Formal graphical and textual descriptions of dynamic nucleic acid hairpin-based systems

Casey Grun Justin Werfel Peng Yin
Wyss Institute for Bioinspired Engineering, Harvard University

April 11, 2012

1 Introduction and motivation

Yin et al.[3] described a method of representing dynamic systems of self-assembling and dis-assembling nucleic acid species using a simple, graphical "reaction graph" abstraction. In this abstraction, "nodes" represent individual strands (or collections of strands) of DNA. "Ports" on a node represent domains which can interact with ports on other nodes to trigger changes in secondary structure. This abstraction is interesting to us for several reasons:

Recursive 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. Likewise, although the examples in this paper all involve hairpin chain reactions (HCR), nodal programs could just as easily describe various other underlying implementations, such as triggered origami opening and aggregation or entropy-driven branch migrations[5].

Geometry — nodal programs are written in 2D space, and nodal "motifs" can encode geometric information.

Here we formalize the conventions of the graphical representation of systems using this abstraction, describe a Javascript Object Notation (JSON)[1]-based textual representation of these systems, called **DyNAML** (**Dy**namic **N**ucleic **A**cid **M**arkup **L**anguage), and finally propose a standard 'library' of useful hairpin motifs based on these conventions.

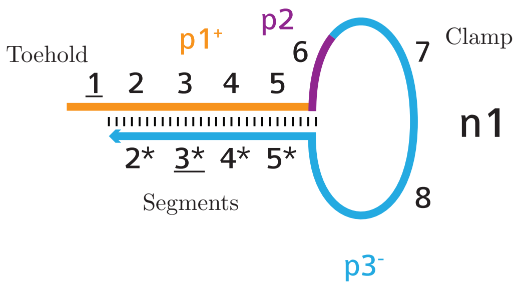

Figure 1: Segment features of a hairpin

1.1 Role of this document

This document serves as a description and specification for the graphical and textual representations of the “nodal” abstraction and **DyNAML** as described above. A reference implementation, called **DyNAMiC** (the **Dy**namic **N**ucleic **A**cid **M**echan**i**sm **C**compiler) is also provided, and the API documentation for DyNAMiC provides a detailed specification of all available classes, properties, and configuration options available to programmers.

1.2 Prior art

2 Concepts and terminology

Strand — a strand is a single contiguous sequence of nucleotides

Segment — a segment consists of a sequence of nucleotides on a strand. Segments perform different ‘roles’, which dictate the necessary length of the segment. For instance, a ‘toehold’ segment is generally shorter than segments with other roles.

Domain — a set of 1 or more segments. Domains map to ports in the nodal representation and are thus semantic entities which are assigned functional roles (see “Port” below), whereas segments are largely created for the purpose of granular control over sequence design.

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.

Pole — the pole of a strand is the first numbered segment of a strand. The pole of a domain is the toehold segment.

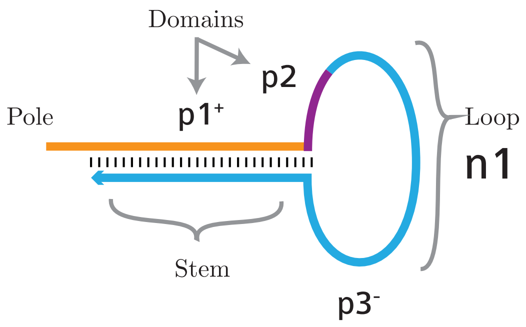

Figure 2: Domain features of a hairpin

Essential properties	**Emergent properties**
Number of segments Segment order Domains (segment grouping) Domain roles	Available inputs, outputs, and bridges
Hybridization	Shape (secondary structure) Internal logic

Figure 3: Various underlying ('essential') properties of a particular motif yield interesting ('emergent') properties at the nodal level

Polarity — polarity of a domain describes the location of that domain's toehold. A 'positive' domain has a toehold closer to the strand's 3' end, while a 'negative' domain's toehold is at the 5' end. Strand polarity refers to whether segment numbering begins at the 3' or 5' end of the strand (whether the 'pole' is at the 3' or the 5' end).

Motif — a 'class' of node. All motifs have some number of 'essential' properties which describe the underlying composition (implementation) of the motif, in terms of a single chemical species (such as a hairpin) or a set of other motifs. These 'essential' properties map to higher-level, 'emergent' properties, which describe features of the motif that are of interest to a nodal programmer. A list of 'essential' and corresponding 'emergent' properties for hairpin-based motifs is described in Figure 3. Each motif may also have a series of implementation 'parameters', which are left to be specified by the node.

