
moclo Documentation

Release 0.4.5

Martin Larralde

Feb 22, 2019

Contents

1	Concepts and Definitions	3
1.1	Concepts	3
1.2	Definitions	5
1.3	Descriptive Theory	5
2	Library	15
2.1	Installation	15
2.2	Examples	16
2.3	Library Reference	21
2.4	Changelogs	29
2.5	About	36
3	Kits	37
3.1	CIDAR Kit	37
3.2	EcoFlex Kit	44
3.3	Icon Genetics Kit	48
3.4	Yeast ToolKit (YTK) / Pichia ToolKit (PTK)	55
4	Indices and tables	67
	Python Module Index	69

The MoClo system is a standard for molecular cloning that relies on the Golden Gate Assembly technique.

Concepts and Definitions

1.1 Concepts

1.1.1 Introduction

The MoClo standard was first presented in the *Weber et al., 2011*²¹³⁶⁴⁷³⁸ paper, as an attempt to standardize the process of assembling complex DNA molecules from smaller genetic elements. It is inspired by two previous standards:

- NOMAD⁸⁸⁵⁵²⁷⁸, which proposed generic notions of *modules* and *vectors*, as well as assembly using Type IIS enzymes. Modules can be combined in any order, but are clone sequentially one module at a time.
- BioBrick¹⁸⁴¹⁰⁶⁸⁸, which defines *parts* with a stable structure: assembling two parts together always gives a part with the same flanking restriction sites.

The MoClo standard enhances both of these assembly standards by relying on the Golden Gate Assembly, which allows single-step assembly of an arbitrary number of modules into a vector. Furthermore, MoClo parts are flanked by stereotypical overhangs, enforcing a particular assembly order, therefore allowing only the desired construct to be obtained.

1.1.2 Type II-S enzymes

Restriction enzymes are enzymes that are able to cut DNA at or near specific recognition sites. Among those enzymes, Type IIS enzymes cut DNA out of the sequence they recognize, at a defined distance. The cut can produce *cohesive ends*, which can then recombine with other sequences sharing the complementary cohesive ends, or *blunt ends*, which cannot recombine. The design of the cohesive ends is of great importance when using Type II-S enzymes to do molecular cloning.

²¹³⁶⁴⁷³⁸ Weber, Ernst, Carola Engler, Ramona Gruetzner, Stefan Werner, and Sylvestre Marillonnet. 'A Modular Cloning System for Standardized Assembly of Multigene Constructs'. *PLOS ONE* 6, no. 2 (18 February 2011): e16765. doi:10.1371/journal.pone.0016765

⁸⁸⁵⁵²⁷⁸ Rebatchouk, D, N Daraselia, and J O Narita. 'NOMAD: A Versatile Strategy for in Vitro DNA Manipulation Applied to Promoter Analysis and Vector Design.' *Proceedings of the National Academy of Sciences of the United States of America* 93, no. 20 (1 October 1996): 10891–96. pmid:8855278

¹⁸⁴¹⁰⁶⁸⁸ Shetty, Reshma P, Drew Endy, and Thomas F Knight. 'Engineering BioBrick Vectors from BioBrick Parts'. *Journal of Biological Engineering* 2 (14 April 2008): 5. doi:10.1186/1754-1611-2-5

1.1.3 Golden Gate Assembly

The Golden Gate Assembly relies on Type II-S enzymes to assemble several DNA sequences. The sequences are first cut by restriction enzymes, and then assembled together using a T4 DNA ligase. These two steps can be repeated in a single reaction tube using a *thermo cycler*, as the two enzymes typically do not work at the same temperature. As standard Type II-S enzymes, such as *BsaI* or *BsmBI*, create a 4-base-long cohesive end when cutting the DNA, there can be as much as 256 fragments combined together in a deterministic way in a single assembly, although *in vivo* the chemical properties of the nucleotides will most likely prevent assemblies that large to succeed.

Fig. 1: *Example GoldenGate assembly of two modules in a vector using BsaI.*

1.1.4 The MoClo system

The MoClo system combines the idea of a standard *part* format from the BioBrick standard, with the Golden Gate assembly protocol, allowing several modules to be assembled in a vector at the same time.

Hierarchy

MoClo modules and vectors are divided into several levels, describing their structural and transcriptional features:

- Level -1 modules are sequences that are not yet in a standardized backbone, but can be assembled in a dedicated vector to form a level 0 module. They are most of the time obtained via oligonucleotide synthesis, or PCR.
- Level 0 modules are standardized genetic elements: promoter, 5' UTR, signal sequence, CDS, terminator.
- Level 1 modules are transcription units, formed by a combination of Level 0 modules, and are able to express proteins
- Level 2 modules are multigenic units, containing several transcription units, and are able to express many genes at once.

Furthermore, the enzyme used during the Golden Gate Assembly depends on the assembly level. Alternating between the two enzymes makes it possible for an infinite number of genes to be inserted in the same plasmid, although biological limits are reached *in vivo*.

Types definition

Although transcription units can be assembled in any possible order in their destination vectors, level 0 modules must be assembled in a specific order to obtain a functional genetic construct. In order to enforce the assembly order, parts are flanked by fusion sites with standard sequences, which are unique to the *type* of the part. A valid level 1 module is obtained by assembling a part of each type into the destination vector.

