configuration of each of the ports (opened or closed).

We represent this implicature by drawing the graphical representation (or 'depiction') of the node type, connected by a fat arrow ($\Rightarrow$) to a schematic of the underlying strand representation. The right-hand side depicts S, T, and $\mathbf{\Gamma}$, while the left-hand side shows the available ports P, internal logic Λ, and the initial configuration $\vec{p}$.

Each node type receives a textual label which, by convention, is a number preceded by the letter 'm,' (e.g. `m1, m2, m3, ...`). Each node type has some number of ports of the various roles described above. Each port in the node type must correspond to a domain on the strand. Conventionally, ports are labeled alphabetically along the node type, and corresponding domains are labeled alphabetically along the strand. Port/domain lettering is local to the node type— that is, the first port of every node type should be 'A.' Ports/domains of a node type can be unambiguously referred to as *¡node type name¿.¡port/domain name¿* (e.g. `m1.A`, `m1.B`, `m2.A`, `m2.B`, ...).

Each port is assigned a role—one of: input, output, or bridge. The role of a port is represented by a shape: inputs are equilateral triangles, bridges are squares, and outputs are circles. Open shapes represent open ports (exposed domains), while filled shapes represent closed ports (sequestered domains). When a node type is defined, a its ports should reflect the minimum free energy secondary structure of its corresponding molecular implementation in isolation. For hairpins, this generally means the 'closed' conformation of a hairpin. Ports can also be optionally colored to indicate their location in the secondary structure. Input ports which appear at the pole are orange. Input ports which appear within loops are red. Bridge ports are pink. Output ports within loops are blue. Output ports at a pole are green. Output ports on initiator (single) strands are brown. For clarity, we will not always obey this convention, simply coloring input ports orange, output ports blue, and and bridge ports pink.

When depicted, the name of each port in the node type should appear within the port shape. The polarity of each port/domain must also be specified. Recall that the polarity of a domain is defined by the location of the domain's toehold. The polarity of a port is denoted by a superscript + or − after the port name in the depiction (e.g. $\mathtt{A}^+$, $\mathtt{B}^-$, ...). Note that this information must appear in the graphical depiction of the node type, but does not need to appear when describing the port elsewhere.

Each domain should contain one or more segments. Segments are also labeled alphabetically, with one exception: segments which are complementary to earlier segments should receive the same label, with a superscript * (e.g. segment $\mathtt{a}^*$ is complementary to a' $\mathtt{b}^*$ is complementary to b$^)$. Segments, like domains, are scoped to a node type, but may be referred to as follows, to avoid ambiguity:
¡node type name¿.¡port/domain name¿.¡segment name¿
(e.g. `m1.A.a`, `m1.A.b`, `m1.B.a`, ...; `m2.A.a`, `m2.A.b`, `m2.B.a`, ...).

Finally, the input-output (or "internal logic") function Λ of the node type may be written underneath the node type. Λ is a function that describes how binding of one set of ports affects another set of ports. It is important to emphasize that Λ is a consequence of the structural features of the implementation, and cannot be set arbitrarily; the notation here is only used as a way of summarizing the behavior. The input-output behavior is written in the form *¡input expression¿→¡output expression¿*, where *¡input expression¿* is a list of one or more input port names (within the node type), separated by logical operators (e.g. ∧, ∨), and *¡output expression¿* is a comma-separated list of output port names (within the node type) that should be opened in this configuration. For example:

$$\mathtt{A} \to \mathtt{B}, \mathtt{C}$$

specifies that input port `A` opens output (or bridge) ports `B` and `C`.

$$\mathtt{A} \wedge \mathtt{B} \to \mathtt{C}, \mathtt{D}$$

indicates that when input ports `a` and `b` are opened, output/bridge ports `c` and `d` are opened. Multiple independent input-output behaviors can be specified, separated by semicolons. For example:

$$\mathtt{A} \to \mathtt{B};\ \mathtt{C} \to \mathtt{D}$$

indicates that `A` opens `B`, while independently, `C` can open `D`.

Rather than specify each node type to be used in each system, one may also choose to simply reference a standard library of node types, such as the one described at the end of this paper.

3.2 Nodes in DyNAML

Once the node types in a system have been described (or referenced), the nodes of the system can be described. For each node, this consists of referencing a node type and specifying complementarity relationships with other nodes.

Each node receives a textual label which is a number preceded by the letter 'n,' (e.g. `n1`, `n2`, `n3`, ...). Ports in a node receive a textual label which is a number preceded by the letter 'p,' (e.g. `p1`, `p2`, `p3`, ...), starting with `p1`, are scoped to the node (i.e. each node's first port is port `p1`), and can be unambiguously referred to with the dot syntax introduced earlier (e.g. `n1.p1`, etc.).[2]

When a node is depicted, the name of the node should appear above the node and the name of the node type should appear in the center. The input-output behavior of a node can be optionally denoted beneath the node using the same syntax described above, although the port names should refer to the ports in the node at hand (e.g. $\mathtt{A} \to \mathtt{B}$ becomes $\mathtt{p1} \to \mathtt{p2}$), if the ports have been renamed. The polarity of each node may be indicated using superscripts, as described above.

Complementarity relationships are depicted with arrows connecting ports. Arrows may be drawn from output ports to input ports. Single-headed arrows must connect ports of opposite polarity (e.g. $^- \to^+$ or

[2]When possible, we use letters (e.g. `A`, `B`, etc.) to refer to components of a *node type*, while we use numbers (e.g. `p1`, `p2`, etc.) to refer to components of a *node*. This serves as a reminder that a domain/port in one node of a given node type is *not* generally the same as the corresponding domain/port in another node of the same node type—for instance, they almost certainly have different sequences.

$^+ \rightarrow^-$) Double-headed arrows can be drawn between bridge ports. No other ports may be connected in any other combination.

3.2.1 Complementarity graph vs. Reaction mechanism

Yin et al. utilize the nodal graphical conventions to draw two types of diagrams of dynamic nucleic acid systems, which convey slightly different types of information. The "complementarity graph" describes the set unique species (nodes) which make up the system, as well as the complementarity relationships between those nodes. The "reaction mechanism" describes the possible reactions which can occur between nodes in the system, based on the complementarity relationships in the complementarity graph. The reaction mechanism can be thought of as the "execution" of the complementarity graph. Fig. 4 demonstrates this distinction. It should be emphasized that *only* the complementarity graph can be programmed; the reaction mechanism is a consequence of the complementarity graph and must be calculated. A "reaction enumerator" can generate a reaction mechanism from a complementarity graph by determining possible reactions between different species in the ensemble. DyNAMiC Workbench includes a reaction enumerator that is able to calculate reaction mechanisms for segment-level systems (described in Chapter **??**), but not for Nodal systems. A reaction enumerator for Nodal systems is an important area for future research.