Node — an instance of a motif which also encodes values for various implementation parameters. In particular, nodes specify the complementarity relationships between each domain/port of the motif and other domains/ports in the system. A node is 'fully-specified' if all parameters necessary for sequence design have been specified or calculated, and the node could be synthesized with no further processing—that is, a set of fully-specified nodes can be unambiguously converted to a set of strands. A node is 'well-specified' if sufficient parameters are provided that all parameters necessary for sequence design can be calculated. Multiple possible implementations of a given set of well-specified nodes may be possible, so a well-specified node is still said to be ambiguous.

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). 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.

System — a DyNAMiC system consists of a set of fully-specified 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 motifs and interactions. Specifically, we distinguish between the 'reaction graph' and the 'reaction mechanism,' and introduce specific conventions for depicting hairpin-based species.

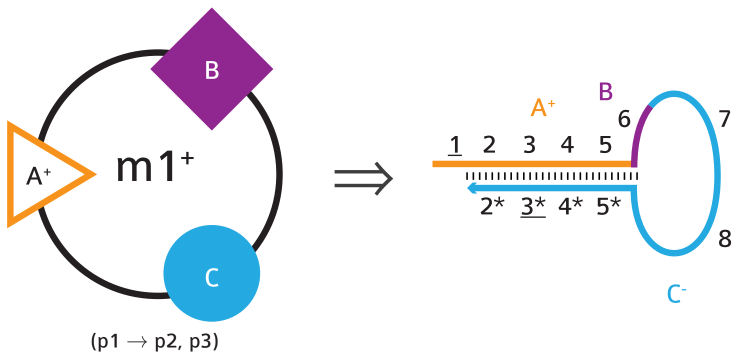

Figure 4: Explicit mapping of a graphical representation (‘depiction’) of a motif to a hairpin implementation. The graphic represents the arrangement of domains, the segment composition of those domains, and the mapping of domains onto ports. Underlined segments represent toeholds segments

3.1 Motifs

To describe fully-specified nodes in a system, one must specify an unambiguous mapping between each node in the system and an implementation in terms of strands. One way to do this would be to explicitly specify the underlying implementation of every single node in the system. However, this would scarcely constitute an abstraction of the system—we would simply be describing the system twice!

Instead, we propose that first, a set of motifs with well-specified implementations should be described. This constitutes mapping a graphical representation of a motif to a strand or set of strands. In the hairpin case, each motif is implemented with a hairpin; therefore, describing a motif requires mapping each port in the motif to a domain on the hairpin. This is done by drawing the graphical representation (or ‘depiction’) of the motif, connected by a fat arrow (⇒) to a schematic of the underlying strand representation.

Each motif receives a textual label which, by convention, is a number preceded by the letter ‘m,’ (e.g. `m1, m2, m3, ...`). Each motif has some number of ports of the various roles described above.

Each port in the motif must correspond to a domain on the strand. Ports are labeled alphabetically along the motif, and corresponding domains are labelled alphabetically along the strand. Port/domain lettering is local to the motif — that is, the first port of every motif should be ‘A.’ Ports/domains of a motif can be unambiguously referred to as `<motif 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 motif is defined, a its ports should reflect the minimum free energy secondary structure of its corresponding implementation strand 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 (see example). Bridge ports within loops are purple. Output ports within loops are blue. Output ports at a pole are green. Output ports on initiator (single) strands are brown.

When depicted, the name of each port in the motif 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 motif, 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 motif, but may be referred to as follows, to avoid ambiguity:
`<motif 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 behavior of the motif should be specified. While in general this may be deduced from the secondary structure of the implementation strand, it is good practice to denote it explicitly. 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, separated by logical operators (e.g. `&`, `|`), and `<output expression>` is a comma-separated list of output port names. The port names used can be within the scope of the motif (i.e. `A` is sufficient; `m1.A` is permissible, but not necessary) For example:

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

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

$$\mathtt{A\ \&\ B} \rightarrow \mathtt{C}, \mathtt{D}$$

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:

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

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

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

3.2 Nodes

Once the motifs in a system have been described (or referenced), the nodes of the system can be described. For each node, this consists of referencing a motif, naming ports of the motif, and specifying complementarity relationships between nodes.

This is best described with an example.