Assembly markers

Once the Golden Gate Assembly is finished, the obtained constructs can be amplified using a bacterial host. After transformation, bacteria are selected using two different factors:

- An antibiotic for which a resistance cassette is only available on the vector, but not on any module: this allows selecting all the bacterias that received the vector plasmid
- A marker for a dropout reporter gene that can only be found in the vector but not in the final construct (such as the *gfp* or *lacZ* genes).

This double screening makes it possible to select only the bacterias that contain the expected construct, discarding the others, and retrieving the assembled plasmid through a miniprep.

1.1.5 References

1.2 Definitions

Molecular Cloning Molecular cloning is the process of assembling together fragments of DNA to obtain a more complex molecule, often presenting genetic features of interest. It describes a *process*, not a *technique*

GoldenGate GoldenGate is a molecular cloning technique that uses Type IIS restriction enzymes to cut and assemble DNA sequences into recombinant DNA molecules. It describes a *technique*

Modular Cloning A Modular Cloning system uses the GoldenGate technique to assemble several genetic *modules* of a given level into a *vector* of the same level. It can also define *types*, which are modules or vectors with specific overhangs that are collections of sequences that are fonctionnally and structurally equivalent to each other.

MoClo MoClo is originally the name of a modular cloning system published by the Marillonnet Lab which defines a set of vectors and modules to be used to assemble multigenic expression devices for plants. An extension was later provided by the same team proposing potentially infinite assemblies multigenic expression devices with the addition of two levels. Other modular cloning systems, inspired by them, were published under the name of MoClo (such as MoClo YTK, MoClo CIDAR, MoClo EcloFlex, etc.). In this work, the original toolkit is named MoClo IG, and MoClo is used as an abbreviation of modular cloning as defined above.

1.3 Descriptive Theory

This section introduces the theory that was developed to support the software implementation of the modular cloning logic. It introduces mathematical definitions of biological concepts, relying on particular on [formal language theory](#).

1.3.1 Preliminary Definitions

Genetic Alphabet

Definition

A *genetic alphabet* $\langle \Sigma, \sim \rangle$ is an algebraic structure on an alphabet Σ with a unary operation \sim verifying the following properties:

- $\sim: \Sigma^* \rightarrow \Sigma^*$ is a bijection
- $\forall x \in \Sigma^*, |\tilde{x}| = |x|$
- $\forall (x, y) \in (\Sigma^*)^2, \widetilde{x \cdot y} = \tilde{y} \cdot \tilde{x}$

Note: To stay consistent with the biology lexicon, we will be referring to a word over a genetic alphabet as a *sequence*, only explicitly naming a mathematical sequence when needed to.

Examples

- $(\{A, T, G, C\}, \sim)$ is the standard genetic alphabet, with \sim defined as $\widetilde{A \cdot G} = C \cdot T$.
 - $(\{A, T, G, C, d5SICS, dNaM\}, \sim)$ is the genetic alphabet using the unnatural base pairs from [Malyshev et al., Nature 2014](#), with \sim defined as $\widetilde{A \cdot G \cdot d5SICS} = dNaM \cdot C \cdot T$
-

Circular Sequences

Definition

A *circular word* over an alphabet Σ is a finite word with no end. It can be noted $w^{(c)}$, where w is a finite word of Σ^* .

Definition: Cardinality

Given a circular sequence $s^{(c)}$, the cardinal of $s^{(c)}$, noted $|s^{(c)}|$, is defined as:

$$|s^{(c)}| = |s|$$

Definition: Equality

Given two sequences $a^{(c)}$ and $b^{(c)}$ with

$$\begin{aligned} a &= a_0 \cdot a_1 \cdot \dots \cdot a_m \in \Sigma^{(m)}, & m \in \mathbb{N} \\ b &= b_0 \cdot b_1 \cdot \dots \cdot b_n \in \Sigma^{(n)}, & n \in \mathbb{N} \end{aligned}$$

let the $=$ relation be defined as:

$$a^{(c)} = b^{(c)} \iff \exists k \in \mathbb{N}, a = \sigma^k(b)$$

where σ is the circular shift defined as:

$$\begin{aligned} \forall u = u_1 \cdot u_2 \cdot \dots \cdot u_k \in \Sigma^k, \\ \sigma(u_1 \cdot u_2 \cdot \dots \cdot u_k) = u_k \cdot u_1 \cdot u_2 \cdot \dots \cdot u_{k-1} \end{aligned}$$

Property

$=$ is a relation of equivalence over $\Sigma^{(c)}$

Demonstration

Given the set of circular sequences $\Sigma^{(c)}$ using an alphabet Σ :

- **Reflexivity:**

$$s^{(c)} \in \Sigma^{(c)} \implies s = Id(s) = \sigma^0(s) \implies s^{(c)} = s^{(c)}$$

- **Symetry:** $\forall s_1^{(c)}, s_2^{(c)} \in \Sigma^{(c)} \times \Sigma^{(c)}$:

$$\begin{aligned} s_1^{(c)} = s_2^{(c)} &\iff \exists k \in \mathbb{N}, s_1 = \sigma^k(s_2) \\ &\iff \exists k \in \mathbb{N}, s_2 = \sigma^{-k}(s_1) \\ &\iff \exists k \in \mathbb{N}, s_2 = \sigma^{|s_1| - k}(s_1) \\ &\iff s_2^{(c)} = s_1^{(c)} \end{aligned}$$