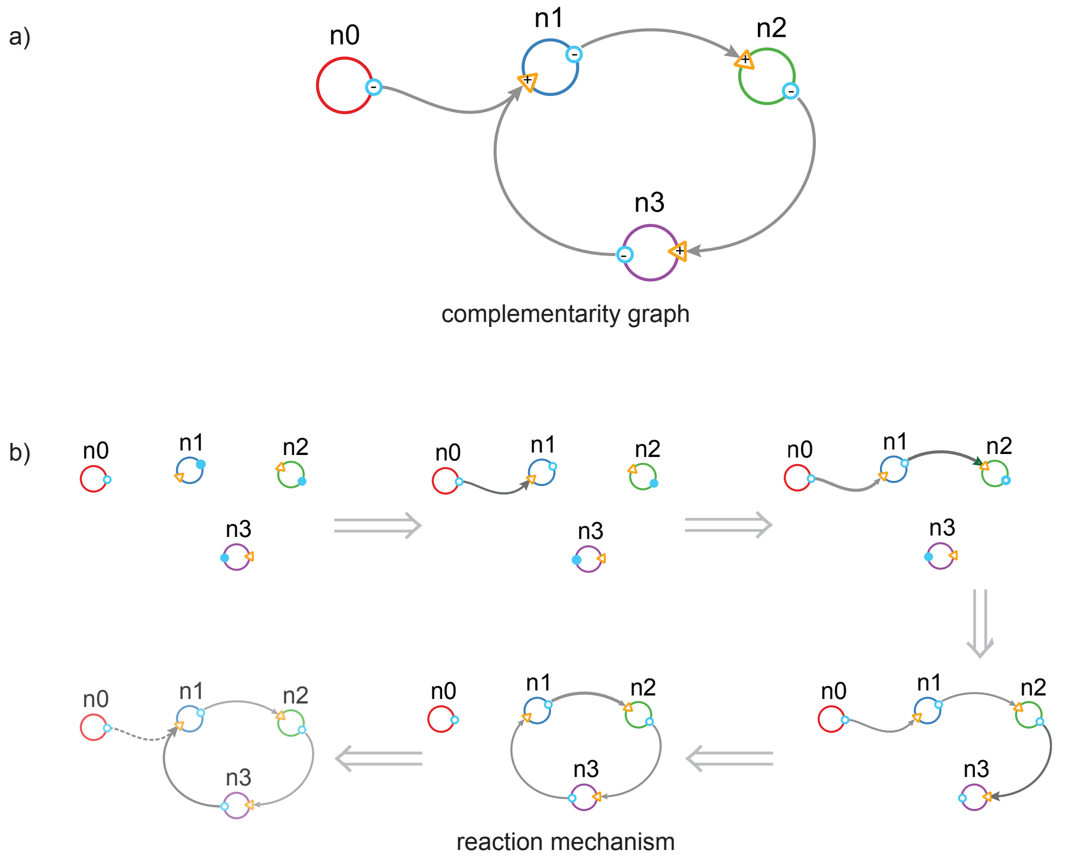


Figure 4: Complementarity Graph vs. Reaction Mechanism **(a)** Complementarity graph showing the desired complementarity relationships between ports on the nodes. **(b)** Reaction mechanism showing anticipated reaction steps, including the opening/closing of ports.

4 Nodal compiler

The goal of the Nodal compiler is to assign a "label" ℓ_s to each segment s on each nodal species in the system. These labels will be integers, such that identical segments (that is, segments which should have the same sequence) must have the same label; any two segments that should be complementary must have labels of the same magnitude, but opposite signs, such that $\ell_s = -\ell_{s^*} \implies \phi s = -\phi s^*$. That is, for some segment s, $\operatorname{sgn}(\ell_s) = \phi(s)$, where sgn is the signum (sign) function. We assume that each segment s on each node in the system can be assigned a unique index $\textsc{Index}(s) \in \{1, \ldots, N_{\text{segments}}\}$, where N_{segments} is the total number of segments in the system. Further, for each segment s, we will use $\textsc{Identity}(s)$ to refer to the identity of the segment, irrespective of its polarity—that is, $\textsc{Identity}(s) = \textsc{Identity}(s^*) = s$, while $\phi(s) = -\phi(s^*)$. For some port p we will represent the strand on which p resides by $\textsc{Strand}(p)$. We will represent the ith segment within p as $p[i]$.

Algorithm 1 Nodal Compiler

```
1: function CompileNodal($X : \langle\text{Complementarity}\rangle, N : \langle\text{Nodes}\rangle$)
2:     global $\phi(n) \forall n \in N$
3:     global $N_{\text{segments}} = \sum_{n \in N} \sum_{d \in n} |d|$                ▷ Count all segments
4:     ComputerNodePolarities
5:     $\mathbf{C} \leftarrow$ BuildComplementarityMatrix()
6:     return LabelSegments
7: end function
```

The input to the compiler a list N of nodes and a list X of complementarity statements $c = (n_s, p_s, n_t, p_t)$ from the source node n_s and the source port p_s, to the target port p_t, on the target node n_t. The algorithm is summarized in Alg. 1. It proceeds three stages:

1. Calculate node polarities (Alg. 2) — traverse each complementarity statement, determining the necessary polarity for the target node based on the relative polarities of the connected ports. Each of complementarity statement (n_s, p_s, n_t, p_t) indicates that $\Phi(n_s.p_s) = -\Phi(n_t.p_t)$; the compiler must determine polarities ϕn_i for each $n_i \in N$, such that these equations are satisfied.

2. Build complementarity matrix (Alg. 3) — generate a $N_{\text{segments}} \times N_{\text{segments}}$ matrix $\mathbf{C}$, specifying the complementarity constraints between each segment in the system. In addition to the *inter*-node complementarities in X, each node may have some set of *intra*-node complementarity relationships determined by the $\mathbf{\Gamma}$ matrix of its node type. For instance, several segments within a hairpin must be complementary for the stem to form. The compiler therefore builds the matrix $\mathbf{C}$ to summarize all of these constraints. Each element $C_{i,j}$ of $\mathbf{C}$ represents a complementarity constraint between two segments; specifically, $C_{\textsc{Index}(s_i),\textsc{Index}(s_j)} = \Phi(s_i) * \Phi(s_j)$. That is, $C_{\textsc{Index}(s_i),\textsc{Index}(s_j)} = -1$ means s_i and s_j should be complementary, $C_{\textsc{Index}(s_i),\textsc{Index}(s_j)} = 1$ means they should be equal, and $C_{\textsc{Index}(s_i),\textsc{Index}(s_j)} = 0$ means they should be orthogonal (they should have different, non-complementary sequences).

3. Label segments (Alg. 4) — generate the segment labeling $\vec{L} = (\ell_1, \ell_2, \ldots, \ell_{N_{\text{segments}}})$. Begin with $\vec{L} = (1, 2, \ldots, N_{\text{segments}})$ and merge elements ℓ_i and ℓ_j by examining entries $C_{i,j}$ of $\mathbf{C}$.

The output is a vector $\vec{L} = (L_1, L_2, \ldots L_{N_{\text{segments}}})^T$ mapping indices to a label for each segment.

5 Standard node types

We propose the following library of standard node types which are likely to be useful for many applications; graphical representations of these node types are given in Figure 5. We describe these behaviors in terms of "signals", where a signal represents a domain, and a signal is present in the system if a domain representing that signal is exposed.

m0 — Initiator Single-strand to begin a cascade by binding an open input.