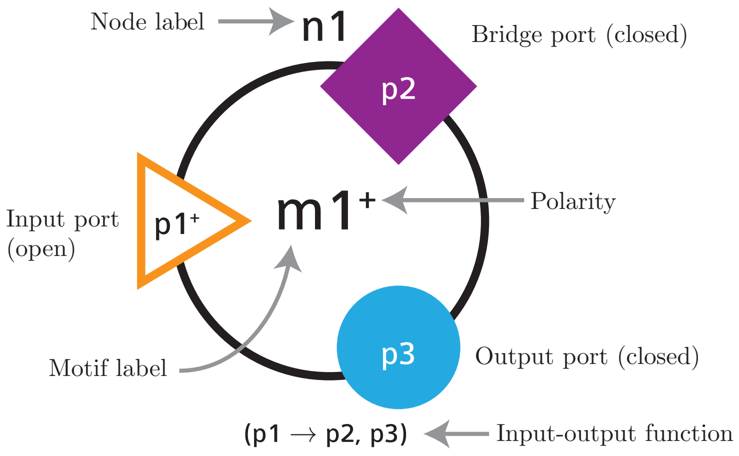

Figure 5: Features of a node.

Each node receives a textual label which is a number preceded by the letter ‘n,’ (e.g. `n1`, `n2`, `n3`, . . .). The motif name can be appended in parenthesis to the node name (e.g. `n1`(`m1`)), although this is only for clarification and not strictly necessary. 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`, `n1`(`m1`).`p1`, etc.).

When a node is depicted, the node name should appear above the node and the motif name 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. `a` $\rightarrow$ `b` becomes `p1` $\rightarrow$ `p2`). In fully-specified nodes, the polarity of each node must be indicated using superscripts, as described above. However, a well-specified node may simply specify the overall polarity of the node with a superscript following the motif name, as the individual port polarities can then be inferred.

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

3.2.1 Geometry

One of the primary attractions to the nodal abstraction is its ability to convey explicit geometries, which can be experimentally measured. Circular nodes do not convey geometric information. However, nodes drawn using other geometric shapes can have explicit geometry.

3.2.2 Reaction graph vs. Reaction mechanism

Yin et al. utilize the nodal graphical conventions to draw essentially two types of diagrams of dynamic nucleic acid systems, which convey slightly different types of information. The *reaction 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 reaction graph. The reaction mechanism can be thought of as the "execution" of the reaction graph. A *reaction enumerator* can generate a reaction mechanism from a reaction graph by determining possible reactions between different species in the ensemble.

4 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 for describe the *components* of a nucleic acid systems in declarative manner. That is, DyNAML describes the reaction graph. Specifically, DyNAML documents capture:

- Motifs
- Nodes
- Complementarities

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

- Reactions
- Complexes
- 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 Pepper.

4.1 DyNAML documents

DyNAML documents are written in Javascript Object Notation (JSON). The root object of all DyNAML documents has two properties, each containing an array of child elements (described below): `motifs` and `nodes`.

4.2 Motifs in DyNAML

Motifs in DyNAML are represented with the `motifs` array. Motifs are described by enumerating, naming, and describing the domains of the motif, then assigning `role`s 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 Figures 6, 7)

4.2.1 Complementarities and secondary structure

The strand(s) which comprise a motif must have at least one meta-stable secondary structure. A unique secondary structure reflects the hybridization of zero or more segments in the motif to other segments (either within in the motif or outside of the motif). 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 motif may have multiple meta-stable secondary structures, and therefore there may be more complementarity relationships between segments in the motif than those implied by a single secondary structure. For this reason, DyNAML allows the programmer to specify both secondary structure and complementarity relationships. Secondary structure(s) are defined on motif elements, using the `structure` property, whereas complementarities are defined separately.

Complementarities may be specified either within `segment` elements or `motif` elements. To specify complementarities within the `segment` element, the `identity` property is used — segments which should have the same sequence should have the same `identity`, whereas segments which should be complementary should have opposite `identity`s (e.g. `identity: "a"` is complementary to `identity: "a*"`); see Figure 7 for a full example. Alternatively, the complementarities can be specified in the `motif` element, using the `complements` and `equals` properties.

4.2.2 `motif` elements

`motif` objects may contain the following properties:

`name` the motif name, following above conventions

`type` the type of species (or set of species) implementing the motif. Currently, only `hairpin` is supported.

`polarity` for `type="hairpin"` motifs, the order in which segments will appear; + for $5' \to 3'$, - for $3' \to 5'$.