- **Transitivity:** $\forall s_1, s_2, s_3 \in \Sigma^{(c)} \times \Sigma^{(c)} \times \Sigma^{(c)}$

$$\begin{aligned} \begin{cases} s_1^{(c)} = s_2^{(c)} \\ s_2^{(c)} = s_3^{(c)} \end{cases} &\implies \begin{cases} \exists k_1 \in \mathbb{N}, s_1 = \sigma^{k_1}(s_2) \\ \exists k_2 \in \mathbb{N}, s_2 = \sigma^{k_2}(s_3) \end{cases} \\ &\implies \exists k_1, k_2 \in \mathbb{N}^2, s_1 = \sigma^{k_1} \circ \sigma^{k_2}(s_3) \\ &\implies \exists k_1, k_2 \in \mathbb{N}^2, s_1 = \sigma^{k_1+k_2}(s_3) \\ &\implies s_1^{(c)} = s_3^{(c)} \end{aligned}$$

Definition: Automaton acceptance

Given a finite automaton A over an alphabet Σ , and $u^{(c)}$ a sequence of $\Sigma^{(c)}$, A *accepts* $u^{(c)}$ iff there exist a sequence v of Σ^* such that:

- $v^{(c)} = u^{(c)}$
 - A accepts v
-

Restriction Enzymes

Definition

Given a genetic alphabet $\langle \Sigma, \sim \rangle$, a restriction enzyme e can be defined as a tuple (S, n, k) where:

- $S \subseteq \Sigma^*$ is the finite set of *recognition sites* that e binds to
 - $\forall (s, s') \in S^2, |s| = |s'|$
 - $n \in \mathbb{Z}$ is the *cutting offset* between the last nucleotides of the site and the first nucleotide of the restriction cut
 - $k \in \mathbb{Z}$ is the *overhang length*:
 - $k = 0$ if the enzyme produces blunt cuts
 - $k > 0$ if the enzyme produces 5' overhangs
 - $k < 0$ if the enzyme produce 3' overhangs
 - $\forall (s, s') \in S^2, |s| = |s'|$
 - $n \geq -|s|, s \in S$
-

Note: This definition only covers single-cut restriction enzymes found *in vivo*, but we don't need to cover the case of double-cut restriction enzymes since they are not used in modular cloning.

Definition: Enzyme types

A restriction enzyme (S, n, k) is:

- a *blunt cutter* is $k = 0$
 - an *asymmetric cutter* if $k \neq 0$
 - a *Type IIS* enzyme if:
 - $n \geq 0$
-

$$- \forall s \in S, s \neq \bar{s}$$

Golden Gate Assembly

Definition

An *assembly* is a function of $\mathcal{P}(\Sigma^* \cup \Sigma^{(c)}) \times \mathcal{P}(E)$ to $\mathcal{P}(\Sigma^* \cup \Sigma^{(c)})$, which to a set of distinct sequences $\{d_1, \dots, d_m\}$ and a set of restriction enzymes $\{e_1, \dots, e_n\}$ associates the set of digested/ligated sequences $A = \{a_1, \dots, a_k\}$.

The notation for an assembly is:

$$d_1 + \dots + d_m \xrightarrow{e_1, \dots, e_n} a_1 + \dots + a_k$$

1.3.2 Standard Modular Cloning System

System Definition

Definition

Given a genetic alphabet $\langle \Sigma, \sim \rangle$, a Modular Cloning System S is defined as a mathematical sequence

$$(M_l, V_l, e_l)_{l \geq -1}$$

where:

- $M_l \subseteq \Sigma^* \cup \Sigma^{(c)}$ is the set of modules of level l
 - $V_l \subseteq \Sigma^{(c)}$ is the set of vectors of level l
 - $e_l \subseteq E$ is the finite, non-empty set of *asymmetric, Type IIS* restriction enzymes of level l
-

Definition: k -cyclicity

A Modular Cloning System $(M_l, V_l, e_l)_{l \geq -1}$ is said to be k -cyclic after a level λ if:

$$\begin{aligned} & \exists k \in \mathbb{N}^*, \\ & \forall l \geq \lambda, \\ & \left\{ \begin{array}{l} M_{l+k} \subseteq M_l \\ V_{l+k} \subseteq V_l \\ e_{l+k} \subseteq e_l \end{array} \right. \end{aligned}$$

Definition: λ -limit

A Modular Cloning System $(M_l, V_l, e_l)_{l \geq -1}$ is said to be λ -limited if:

$$\forall l \geq \lambda, M_l = \emptyset, V_l = \emptyset, e_l = \emptyset$$

Modules

Definition

For a given level l , M_l is defined as the set of modules $m \in \Sigma^* \cup \Sigma^{(c)}$ for which:

$$\begin{aligned} & \exists!(S, n, k) \in e_l, \\ & \exists!(S', n', k') \in e_l, \\ & \exists!(s, s') \in S \times S', \\ & \exists!(x, y, o_5, o_3) \in (\Sigma^*)^4, \\ & \exists!t \in \Sigma^*, \begin{cases} \exists!b \in \Sigma^*, & m = (s \cdot x \cdot o_5 \cdot t \cdot o_3 \cdot y \cdot \widetilde{s'} \cdot b)^{(c)}, & \text{if } m \in \Sigma^{(c)} \\ \exists!u, v \in (\Sigma^*)^2, & m = u \cdot s \cdot x \cdot o_5 \cdot t \cdot o_3 \cdot y \cdot \widetilde{s'} \cdot v, & \text{if } m \notin \Sigma^{(c)} \end{cases} \end{aligned}$$

with:

- $|x| = n$
- $|y| = n'$
- $|o_5| = \text{abs}(k)$
- $|o_3| = \text{abs}(k')$

Note: This decomposition is called the *canonic module decomposition*, where:

- t is the *target sequence* of the module m
- b is the *backbone* of the module m (if m is circular)
- u and v are called the *prefix* and *suffix* of the module m (if m is not circular)
- o_5 and o_3 are the *upstream* and *downstream overhangs* respectively.