Algorithm 2 Nodal Compiler: Calculate Node Polarities

```
8: procedure ComputeNodePolarities                         ▷ Calculate node polarities
9:     $\phi(n) \leftarrow 0 \forall n \in N$     ▷ $\phi(n) = 0$ means polarity of $n$ has not been determined.
10:    repeat
11:        for all complementarities $c \in X$ do
12:            $(n_s, p_s, n_t, p_t) \leftarrow c$
13:            if $\phi(n_s) = 0$ then
14:                $x_t \leftarrow \phi(p_s) * \phi(\textsc{Strand}(p_s)) * \phi(n_s) * \phi(p_t) * \phi(\textsc{Strand}(p_t))$
                                                           ▷ Calculate target polarity
15:                if $\phi(n_t) \neq 0 \wedge \phi(n_t) \neq x_t$ then
16:                    Raise error; complementarity relationships conflict.
17:                else
18:                    $\phi(n_t) \leftarrow x_t$
19:                end if
20:            end if
21:        end for
22:    until polarities stop changing
23:    for all nodes $n \in N$ do
24:        if $\phi(n) = 0$ then
25:            $\phi(n) \leftarrow 1$                       ▷ Set all un-assigned polarities
26:        end if
27:    end for
28: end procedure
```

Algorithm 3 Nodal Compiler: Build Complementarity Matrix

```
29: function BuildComplementarityMatrix                  ▷ Build complementarity matrix
30:    $\mathbf{C} \leftarrow 0_{N_{\text{segments}}, N_{\text{segments}}}$     ▷ Initialize complementarity matrix to all zeros
                                                       ▷ Intra-node complementarities
31:    for all nodes $n \in N$ do
32:        for all segments $g < h \in n$ do
33:            if $\textsc{Identity}(\mathrm{g}) = \textsc{Identity}(\mathrm{h})$ then
34:                $C_{\textsc{Index}(g),\textsc{Index}(h)} \leftarrow C_{\textsc{Index}(h),\textsc{Index}(g)} \leftarrow \phi(g) * \phi(h)$
35:            end if
36:        end for
37:    end for
                                                       ▷ Inter-node complementarities
38:    for all $(n_s, p_s, n_t, p_t) \in X$ do
39:        if $(\phi(n_s) * \phi(\textsc{Strand}(p_s))) = \phi(n_t) * \phi(\textsc{Strand}(p_t))$ then
                   ▷ If domains aren't already anti-parallel, flip order of target segments
40:            for all $i \in \{1 \ldots |p_s|\}$ do $C_{\textsc{Index}(p_s[i]),\textsc{Index}(p_t[|p_s|-i])} \leftarrow -1$
41:            end for
42:        else
43:            for all $i \in \{1 \ldots |p_s|\}$ do $C_{\textsc{Index}(p_s[i]),\textsc{Index}(p_t[i])} \leftarrow -1$
44:            end for
45:        end if
46:    end for
47:    return C
48: end function
```

Algorithm 4 Nodal Compiler: Generate segment labels

```
49: function LabelSegments($\mathbf{C}$)                                      ▷ Label segments
50:     $\vec{L} \leftarrow (1, 2, \ldots N_{\text{segments}})^T$                 ▷ Assign all segments unique labels
51:     for all $i \in \{1 \ldots N_{\text{segments}}\}$ do                      ▷ Traverse $\mathbf{C}$, merging segment labels
52:         for all $j \in \{1 \ldots N_{\text{segments}}\}$ do
53:             if $C_{i,j} = -1$ then                                         ▷ Segments must be complementary
54:                 $\ell \leftarrow -\min |L_i|, |L_j| * \phi(L_i) * \phi(L_j)$   ▷ All instances of both segments get the smaller label
55:                 $L_k \leftarrow \ell \; \forall L_k = (L_j \text{ if } |\ell| = |L_i| \text{ else } L_i)$
56:             else if $C_{i,j} = 1$ then                                     ▷ Segments must be equal
57:                 $\ell \leftarrow \operatorname{argmin} |L_i|, |L_j|$           ▷ All instances of both segments get the smaller label
58:                 $L_k \leftarrow \ell \; \forall L_k = (L_j \text{ if } |\ell| = |L_i| \text{ else } L_i)$
59:             end if
60:         end for
61:     end for
62:     return $\vec{L}$
63: end function
```

m1 — One-output transducer Transduces input signal to output signal; binds to the complex displaying the input signal.

m2 — Two-output transducer Transduces input to two different outputs; binds to the complex displaying the input signal.

m3 — Triggered bridge Allows formation of a ring-closing “bridge” upon binding the input.

m4 — Triggered bridge, One-output Allows formation of a ring-closing bridge upon binding the input; transduces input to output.

m5 — One-output, Triggered bridge Allows formation of a ring-closing bridge upon binding the input; transduces input to output.

m6 — Cooperative complex Allows binding of two inputs; binds irreversibly complexes displaying both input signals *only* when both inputs are present.

m7a–d, m8a–d — Boolean AND Allows binding of two inputs; displays the output signal only when both inputs are present.

m9 — Boolean OR Allows binding of two inputs; transduces to the output when either of the inputs are present.

6 Textual representations

While graphical representations are appropriate for human design of dynamic systems, a method of serializing this graphical representation into a semantically meaningful form is necessary; therefore, we propose a JSON-based textual interchange format, called **DyNAML** (**Dy**namic **N**ucleic **A**cid **M**arkup **L**anguage) for encoding such systems.

DyNAML is used to describe the *components* of a nucleic acid system in declarative manner. That is, DyNAML describes the complementarity graph. Specifically, DyNAML documents capture:

- Node types
- Nodes
- Complementarities

DyNAML documents do not encode or capture anything about *mechanism* or *experimental conditions*, e.g.:

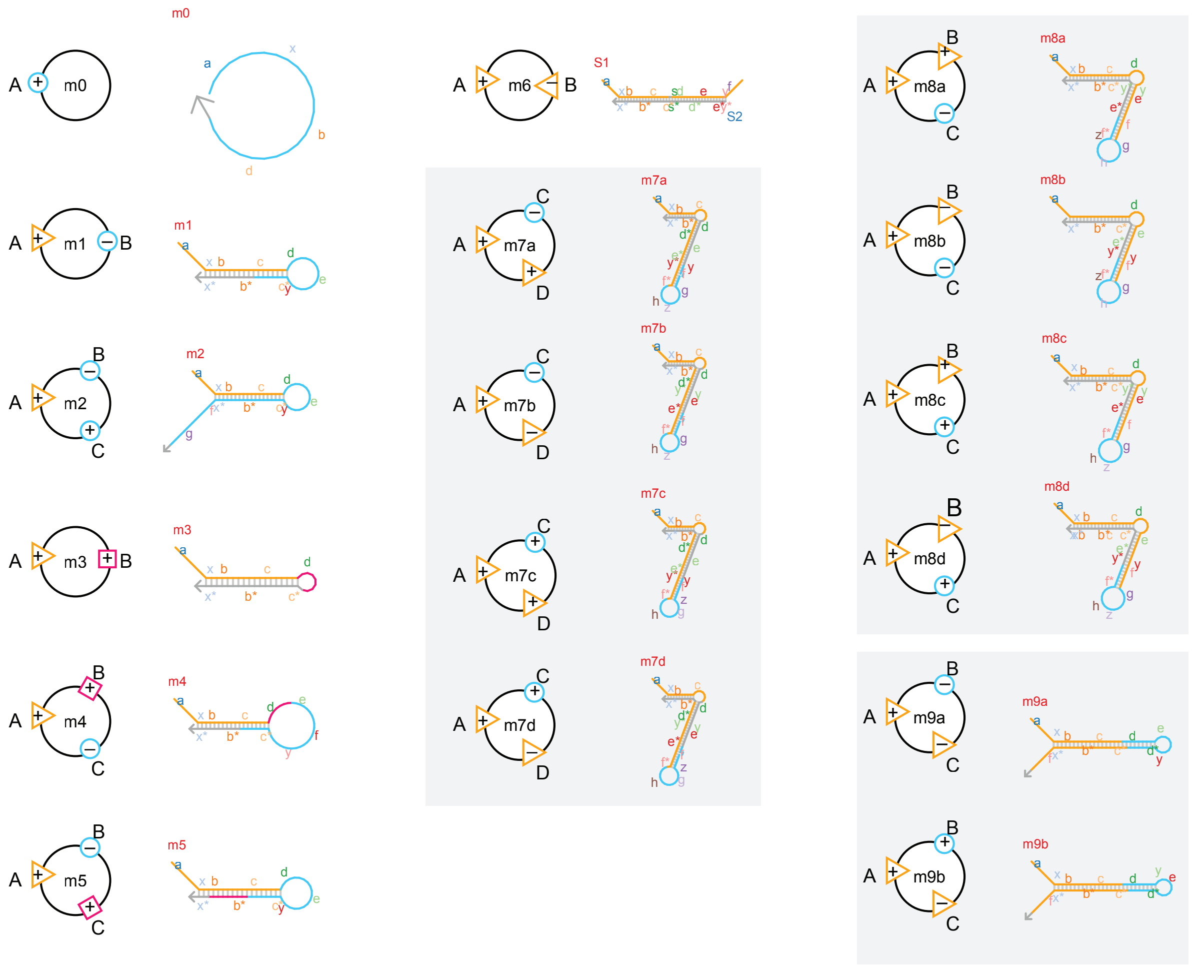


Figure 5: (Reproduced from Fig. **??**) Built-in node types. These node types and their corresponding molecular implementation are bundled with DyNAMiC Workbench, and can be added to Nodal designs via a simple drag-and-drop mechanism. Node types are labeled with a lowercase 'm', a number, and a letter representing a particular variation (e.g. m7a, m7b, m7c, and m7d are all variations of a single shape m7). Node types are based on [1, 8–10].

- Reactions
- Complexes of nodes
- System states
- Concentrations
- . . . anything else about the state or emergent behavior of the system

Much of this information can be calculated from a DyNAML specification and compiled into an interchange format, such as the Pepper Intermediate Language [5].

6.1 DyNAML documents

DyNAML documents are written in Javascript Object Notation (JSON) [3]. In JSON, objects or "elements" are given by comma-separated lists of quoted *key*:*value* pairs ("properties"), surrounded by curly braces:

```
{ "key": "value", "key 2": "value 2", "key 3": "value 3" }
```

and lists ("arrays") are surrounded by square brackets:

```
{ "item 1", "item 2", "item 3" }
```

The root object of all DyNAML documents is known as the "library" has two required properties, each containing an array of child elements (described below): `motifs` and `nodes`. In addition, a `import` property may be specified to load node types and/or nodes from other libraries, including the built-in library of standard node types.

6.2 Node types in DyNAML

Node types in DyNAML are represented with the `motifs` array. Node types are described by enumerating, naming, and describing the domains of the node type, then assigning `roles` to each domain to expose domains to the nodal programmer. Domains are in turn described by enumerating and naming each of their constituent segments. Several examples on the following pages illustrate this (see List. 1, 2, 3, 9).

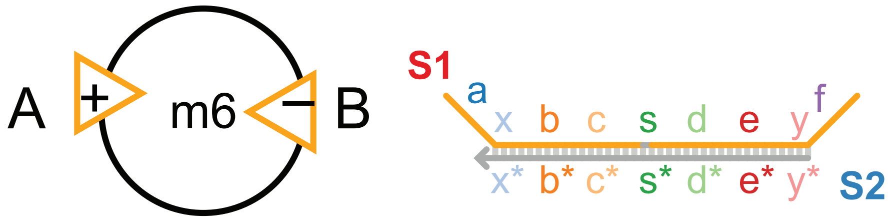


Figure 6: Node type `m6`; DyNAML source code shown in List. 3.

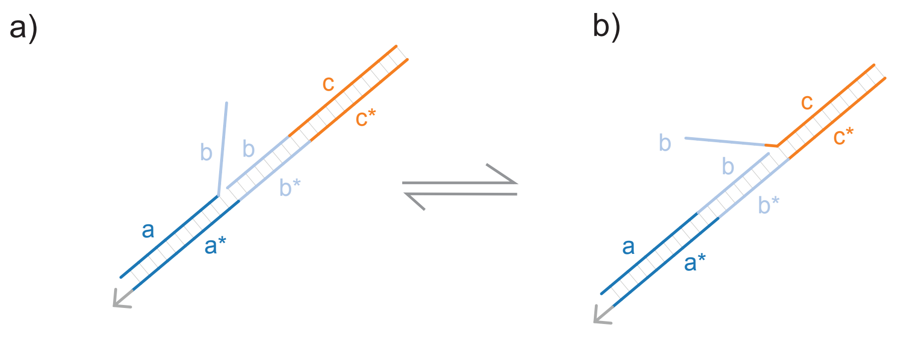


Figure 7: Complementarities vs. Secondary Structure. There may be internal complementarity relationships within a complex; for instance, although the secondary structure of a complex may be as shown in **(a)**, there is an additional complementarity with `b`, such that the complex may adopt the configuration in **(b)**.

6.2.1 Complementarities and secondary structure

The strand(s) which comprise a node type must have at least one meta-stable secondary structure, reflecting the node type's internal hybridization state (e.g. which segments within the node type are hybridized in the node type's initial configuration). While a particular secondary structure may imply a certain number of complementarity relationships (or a set of complementarity relationships may imply only one secondary structure), this is not always the case. For instance, a node type may have multiple meta-stable secondary structures, and therefore there may be more complementarity relationships between segments in the node type than those implied by a single secondary structure (see Fig. 7). For this reason, DyNAML allows the programmer to specify both secondary structure and complementarity relationships. Secondary structure(s) are defined on node type elements, using the `structure` property, whereas complementarities are defined separately.

Complementarities may be specified within `segment` elements. To specify complementarities within the `segment` element, the `identifier` property is used — segments which should have the same sequence should have the same `identifier`, whereas segments which should be complementary should have complementary `identifiers` (e.g. `identifier: "a"` is complementary to `identifier: "a*"`); see List. 2 for a full example.

6.2.2 `motif` elements

Node types are represented by `motif` elements. `motif` objects may contain the following properties:

`name` the node type name, following above conventions.

`type` the type of species (or set of species) implementing the node type. Currently, only `hairpin`, `initiator` (for linear, single-stranded initiators), and `cooperative` (for cooperative complexes, as in [9]) are supported.