`structure` specifies the meta-stable secondary structure(s) of the strand(s) comprising the motif. Specifically, `structure` may be a `structure` element or an array of `structure` elements, each representing a meta-stable secondary structure of the motif(s) in question.

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

`complements` this property, along with `equals` property, allow segment equality/complementarity relationships to be specified within the `motif` element (rather than the `segment` level; see section 4.2.1). This property should contain an array of segment `name` pairs; each pair represents a set of two segments which must be complementary. It is not necessary to specify bidirectional constraints (e.g. `["1","2"]` is treated the same as `["2","1"]`)

`equals` This property should contain an array of segment `name` pairs; each pair represents a set of two segments which must have equal identities.

4.2.3 `domain` elements

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

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

`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 motif. 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 based on the secondary structure. For `type="hairpin"` motifs, 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

`polarity` for `type="hairpin"` motifs, the location of the toehold segment.

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

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

4.2.4 `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 motif)

`identifier` the identifier of the segment. Can be used to specify complementarity relationships within the motif — i.e. a segment with `identifier: "a"` will be complementary to a segment with `identifier: "a*"`, and two segments with identical identifiers 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.

4.2.5 `structure` elements

A `structure` element describes a single unique meta-stable secondary structures which may be adopted by the strands in a particular motif. Motifs may contain multiple `structure` elements, in order to specify multiple meta-stable secondary structures. 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[2] or HU+/DU+[4] 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.

Multiple secondary structures may be specified for a particular motif, as most motifs will have multiple meta-stable secondary structures. These can be introduced by passing an array of `structure` elements, to the `structure` property, rather than a single structure element. These states are used by thermodynamic sequence designers (like NUPACK), by enumerators to determine whether the motif behaves as expected, and by the user interface to calculate the possible sequestration states of the domains in the motif. The listing of possible secondary structures need not be exhaustive (for instance, intermediate secondary structures representing transient branch migration states should not be included) — rather, the list should contain all secondary structures which are intended to be long-lived resting states (meta-stable).

In some secondary structures, some of the segments may be hybridized to segments in other motif(s). For instance, an opened hairpin is likely to have segments of some domains hybridized to domains in other species. In this case, the structure specification may include an "incomplete" hybridization. For instance, in the dot-bracket notation, a single strand with six segments, hybridized entirely to another strand (or multiple other strands; this notation does not differentiate) would be represented `((((((`.

4.2.6 Mutli-stranded Motifs

Motifs containing multiple strands may be represented using the `strands` array, in place of the `domains` array. The `strands` array must contain one or more `strand` elements. `strand` elements are described in exactly the same manner as motifs, and all the same properties (with the somewhat obvious exception of the `strands` array) are allowed. Note: all named `domain` and `segment` elements remain scoped to the motif.

4.3 Nodes

The root object should have a `nodes` array, containing one or more`node` elements. Nodes are described by invoking a motif (or describing a motif inline; see Section 12), naming domains, optionally specifying complementarities, and optionally specifying values for various other design parameters. For a full example, see Figure 8.

4.3.1 `node` elements

`node` elements may have the following properties:

`name` the node name, following above conventions

`motif` the motif to be invoked to describe this node's implementation. This may optionally be omitted if the motif is described inline (see Section 12).

`domains` an array of pseudo-`domain` objects. Pseudo-`domain` objects are equivalent to normal `domain` objects, but when a node references a motif, all properties except the `name` may be omitted. In essence, a pseudo-`domain` object allows the programmer to name a domain exposed by the referenced motif.

`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.

4.3.2 Psudo-`domain` elements in nodes

In nodes, `domain` elements allow motif domains to be named in the node. If a `motif` attribute is specified on the containing `node` element, then `domain` elements may have the following attributes:

`name` the domain/port name, following above conventions

`identity` the name of the domain/port in the motif (e.g. the value of the `name` attribute in the corresponding `domain` element in the `motif`).

4.3.3 `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

4.3.4 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 Parameterization."

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 an integer number of nucleotides specifying the length of toehold segments in this node

`clampLength` accepts an integer number of nucleotides specifying the length of of clamp segments in this node

4.3.5 Cascading parameterization

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 6).

4.3.6 Inline Motifs

Normally, motifs are defined separately from nodes; this allows a motif to be re-used among several nodes. However, sometimes it may be useful to define a motif solely for use in a single node. In this case, an inline motif may be used.