Property

$\forall \langle \Sigma, \sim \rangle, \forall l \geq -1, \forall e_l \subset E$:

M_l is a rational language

Demonstration

Let there be a genetic alphabet $\langle \Sigma, \sim \rangle$ and a Modular Cloning System $(M_l, V_l, e_l)_{l \geq -1}$ over it.

$\forall l \geq -1$, the regular expression:

$$\bigcup_{\substack{(S, n, k) \in e_l \\ (S', n', k') \in e_l}} \Sigma^* \cdot S \cdot \Sigma^n \cdot \Sigma^{\text{abs}(k)} \cdot \Sigma^* \cdot \overline{(S|S')} \cdot \Sigma^* \cdot \Sigma^{\text{abs}(k')} \cdot \Sigma^{n'} \cdot \widetilde{S'} \cdot \Sigma^*$$

where:

- \star is the **Kleene star**.
- $\widetilde{S} = \{\widetilde{s}, s \in S\}$ (*reverse complementation operator*).
- $\overline{S} = \{w \in \Sigma^*, w \notin S\}$ (*complement operator*).

- $S|S' = S \cup S'$ (alternation operator).

matches a sequence $m \in \Sigma^* \cup \Sigma^{(c)}$ if and only if $m \in M_l$.

M_l is regular, so given Kleene's Theorem, M_l is rational.

Vectors

Definition

For a given level l , V_l is defined as the set of vectors $v \in \Sigma^{(c)}$ for which:

$$\begin{aligned} \exists!(S, n, k) &\in e_l, \\ \exists!(S', n', k') &\in e_l, \\ \exists!(s, s') &\in S \times S', \\ \exists!(x, y, o_5, o_3) &\in (\Sigma^*)^4, \\ \exists!(b, p) &\in (\Sigma^*)^2, \exists!b \in \Sigma^*, v = (o_3 \cdot b \cdot o_5 \cdot y \cdot \tilde{s} \cdot p \cdot st \cdot x)^{(c)} \end{aligned}$$

with:

- $|x| = n$
 - $|y| = n'$
 - $|o_5| = \text{abs}(k)$
 - $|o_3| = \text{abs}(k')$
 - $o_3 \neq o_5$
-

Note: This decomposition is called the *canonic vector decomposition*, where:

- p is the *placeholder sequence* of the vector v
 - b is the *backbone* of the vector v
 - o_3 and o_5 are the *upstream* and *downstream overhangs* respectively.
-

Overhangs

By definition, every valid level l module and vector only have a single canonic decomposition where they have unique o_5 and o_3 overhangs. As such, let the function *up* (resp. *down*) be defined as the function which:

- to a module m associates the word o_5 (resp. o_3) from its canonic module decomposition
- to a vector v associates the word o_3 (resp. o_5) from its canonic vector decomposition.

Standard Assembly

Definition: *Standard MoClo Assembly*

Given an assembly of level l , where $m_1, \dots, m_k \in M_l^k, v \in V_l$:

$$a : m_1 + \dots + m_k \xrightarrow{e_l} A \subset (\Sigma^* \cup \Sigma^{(c)})$$

and the partial order le over $S = \{m_1, \dots, m_k\}$ defined as:

$$\forall x, y \in S^2, \\ x \leq y \iff \begin{cases} x = y & \\ \text{down}(x) = \text{up}(y) & \text{if } x \neq y \\ \exists z \in S \setminus \{x, y\}, \text{down}(x) = \text{up}(z), z \leq y & \text{if } x \neq y \text{ and } \text{down}(x) \neq \text{up}(y) \end{cases}$$

then a chain $\langle S', \leq \rangle \subset \langle S, \leq \rangle$ is an *insert* if:

$$\begin{cases} v \leq \min(S') \\ \max(S') \leq v \end{cases} \iff \begin{cases} \text{down}(v) = \text{up}(\min(S')) \\ \text{up}(v) = \text{down}(\max(S')) \end{cases}$$

a is:

- *invalid* if $\langle S, \leq \rangle$ is an antichain or $\langle S, \geq \rangle$ has no insert.
- *valid* if $\langle S, \leq \rangle$ has at least one insert.
- *ambiguous* if $\langle S, \leq \rangle$ has more than one insert.
- *unambiguous* if $\langle S, \leq \rangle$ has exactly one insert.
- *complete* if $\langle S, \leq \rangle$ is an insert.