`polarity` the order in which strands will appear; `+` or `1` for $5' \rightarrow 3'$, `-` or `-1` for $3' \rightarrow 5'$.

`strands` an array of one or more `strand` elements, as described below.

`structure` specifies the initial, meta-stable secondary structure of the strand(s) comprising the node type.

6.2.3 `strand` elements

`strand` objects may contain the following properties:

`name` the strand name.

`polarity` the order in which segments will appear; `+` or `1` for $5' \rightarrow 3'$, `-` or `-1` for $3' \rightarrow 5'$.

`structure` specifies the meta-stable secondary structure(s) of this strand. Alternatively, this property may be omitted from each `strand` and the `structure` property of the `motif` may be specified instead.

`domains` an array of one or more `domain` elements, as described below.

6.2.4 `domain` elements

`domain` elements represent regions of a strand that expose a functional behavior. Each `domain` element must have a `segments` array containing one or more `segment` elements. `domain` elements can also have the following attributes:

`name` the domain/port name, following above conventions (scoped to node type)

`role` the role of the corresponding port. If this domain is not to be exposed as an interacting port, this attribute may be omitted, and the domain will be assigned a role of "structural". Structural domains may be useful for, for instance, sequestering ring-closing "bridge" domains within a loop.

`type` the particular structural feature of the domain which implements the input-output behavior; options for this parameter will depend on the `type` of the parent node type. This parameter is optional, and may be specified by the programmer for descriptive or error-checking purposes; however, it may also be omitted, in which case it will be generated by the compiler by applying a regular expression to the secondary structure. The compiler will recognize the components of `type="hairpin"` node types, but may not recognize such components for other node types. For `type="hairpin"` node types, the following options are available:

`input` an orange input port

`loop` a blue output port which is partially sequestered in the loop of the hairpin

`tail` a green output port which is partially sequestered within the hairpin stem

`loop-bridge` a pink bridge port within the loop

`tail-bridge` a purple bridge port within the tail

`polarity` the location of the toehold segment (corresponding the the relative polarity, ϕ, of this domain). `+` if the toehold is on the end closer to the positive pole, `-` if the toehold is on the end closer to the negative pole.

6.2.5 `segment` elements

`segment` elements describe distinct regions on the strand to be handled by the sequence designer. Segment elements should contain the following attributes:

`name` the segment name, following above conventions (scoped to node type)

`identifier` the identifier of the segment. Can be used to specify complementarity relationships within the node type— i.e. a segment with `identifier: "a"` will be complementary to a segment with `identifier: "a*"` (within the same node type), and two segments with identical identifiers (within the same node type) will have identical sequences.

`role` the role of the segment; one of `toehold` (short toehold segment which initiates branch migration by binding to a complementary sequence of the same length) or `clamp` (segment which "pads" toeholds sequestered in the hybridized region, distancing them from the exposed single-stranded region in order to reduce leakage). Segment roles may be used by the compiler to provide additional guidance to the sequence designer.

6.2.6 `structure` elements

A `structure` element describes a single unique meta-stable secondary structures which may be adopted by the strands in a particular node type. Structure elements may be specified as JSON objects or as strings. When specified as objects, `structure` elements can have the following properties

`value` A string containing a secondary structure specification encoded in dot-bracket [11] or HU+/DU+ [12] notation. The secondary structure is depicted segment-wise. That is, each "unit" (e.g. a dot or bracket) of this string will refer to a single *segment* — not to a single base, as is usually the convention. Likewise, the numbers in an HU+/DU+ string will refer to numbers of segments, rather than numbers of bases.

`format` The name of the format used to encode `value`; one of: `dot-paren` (or `dot-bracket`), `HU+`, or `DU+`.

As a short-hand, a string can be specified in place of the entire object; the string should contain the structure specification in one of the above formats; the `format` used will be inferred by the compiler.

6.2.7 Single-stranded Node types

As a shorthand, a node type containing a single strand may be represented using the `domains` attribute directly, in place of the `strands` attribute. The `domains` array must contain one or more `domain` elements. The Nodal compiler will automatically generate a single `strand` element to represent the node type, populated with the appropriate `domains`. Note: all named `domain` and `segment` elements remain scoped to the node type.

6.3 Nodes

The root object should have a `nodes` array, containing one or more `node` elements. Nodes are described by invoking a node type (or describing a node type inline; see Sec. 6.3.5), naming domains, optionally specifying complementarities, and optionally specifying values for various other design parameters. For a full example, see Figure 4.

6.3.1 `node` elements

`node` elements may have the following properties:

`name` the node name, following above conventions

`motif` the node type to be invoked to describe this node's implementation. This may optionally be omitted if the node type is described inline (see Sec. 6.3.5).

`complementarities` **or** `complements` an array of `complement` objects. Each `complement` object specifies a complementarity between this node and another node in the system.

other Nodes may also specify other parameters, called directives. Directives are described below.

6.3.2 `complement` elements

`complement` elements allow complementarities to be specified between domains a given node and domains in another node. For each `complement` element, this element's parent `node` is known as the *source* node; the other node is the *target* node.

`complement` elements may have the following attributes:

`source` the name of the port on the source node

`node` the name of the target node

`target` the name of the port on the target node

6.3.3 Directives

Additional parameters of objects, generally related to sequence design, can be specified with directives. A directive refers to the declaration of a parameter. While a node may be subject to multiple directives for a given parameter, each parameter will ultimately have only one value. This is discussed below in "Cascading Parametrization."

To specify a directive, an additional property is simply added to the `segment`, `domain`, `node`, or even root element. A a general rule, directives not recognized by the compiler will be ignored. This allows forwards-compatibility among directives. Several directives are available; however, many more directives will likely become available in the future as more sequence designers are implemented.

`toeholdLength` accepts a number of nucleotides specifying the length of toehold segments (`segment`s with `role="toehold"`) in this node; defaults to `8`.

`clampLength` accepts a number of nucleotides specifying the length of of clamp segments (`segment`s with `role="clamp"`) in this node; defaults to `2`.

6.3.4 Cascading parametrization

The final set of parameters assigned to each component of a given node is calculated as follows:

1. If a directive exists within a segment, the parameter is given by that directive.
2. Else, if a directive exists within a domain, the parameter is given by that directive.
3. Else, if a directive exists within a node, the parameter is given by that directive.
4. Else, if a parameter can be calculated by a complementarity relationship, the value of the parameter is given by that directive. For example, segment lengths must match between complementary domains.
5. Else, if a directive exists within a system, the parameter is given by that directive.
6. Else, if no directive is given, the parameter is assigned by the compiler.

In general, directives do not need to be explicitly specified if they match default values or can be calculated from complementarity relationships. They may be specified explicitly for error-checking purposes, as an error will be thrown by the compiler if there are conflicting directives (see Section 7.1).

6.3.5 Inline Node types

Normally, node types are defined separately from nodes; this allows a node type to be re-used among several nodes. However, sometimes it may be useful to define a node type solely for use in a single node—that is, directly specify the molecular implementation of a single nodal species. In this case, an "inline node type" may be used.

To define an inline node type, a normal `node` element is simply decorated with a `domains` or `strands` property. These are specified exactly as one would specify a node type. An example is demonstrated in List. 8.