To define an inline motif, a normal `node` element is simply decorated with a `domains` or `strands` property. These are specified exactly as one would specify a motif, with the exception that all names become the node-specific name, rather than the abstract motif-scoped name.

5 Standard motifs

We propose the following library of standard motifs which are likely to be useful for many applications; graphical representations of these motifs are given in Figure 13.

1. **Initiator** (2 segments).
2. **Initiator** (3 segments).
3. **One-output circle** (4 segments, blue output).
4. **Two-output circle** (9 segments, green/blue outputs).
5. **One-output circle** (6 segments, pink bridge).
6. **Three-output circle** (10 segments, green/blue output, pink bridge).
7. **Two-output circle** (7 segments, purple bridge, blue output).
8. **Two-output circle** (8 segments, green output, purple bridge).
9. **One-output circle** (6 segments, purple bridge).

6 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/motifs 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 partial 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).

6.1 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 partial 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 motif. That is, there are unintended complementary segments within a motif which may cause unintended secondary structure and hamper execution of the reaction graph.

7 Outstanding problems

Domain polarity problem our definition of domain polarity (toehold location) fails to account for domains where the toehold is in the middle (see Figure 6).

Intra-motif complementarities what does it mean to have segment (or even complementarities) within a motif?

Well-specified vs. Fully-specified is this distinction useful?

References

[1] Crockford, D. The application/json Media Type for JavaScript Object Notation (JSON) IETF, RFC 4627 (2006). Available: http://www.ietf.org/rfc/rfc4627.txt?number=4627

[2] Hofacker, I.L., Fontana, W., Stadler, P.F., Bonhoeffer, L.S., Tacker, M., Schuster, P. Fast folding and comparison of RNA secondary structures. Chemical Monthly 125 (2), 167188. (1994)

[3] Yin, P., Choi, H. M. T., Calvert, C. R. & Pierce, N. A. Programming biomolecular self-assembly pathways. Nature 451, 318-322 (2008).

[4] Zadeh, Joseph N. Algorithms for nucleic acid sequence design. Dissertation (Ph.D.), California Institute of Technology. (2010), Available: http://resolver.caltech.edu/CaltechTHESIS:05112010-205335518

[5] Zhang, D. Y., Turberfield, A. J., Yurke, B. & Winfree, E. Engineering Entropy-Driven Reactions and Networks Catalyzed by DNA. Science 318, 1121-1125 (2007).

```
{
        motifs: [{
                name:"m1",
                type:"hairpin",
                structure: ".(((..)))",
                domains:[{
                        name:"a",
                        role:"input",
                        polarity:"-",
                        segments:[
                                {role:"toehold", identifier:"a",},
                                {role:"clamp",   identifier:"b",},
                                {role:"clamp",   identifier:"c",},
                                {role:"toehold", identifier:"d",},
                        ]
                },{
                        name:"b",
                        role:"output",
                        polarity:"-",
                        segments:[
                                {role:"loop",    identifier:"e",},
                                {role:"loop",    identifier:"f",},
                                {role:"toehold", identifier:"d*",},
                                {role:"clamp",   identifier:"c*",},
                                {role:"clamp",   identifier:"b*",},
                        ]
                }]
        }]
}
```

Figure 6: This DyNAML fragment describes the hairpin motif depicted in Figure S3.1 of [3]

```
{
        motifs: [{
                name: "m1",
                type: "hairpin",
                structure: ".((((...))))",
                polarity: "+",
                domains: [{
                        name: "a",
                        role: "input",
                        polarity: "+",
                        segments: [
                                {role:"toehold", identifier:"a"}
                                {role:"clamp",   identifier:"b"}
                                {role:"clamp",   identifier:"c"}
                                {role:"clamp",   identifier:"d"}
                                {role:"",        identifier:"e"}
                        ]
                },{
                        name:"b",
                        role:"output",
                        polarity:"-",
                        segments:[
                                { role:"",        identifier:"f" },
                                { role:"",        identifier:"g" },
                                { role:"",        identifier:"h" },
                                { role:"",        identifier:"i" },
                                { role:"toehold", identifier:"e*" },
                                { role:"clamp",   identifier:"d*" },
                                { role:"clamp",   identifier:"c*" },
                                { role:"clamp",   identifier:"b*" },
                        ]
                }]
        }]
}
```

Figure 7: Another example: motif C, Figure S4.2 of [3]