Corollary

If an assembly a is complete, then there exist a permutation π of $\llbracket 1, k \rrbracket$ such that:

$$m_{\pi(1)} \leq m_{\pi(2)} \leq \dots \leq m_{\pi(k-1)} \leq m_{\pi(k)}$$

and:

$$\begin{aligned} \text{up}(m_{\pi(1)}) &= \text{down}(v) \\ \text{down}(m_{\pi(k)}) &= \text{up}(v) \end{aligned}$$

Property: Uniqueness of the cohesive ends

If an assembly

$$m_1 + \dots + m_k \xrightarrow{e_l} A \subset (\Sigma^* \cup \Sigma^{(c)})$$

is unambiguous and complete, then $\forall i \in \llbracket 1, k \rrbracket$,

$$\begin{cases} \text{up}(m_i) & \neq \text{down}(m_i) \\ \text{up}(m_i) & \neq \text{up}(m_j), & j \in \llbracket 1, k \rrbracket \setminus \{i\} \\ \text{down}(m_i) & \neq \text{down}(m_j), & j \in \llbracket 1, k \rrbracket \setminus \{i\} \end{cases}$$

Demonstration

Let there be an unambiguous complete assembly

$$a : m_1 + \dots + m_k \xrightarrow{e_l} A$$

- $up(m_i) \neq down(m_i)$

Let's suppose that $\exists i \in \llbracket 1, k \rrbracket$ such that

$$up(m_i) = down(m_i)$$

then $\langle \{m_1, \dots, m_k\} \setminus \{m_i\}, \leq \rangle$ is also an insert, which cannot be since a is complete.

- $up(m_i) \neq up(m_j)$

Let's suppose that $\exists (i, j) \in \llbracket 1, k \rrbracket^2$ such that

$$up(m_i) = up(m_j)$$

Since the a is complete, there exists p_i such that

$$m_{\pi(1)} \leq m_{\pi(2)} \leq \dots \leq m_{\pi(k-1)} \leq m_{\pi(k)}$$

and since a is unambiguous, $\langle \{m_1, \dots, m_k\}, \leq \rangle$ is the only insert.

- $down(m_i) \neq down(m_j)$

TODO

Property: *Uniqueness of the assembled plasmid*

If an assembly

$$m_1 + \dots + m_k \xrightarrow{e_l} A \subset (\Sigma^* \cup \Sigma^{(c)})$$

is unambiguous, then

$$A \cap \Sigma^{(c)} = \{p\}$$

with

$$p = (up(v) \cdot b \cdot up(m_{\pi(1)}) \cdot t_{\pi(1)} \cdot \dots \cdot up(m_{\pi(n)}) \cdot t_{\pi(n)})^{(c)}$$

($n \leq k$, $n = k$ if a is complete).

Demonstration

TODO

1.3.3 Typed Modular Cloning System

System Definition

Definition

Given a genetic alphabet $\langle \Sigma, \sim \rangle$, a Typed Modular Cloning System S is defined as a mathematical sequence

$$(M_l, V_l, \mathcal{M}_l, \mathcal{V}_l, e_l)_{l \geq -1}$$

where:

- $(M_l, V_l, e_l)_{l \geq -1}$ is a standard Modular Cloning System
- $\mathcal{M}_l \subseteq \mathcal{P}(M_l) \rightarrow \mathcal{P}(M_l)$ is the set of *module types* of level l
- $\mathcal{V}_l \subseteq \mathcal{P}(V_l) \rightarrow \mathcal{P}(V_l)$ is the set of *vector types* of level l

Types

Definition

$\forall l \geq -1$, we define types using their signatures (*i.e.* the sets of upstream and downstream overhangs of elements using this type):

$$\begin{aligned} \forall t \in \mathcal{M}_l, \quad & \begin{cases} Up(t) &= \bigcup_{m \in t(M_l)} \{up(m)\} \\ Down(t) &= \bigcup_{m \in t(M_l)} \{down(m)\} \end{cases} \\ \forall t \in \mathcal{V}_l, \quad & \begin{cases} Up(t) &= \bigcup_{v \in t(V_l)} \{up(v)\} \\ Down(t) &= \bigcup_{v \in t(V_l)} \{down(v)\} \end{cases} \end{aligned}$$

Corollary

$\forall l \geq -1$,

$$\begin{aligned} \forall t \in \mathcal{M}_l, \quad t(M_l) &= \{m \in M_l \mid up(m) \in Up(t), down(m) \in Down(t)\} \\ \forall t \in \mathcal{V}_l, \quad t(V_l) &= \{v \in V_l \mid up(v) \in Up(t), down(v) \in Down(t)\} \end{aligned}$$

Property: *Structural equivalence of module types*

Given a valid (*resp.* unambiguous) (*resp.* complete) assembly

$$m_1 + \dots + m_k + v \xrightarrow{e_l} A \subset (\Sigma^* \cup \Sigma^{(c)})$$

then if there exist $t \in \mathcal{M}_l$ such that

$$\begin{cases} |Up(t)| = |Down(t)| = 1 \\ m_1 \in t(M_l) \end{cases}$$

then $\forall m_1' \in t(M_l)$,

$$m_1' + \dots + m_k + v \xrightarrow{e_l} A \subset (\Sigma^* \cup \Sigma^{(c)})$$

is valid (*resp.* unambiguous) (*resp.* complete).

2.1 Installation

The `moclo` module is designed to be modular, and as such, you only need to install whatever functionalities you are willing to use. Packages are distributed on PyPI, and it is advised to use `pip` to install them. See the [pip documentation](#) to get `pip` if it is not installed on your system.

Commands below use `pip` in user mode: the packages will be installed in a user-dependent location, and no additional permissions are needed. If for some reason you need a system-wide setup, remove the `--user` flag. Installing in user-mode should be preferred to avoid dependency issues, in particular when on an OS which provides a package manager (such as `aptitude` on Debian, or even `homebrew` on Mac OSX).