6.4 `import` elements

In addition to `motifs` and `nodes`, DyNAML libraries can also import node types and nodes from other libraries. Currently this functionality only supports importing node types from the standard library of built-in node types (Sec. 5). The `import` property of the library should contain an array of `import` elements, each with the following properties:

`type` the kind of object to import; either `motif` or `node` (currently only `motif` is supported).

`name` the name of the object to import.

If the `import` property is used, the `version` property should also be added to the library, containing the version of the standard library to be loaded—this allows changes to the standard library to be made without breaking existing systems. The current version is `1`. List. 9 provides an example importing the built-in node types from the standard library.

6.5 Inline libraries

Finally, an entire DyNAML library may be "embedded" for re-use as a node type. The effect of this process is similar to creating an entirely separate DyNAML library, compiling it, and building a node type out of the result. To do this, two properties are added to a `motif` element.

`motifs` a list of `motif` elements to be declared or used within the embedded library.

`nodes` a list of `node` elements to be created within the embedded library.

The `motif` and `node` elements in an embedded library have exactly the same form as those in any other DyNAML library. However, the `domain` elements have one additional property:

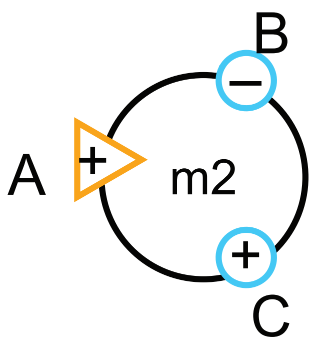
`expose` exposes this domain as a port in the outer node type. This object should have the following properties:

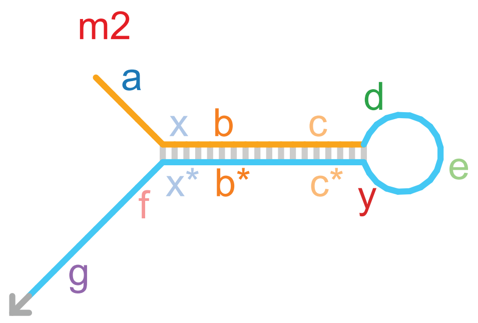
- `name` name that the exposed port should be given
- `role` role of the exposed port

This property is needed in order to specify which domains within the inner library should be exposed as ports on the resulting outer node type. domains which have no such statement will not be available for binding to other species when the node type is instantiated.

Listing 1 This DyNAML fragment describes the built-in hairpin node type `m2`.

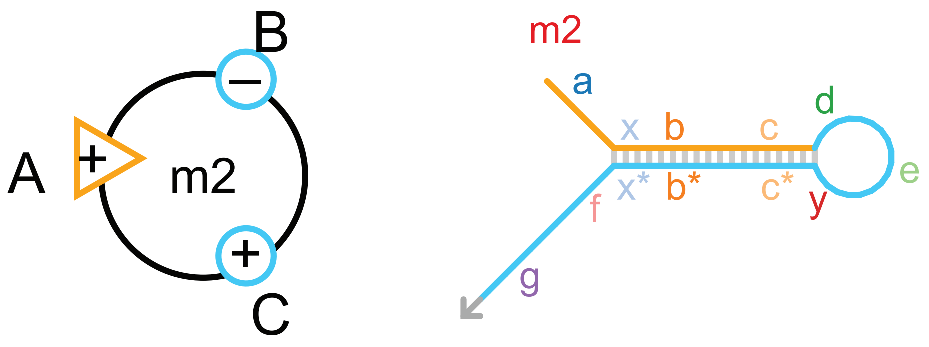
```
{
        "motifs": [{
                "name":"m2",
                "type":"hairpin",
                "structure": ".(((...)))..",
                "domains":[{
                        "name":"A",
                        "role":"input",
                        "polarity":"-",
                        "segments":[
                                { "role":"toehold", "identifier":"a" },
                                { "role":"clamp",   "identifier":"x" },
                                { "role":"",        "identifier":"b" },
                                { "role":"",        "identifier":"c" }
                        ]
                },{
                        "name":"B",
                        "role":"output",
                        "polarity":"-",
                        "segments":[
                                { "role":"",        "identifier":"d" },
                                { "role":"",        "identifier":"e" },
                                { "role":"clamp",   "identifier":"y" },
                                { "role":"toehold", "identifier":"c*" },
                        ]
                },{
                        "name":"C",
                        "role":"output",
                        "polarity":"+",
                        "segments":[
                                { "role":"toehold", "identifier":"b*" },
                                { "role":"clamp",   "identifier":"x*" },
                                { "role":"",        "identifier":"f" },
                                { "role":"",        "identifier":"g" },
                        ]
                }]
        }]
}
```





Listing 2 This DyNAML fragment describes the built-in hairpin node type `m4`.

```
{
        "motifs": [{
                "name":"m4",
                "type":"hairpin",
                "structure": ".(((....)))",
                "domains":[{
                        "name":"A",
                        "role":"input",
                        "polarity":"-",
                        "segments":[
                                { "role":"toehold", "identifier":"a" },
                                { "role":"clamp",   "identifier":"x" },
                                { "role":"",        "identifier":"b" },
                                { "role":"",        "identifier":"c" }
                        ]
                },{
                        "name":"B",
                        "role":"bridge",
                        "polarity":"+",
                        "segments":[
                                { "role":"toehold", "identifier":"d" }
                        ]
                },{
                        "name":"C",
                        "role":"output",
                        "polarity":"-",
                        "segments":[
                                { "role":"",        "identifier":"e" },
                                { "role":"",        "identifier":"f" },
                                { "role":"clamp",   "identifier":"y" },
                                { "role":"toehold", "identifier":"c*" }
                        ]
                },{
                        "name":"D",
                        "role":"structural",
                        "segments":[
                                { "role":"clamp",   "identifier":"x*" },
                                { "role":"",        "identifier":"b*" }
                        ]
                }]
        }]
}
```

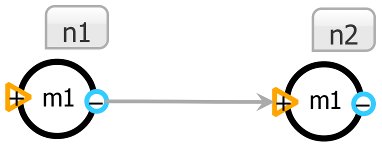


Listing 3 This DyNAML fragment describes the built-in multi-stranded cooperative complex node type `m6`, depicted in Fig. 6.

```
{
        "motifs": [{
                "name":"m6",
                "type":"cooperative",
                "structure": ".(((((((.+)))))))",
                "strands":[{
                        "name":"S1",
                        "domains":[{
                                "name":"A",
                                "role":"input",
                                "polarity":"+",
                                "segments":[
                                        { "role":"toehold", "identifier":"a" },
                                        { "role":"clamp",   "identifier":"x" },
                                        { "role":"",        "identifier":"b" },
                                        { "role":"",        "identifier":"c" }
                                ]
                        },{
                                "name":"S",
                                "role":"structural",
                                "segments":[
                                        { "role":"", "identifier":"s", "length":1 }
                                ]
                        },{
                                "name":"B",
                                "role":"input",
                                "polarity":"-",
                                "segments":[
                                        { "role":"",        "identifier":"d" },
                                        { "role":"",        "identifier":"e" },
                                        { "role":"clamp",   "identifier":"y" },
                                        { "role":"toehold", "identifier":"f" }
                                ]
                        }]

                },{
                        "name":"S2",
                        "domains":[{
                                "name":"D",
                                "role":"structural",
                                "segments":[
                                        { "role":"clamp",   "identifier":"y*" },
                                        { "role":"",        "identifier":"e*" },
                                        { "role":"",        "identifier":"d*" },
                                        { "role":"",        "identifier":"s*", "length":1 },
                                        { "role":"",        "identifier":"c*" },
                                        { "role":"",        "identifier":"b*" },
                                        { "role":"clamp",   "identifier":"x*" }

                                ]
                        }]

                }]
        }]
}
```