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

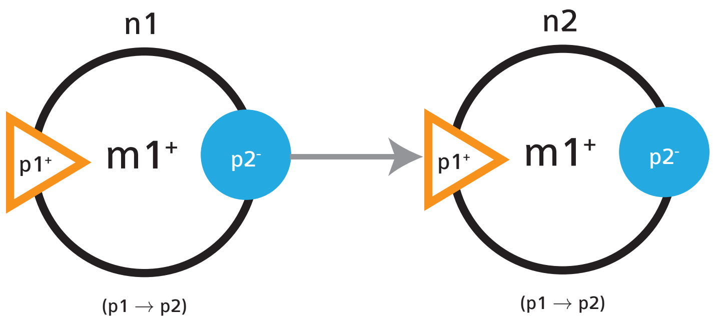

Figure 8: A simple DyNAML system with two nodes, each of standard motif shape `m1`. `n1`.p2 is complementary to `n2`.p2.

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

Figure 9: 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", identity: "A", },
                        { name: "p2", identity: "B", },
                ],
                complements: [
                        { source: "p2", node: "n2", target: "p2"}
                ],
                toeholdLength: 6,
        },{
                name: "n2",
                motif: "m1",
                domains: [
                        { name: "p1", identity: "A", },
                        { name: "p2", identity: "B", },
                ]
        }]
}
```

Figure 10: 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", identity: "A", },
                        { name: "p2", identity: "B", },
                ],
                complements: [
                        { source: "p2", node: "n2", target: "p2"}
                ],
                toeholdLength: 6,
        },{
                name: "n2",
                motif: "m1",
                domains: [
                        { name: "p1", identity: "A", },
                        { name: "p2", identity: "B", },
                ]
                toeholdLength: 8,
        }]
}
```

Figure 11: 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",
                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", },
                ]
        }]
}
```

Figure 12: Optionally, a single-use motif 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 motif by specifying a `motif` property.

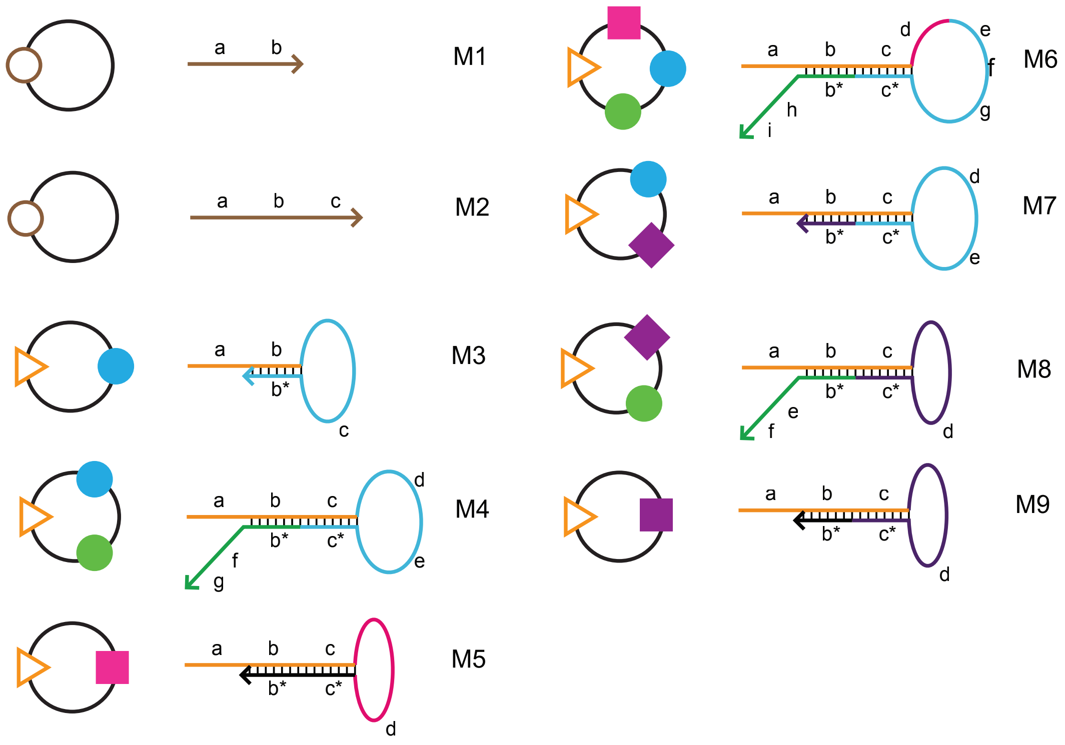

Figure 13: Standard library of motifs