2.1.1 PyPI + pip

To download the latest release from the Python Package Index:

```
$ pip install --user moclo moclo-ytk moclo-cidar
```

2.1.2 GitHub + pip

To download the development version from the source repository, you can specify a subfolder in the installation command and directly install it:

```
$ pip install --user git+https://github.com/althonos/moclo#subdirectory=moclo
$ pip install --user git+https://github.com/althonos/moclo#subdirectory=moclo-ytk
$ pip install --user git+https://github.com/althonos/moclo#subdirectory=moclo-cidar
```

Check the CI build is passing, or else you may be installing a broken version of the library !

2.2 Examples



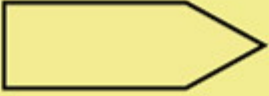


This page contains examples in Python code, generated from Jupyter notebooks with [nbsphinx](#).

2.2.1 YTK integration vector

In this example, we will be using the `moclo` library as well as the `moclo-ytk` extension kit to generate the pre-assembled YTK integration vector (**pYTK096**) from the available YTK parts, as described in the **Lee et al.** [paper](#)

Structure

The list of parts, as well as the vector structure, can be found in the **Supporting Table S1** from the *Lee et al.* supplementary materials:

Assembly Connector	Promoter	Coding Sequence	
			
1	2	3	
ConLS	<i>pTDH3</i>	mTurquoise2	
ConL1	<i>pCCW12</i>	Venus	
ConL2	<i>pPGK1</i>	mRuby2	
ConL3	<i>pHHF2</i>	I-SceI (ORF)	
ConL4	<i>pTEF1</i>	Cas9	
ConL5	<i>pTEF2</i>		
ConLS'	<i>pHHF1</i>		
	<i>pHTB2</i>		
	<i>pRPL18B</i>		
	<i>pALD6</i>		
	<i>pPAB1</i>		
	<i>pRET2</i>		
	<i>pRNR1</i>		
	<i>pSAC6</i>		
	<i>pRNR2</i>		
	<i>pPOP6</i>		
		N-terminal CDS	CDS
			
		3a	3b
		mTurquoise2	¹⁷ mTurquoise

Loading parts

We'll be loading each of the desired parts from the `moclo-ytk` registry. It is generated from the GenBank distributed with the YTK kits. They can be found on the [AddGene YTK page](#).

```
[2]: from moclo.registry.ytk import YTKRegistry
registry = YTKRegistry()

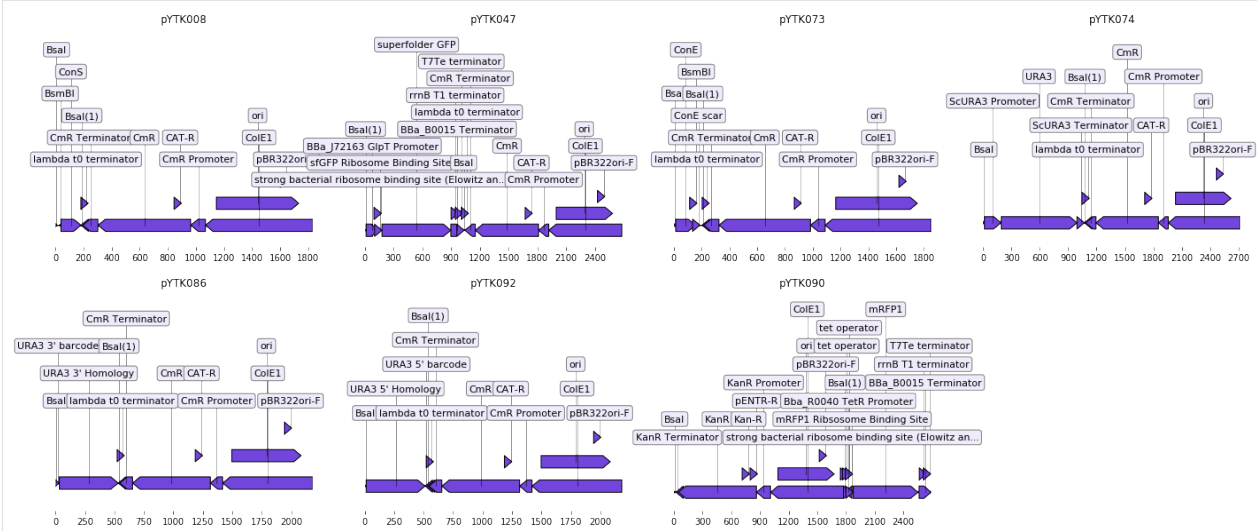
vector = registry['pYTK090'].entity           # Part 8a
modules = [registry['pYTK008'].entity,        # Part 1
            registry['pYTK047'].entity,        # Part 234r
            registry['pYTK073'].entity,        # Part 5
            registry['pYTK074'].entity,        # Part 6
            registry['pYTK086'].entity,        # Part 7
            registry['pYTK092'].entity]        # Part 8b
```

Checking parts

We can use `dna_features_viewer` to visualize your records before proceeding (for readability purposes, we'll show the records as linear although they are plasmids):

```
[3]: import itertools
import dna_features_viewer as dfv
import matplotlib.pyplot as plt

translator = dfv.BiopythonTranslator([lambda f: f.type != 'source'])
plt.figure(1, figsize=(24, 10))
for index, entity in enumerate(itertools.chain(modules, [vector])):
    ax = plt.subplot(2, 4, index + 1)
    translator.translate_record(entity.record).plot(ax)
    plt.title(entity.record.id)
plt.show()
```