Listing 4 A simple DyNAML system with two nodes, each of standard node type shape `m1`. `n1.p2` is complementary to `n2.p2`.

```
{
        "import": [{"type": "motif", "name": "m1"}],
        "nodes": [{
                "name": "n1",
                "motif": "m1",
                "domains": [
                        { "name": "p1", "identifier": "A" },
                        { "name": "p2", "identifier": "B" }
                ],
                "complements": [{
                        "source": "p2", "node": "n2", "target": "p2"
                }]
        },{
                "name": "n2",
                "motif": "m1",
                "domains": [
                        { "name": "p1", "identifier": "A" },
                        { "name": "p2", "identifier": "B" },
                ],
        }]
}
```



Listing 5 The `toeholdLength` directive allows the programmer to specify the length of toehold segments for a domain, node, or entire system.

```
{
        "import": [{"type": "motif", "name": "m1"}],
        "nodes": [{
                "name": "n1",
                "motif": "m1",
                "domains": [
                        { "name": "p1", "identifier": "A" },
                        { "name": "p2", "identifier": "B" }
                ],
                "toeholdLength": 6
        }]
}
```

Listing 6 The `toeholdLength` directive is applied only to `n1`; however, the `complement` implies that the `toeholdLength` of `n2` should also equal `6`.

```
{
        "import": [{ "type": "motif", "name": "m1"}],
        "nodes": [{
                "name": "n1",
                "motif": "m1",
                "domains": [
                        { "name": "p1", "identifier": "A" },
                        { "name": "p2", "identifier": "B" }
                ],
                "complements": [
                        { "source": "p2", "node": "n2", "target": "p2"}
                ],
                "toeholdLength": 6
        },{
                "name": "n2",
                "motif": "m1",
                "domains": [
                        { "name": "p1", "identifier": "A" },
                        { "name": "p2", "identifier": "B" }
                ]
        }]
}
```

Listing 7 Cascading parameter directives which conflict result in a compiler error. In this case, the `toeholdLength` of `n2` is set explicitly at 8; however, the complementarity with `n1` implies that the `toeholdLength` must be 6. the compiler will report an error identifying the node at which the conflict is generated, as well as the contradictory constraints (see Errors below for more detail).

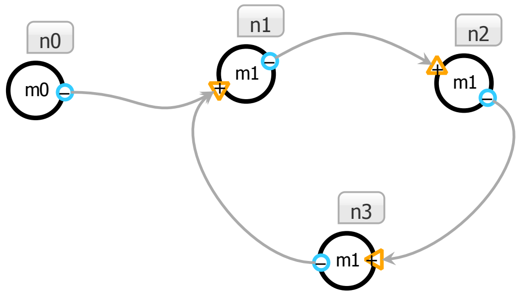
```
{
        "import": [{ type: "motif", name: "m1" }],
        "nodes": [{
                "name": "n1",
                "motif": "m1",
                "domains": [
                        { "name": "p1", "identifier": "A" },
                        { "name": "p2", "identifier": "B" }
                ],
                "complements": [
                        { "source": "p2", node: "n2", target: "p2"}
                ],
                "toeholdLength": 6
        },{
                "name": "n2",
                "motif": "m1",
                "domains": [
                        { "name": "p1", "identifier": "A" },
                        { "name": "p2", "identifier": "B" }
                ]
                "toeholdLength": 8
        }]
}
```

Listing 8 Optionally, a single-use node type may be specified within a `node` element. Rather than specifying a `motif` property, the node simply specifies its own `domains` property, populated with complete `domain` elements. This kind of node is treated exactly like a node which had inherited a pre-defined node type by specifying a `motif` property.

```
{
        "import": [{type: "motif", name: "m1"}],
        "nodes": [{
                "name": "n1",
                "domains": [{
                        "name": "p1",
                        "role": "input",
                        "polarity": "+",
                        "segments": [
                                { "role":"toehold",  "identifier":"a" },
                                { "role":"clamp",    "identifier":"b" },
                                { "role":"clamp",    "identifier":"c" },
                                { "role":"toehold",  "identifier":"d" }
                        ]
                },{
                        "name": "p2",
                        "role": "output",
                        "polarity": "-",
                        "segments": [
                                { "role":"clamp",    "identifier":"e"  },
                                { "role":"clamp",    "identifier":"f"  },
                                { "role":"toehold",  "identifier":"d*" },
                                { "role":"clamp",    "identifier":"c*" },
                                { "role":"clamp",    "identifier":"b*" }
                        ]
                }],
        },{
                "name": "n2",
                "motif": "m1",
                "domains": [
                        { "name":"p1", "identity":"a" },
                        { "name":"p2", "identity":"b" }
                ]
        }]
}
```

Listing 9 3-arm junction system in DyNAML. This is the same system discussed in detail in Chapter **??**, implemented in DyNAML.

```
{
  "import": [
    { "type": "motif", "name": "m0" },
    { "type": "motif", "name": "m1" }
  ],
  "nodes": [
    {
      "name": "n0",
      "motif": "m0",
      "polarity": 1,
      "complementarities": [
        { "source": "A", "target": "A", "node": "n1" }
      ]
    },
    {
      "name": "n1",
      "motif": "m1",
      "polarity": 0,
      "complementarities": [
        { "source": "B", "target": "A", "node": "n2" }
      ]
    },
    {
      "name": "n2",
      "motif": "m1",
      "polarity": 0,
      "complementarities": [
        { "source": "B", "target": "A", "node": "n3" }
      ]
    },
    {
      "name": "n3",
      "motif": "m1",
      "polarity": 0,
      "complementarities": [
        { "source": "B", "target": "A", "node": "n1" }
      ]
    }
  ],
  "motifs": [ ],
  "version": 1
}
```



7 Implementation

We have implemented the Nodal compiler described above as a Javascript library. The library accepts as input a string of DyNAML code, and produces output in one of several formats described in Sec. 7.3 below. The library also has a flexible API for creating and manipulating node type and node objects. This choice to implement the compiler in Javascript allows it to be run either as a command-line tool or within a web browser. DyNAMiC Workbench uses the compiler in both modes—while the user designs a system using graphical Nodal user interface (see Fig. **??**), the compiler runs asynchronously in the background, recompiling the system on the fly as it is modified. This allows errors to be detected and highlighted immediately, as well as the compiled secondary structure of individual nodes to be visualized. When the user wishes to generate a full segment-level representation of the system, or produce an output file to be read by another program (see Sec. 7.3 below), DyNAMiC Workbench invokes the command-line version of the compiler and saves the output file(s).

7.1 Errors

When compiling a DyNAML document, the compiler may encounter errors which should be reported in a semantically meaningful way. This allows tools such as the DyNAMiC Workbench interface to graphically report errors in the two-dimensional interface. Errors are reported as JSON objects, and have the following standard set of properties:

type String specifying the general class of the error raised. A list of error `type`s is below.

message String containing an human-readable message describing the error. The message may include human-readable references to the nodes/ports/node types generating the error, but this is not required if references are provided in the `nodes` field.

nodes Array of pseudo-`node` objects involved in generating the error. This is used to highlight error-causing nodes in a graphical interface.

line Line number at which the error occurred, if applicable.

column Column number at which the error occurred, if applicable.

The following is a list of error `type`s which may occur in DyNAML programs:

polarity conflict Complementarity relationships dictate that a segment must be complementary to two segments of different polarities.

domain length mismatch Complementarity was indicated between two domains with incompatible numbers of segments.

segment length mismatch Complementarity relationships require segments of different lengths be complementary.

illegal connection Connection of one port to an incompatible downstream port type (e.g. input to input, output to output, or bridge to input/output).

7.2 Warnings

The compiler may also on occasion issue warnings. Warnings are non-fatal messages which may provide important information to the programmer about possible issues with the program. Warnings have all the same properties as errors, but they do not halt execution of the compiler.

The following is a list of warning `type`s which may occur in DyNAML programs:

complementarity warning Complementarities may occur between segments within a node which are not due to intra-segment complementarities specified by the node type. That is, there are unintended complementary segments within a node type which may cause unintended secondary structure and hamper execution of the complementarity graph.

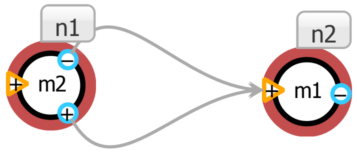


Figure 8: An example of a Nodal configuration that would produce a DyNAML error. The error generated is: "Complementarity statement in node n1 implies node n2 should have polarity -1, but instead it has polarity 1"

7.3 Intermediate Language and Output formats

Once the Nodal compiler has generated a labeling for the segments in the system (see Sec. 4), this information is stored in a special type of DyNAML document, called a DIL (**D**yNAML **I**ntermediate **L**anguage) document. DIL documents are identical to DyNAML documents, with the following restrictions: all nodes must be compiled, and all segments must have globally-unique `identifier`s. The labeling generated by the compiler is thus stored by overwriting the `identifier` field on each `segment` element with the calculated label.

Once a DIL document has been produced, our implementation of the Nodal compiler is able to generate several types of output files:

Design script for Web DD — script that allows sequences to be design with the stochastic domain designer DD. [13]

Design script for NUPACK Multi-objective sequence designer — script that allows sequences to be designed with the NUPACK thermodynamic sequence design package. [14]

Design script for Multisubjective sequence designer — script that allows sequences to be designed with the Multisubjective sequence design package. [15]

Pepper Intermediate Language representation — an alternative text-based representation used by several pieces of software from Erik Winfree's group. [5]

Input file for segment-level reaction enumerator — script that allows reactions between species in the system to be enumerated at a segment level by the reaction enumerator (Chapter **??**).

Scalable Vector Graphics (`.svg`) image — graphical depiction of the strands comprising the implementation of the system.

DyNAMiC Workbench contains rich utilities for visualizing and editing the DIL representation of a system, which can be used to (for instance) edit the composition of a particular strand or impose sequence constraints on the system.

8 Discussion

We have presented a formal specification of the Nodal language—a flexible programming language for dynamic DNA nanosystems, as well as a compiler. The Nodal language allows for a wider range of structural motifs than previous compilers. Specifically, hairpin structures and branched structures (such as n-arm junctions) containing multiloops, which are impossible to represent in the formalism of DSD [16] or using the Seesaw-gate abstraction [6]. The Nodal language allows for a high level of abstraction—structures are represented as abstract behavioral units (node types), and these can be composed without regard to their internal molecular implementation. Finally, the implementation of the Nodal compiler is able to produce output that can be transferred to numerous other tools—sequence designers, reaction enumerators, etc.

References

[1] P. Yin, H. M. T. Choi, C. R. Calvert, and N. A. Pierce, “Programming biomolecular self-assembly pathways.,” *Nature*, vol. 451, pp. 318–322, Jan. 2008.

[2] D. Y. Zhang, A. J. Turberfield, B. Yurke, and E. Winfree, “Engineering Entropy-Driven Reactions and Networks Catalyzed by DNA,” *Science*, vol. 318, pp. 1121–1125, Nov. 2007.

[3] Ecma International, “The JSON Data Interchange Format,” 2013.

[4] M. R. Lakin, S. Youssef, F. Polo, S. Emmott, and A. Phillips, “Visual DSD: a design and analysis tool for DNA strand displacement systems.,” *Bioinformatics*, vol. 27, pp. 3211–3213, Nov. 2011.

[5] S. Ligocki, C. Berlind, J. Schaeffer, and E. Winfree, “PIL Specification,” Nov. 2010.

[6] L. Qian and E. Winfree, “A simple DNA gate motif for synthesizing large-scale circuits,” *J. R. Soc. Interface*, Feb. 2011.

[7] D. Soloveichik, G. Seelig, and E. Winfree, “DNA as a universal substrate for chemical kinetics,” *Proc. Natl. Acad. Sci. U. S. A.*, vol. 107, pp. 5393–5398, Jan. 2010.

[8] G. Seelig, D. Soloveichik, D. Y. Zhang, and E. Winfree, “Enzyme-Free Nucleic Acid Logic Circuits,” *Science*, vol. 314, pp. 1585–1588, Dec. 2006.

[9] D. Y. Zhang, “Cooperative hybridization of oligonucleotides,” *J. Am. Chem. Soc.*, 2011.

[10] J. P. Sadowski, C. R. Calvert, D. Y. Zhang, N. A. Pierce, and P. Yin, “Developmental Self-Assembly of a DNA Tetrahedron ,” *ACS Nano*.

[11] I. Hofacker, W. Fontana, P. Stadler, L. Bonhoeffer, M. Tacker, and P. Schuster, “Fast folding and comparison of RNA secondary structures,” *Monatshefte für Chemie/Chemical Monthly*, vol. 125, no. 2, pp. 167–188, 1994.

[12] J. N. Zadeh, B. R. Wolfe, and N. A. Pierce, “Nucleic acid sequence design via efficient ensemble defect optimization.,” *J Comput Chem*, vol. 32, pp. 439–452, Feb. 2011.

[13] D. Y. Zhang, “Towards domain-based sequence design for DNA strand displacement reactions,” *Lecture Notes in Comput. Sci.*, vol. 6518, pp. 162–175, 2011.

[14] J. N. Zadeh, C. D. Steenberg, J. S. Bois, B. R. Wolfe, M. B. Pierce, A. R. Khan, R. M. Dirks, and N. A. Pierce, “NUPACK: Analysis and design of nucleic acid systems.,” *J Comput Chem*, vol. 32, pp. 170–173, Jan. 2011.

[15] J. P. Sadowski, “Multisubjective Sequence Designer,”

[16] A. Phillips and L. Cardelli, “A programming language for composable DNA circuits.,” *J. R. Soc. Interface*, vol. 6 Suppl 4, pp. S419–36, Aug. 2009.