Creating the assembly

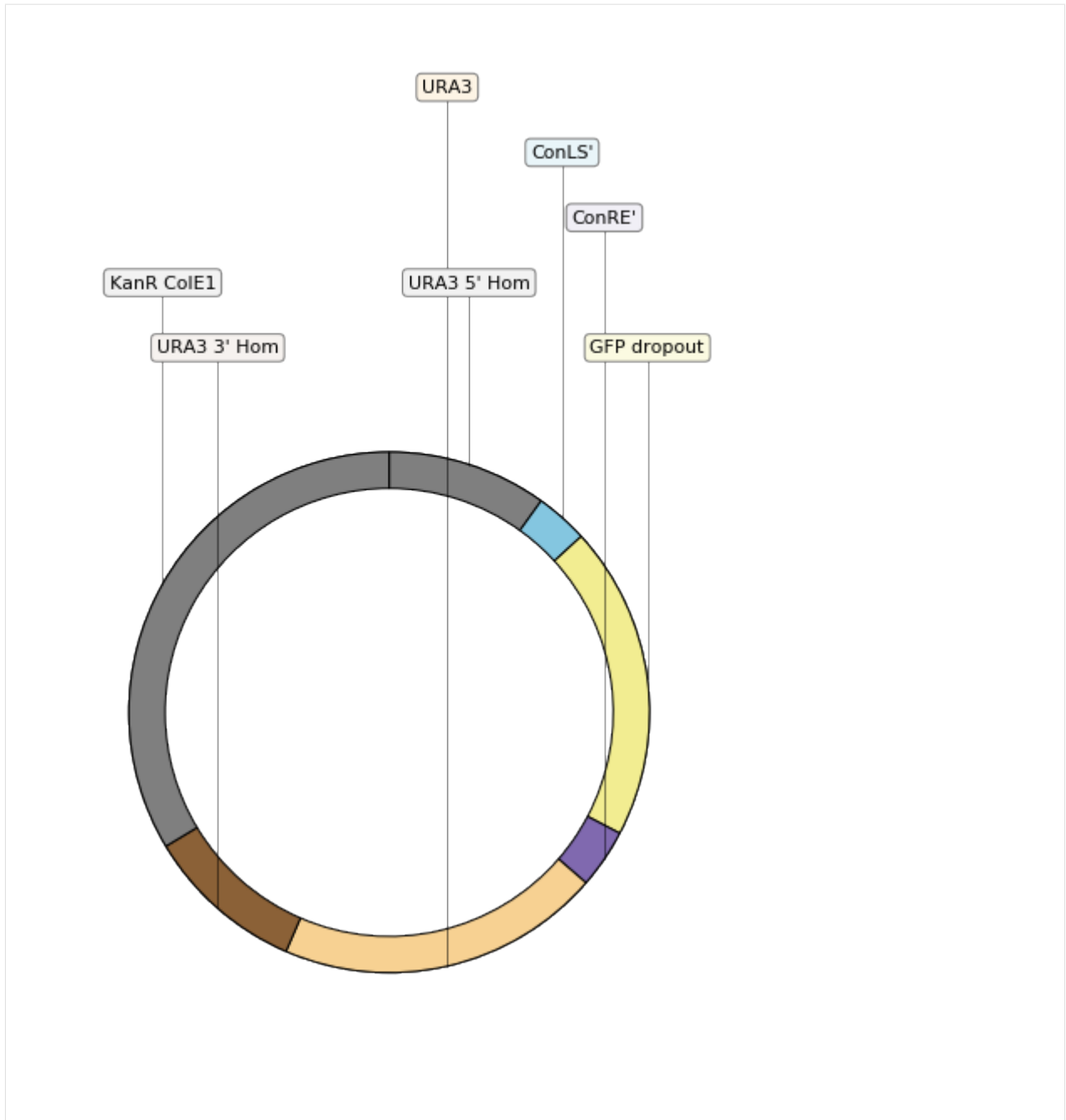
We use the Part 8a as our base assembly vector, and then assemble all the other parts into that vector:

```
[4]: assembly = vector.assemble(*modules)
```

Rendering the assembly sequence map

When creating an assembly, corresponding regions of the obtained sequence will be annotated with the ID of the sequence they come from.

```
[6]: vec_translator = IntegrationVectorTranslator([lambda f: f.type == 'source'])
vec_translator.translate_record(assembly, dfv.CircularGraphicRecord).plot(figure_
↪width=8)
plt.show()
```



Comparing the assembly to the expected vector

Hopefully the obtained assembly should look like the pYTK096 plasmid, distributed with the official YTK parts:

```
[7]: plt.figure(3, figsize=(24, 10))

ax = plt.subplot(2, 1, 1)
translator.translate_record(assembly).plot(ax)
plt.title('Assembly')
```

(continues on next page)

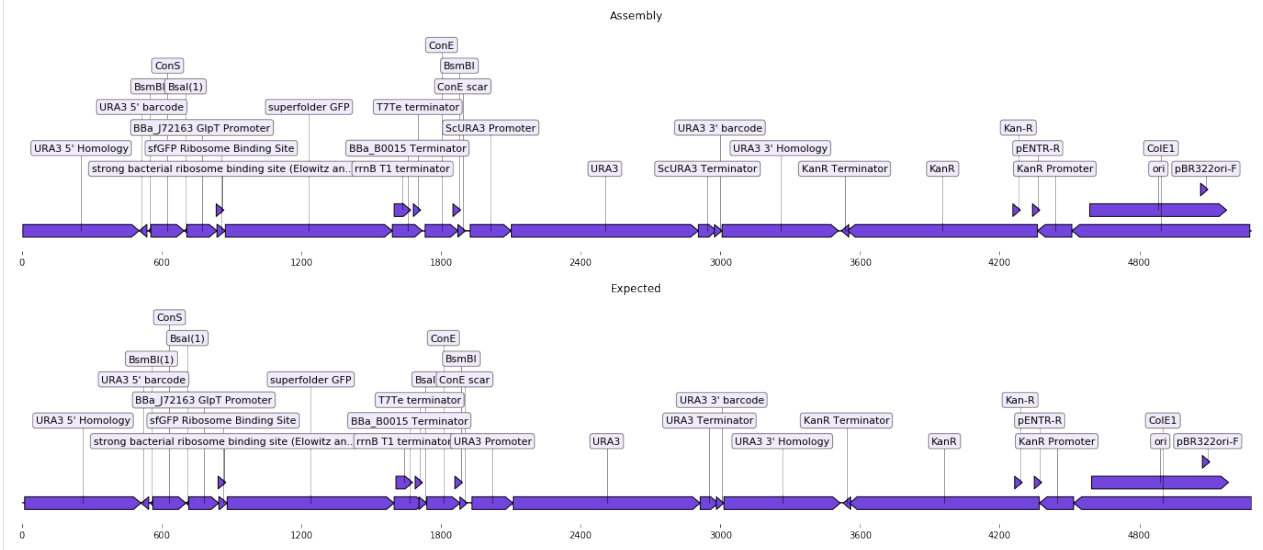
(continued from previous page)

```

ax = plt.subplot(2, 1, 2)
translator.translate_record(registry['pYTK096'].entity.record).plot(ax)
plt.title('Expected')

plt.show()

```



[]:

2.3 Library Reference

2.3.1 Record

class moclo.record.CircularRecord(*SeqRecord*)

A derived *SeqRecord* that contains a circular DNA sequence.

It handles the `in` operator as expected, and removes the implementation of the `+` operator since circular DNA sequence do not have an end to append more nucleotides to. In addition, it overloads the `>>` and `<<` operators to allow rotating the sequence and its annotations, effectively changing the `0` position.

See also:

`Bio.SeqRecord.SeqRecord` documentation on the [Biopython wiki](#).

__add__ (*other*)

Add another sequence or string to this sequence.

Since adding an arbitrary sequence to a plasmid is ambiguous (there is no sequence end), trying to add a sequence to a *CircularRecord* will raise a *TypeError*.

__contains__ (*char*)

Implement the `in` keyword, searches the sequence.

__getitem__ (*index*)

Return a sub-sequence or an individual letter.

The sub-sequence is always returned as a *SeqRecord*, since it is probably not circular anymore.

__init__ (*seq*, *id*='<unknown id>', *name*='<unknown name>', *description*='<unknown description>', *dbxrefs*=None, *features*=None, *annotations*=None, *letter_annotations*=None)
Create a new *CircularRecord* instance.

If given a *SeqRecord* as the first argument, it will simply copy all attributes of the record. This allows using *Bio.SeqIO.read* to open records, then loading them into a *CircularRecord*.

__lshift__ (*index*)
Rotate the sequence counter-clockwise, preserving annotations.

__radd__ (*other*)
Add another sequence or string to this sequence (from the left).

Since adding an arbitrary sequence to a plasmid is ambiguous (there is no sequence end), trying to add a sequence to a *CircularRecord* will raise a *TypeError*.

__rshift__ (*index*)
Rotate the sequence clockwise, preserving annotations.

reverse_complement (*id*=False, *name*=False, *description*=False, *features*=True, *annotations*=False, *letter_annotations*=True, *dbxrefs*=False)
Return a new *CircularRecord* with reverse complement sequence.

2.3.2 Registry

Base class

class *moclo.registry.base.AbstractRegistry*
An abstract registry holding MoClo plasmids.

Implementations

class *moclo.registry.base.CombinedRegistry*
A registry combining several registries into a single collection.

__init__ ()
Initialize self. See *help(type(self))* for accurate signature.

class *moclo.registry.base.EmbeddedRegistry*
An embedded registry, distributed with the library source code.

Records are stored within a BZ2 compressed JSON file, using standard annotations to allow retrieving features easily.

2.3.3 Modules

Moclo module classes.

A module is a sequence of DNA that contains a sequence of interest, such as a promoter, a CDS, a protein binding site, etc., organised in a way it can be combined to other modules to create an assembly. This involves flanking that target sequence with Type IIS restriction sites, which depend on the level of the module, as well as the chosen MoClo protocol.

Abstract

class `moclo.core.modules.AbstractModule` (*object*)

An abstract modular cloning module.

cutter

the enzyme used to cut the target sequence from the backbone plasmid during Golden Gate assembly.

Type `RestrictionType`

__init__ (*record*)

Initialize self. See `help(type(self))` for accurate signature.

is_valid ()

Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end ()

Get the downstream overhang of the target sequence.

Returns the downstream overhang.

Return type `Seq`

overhang_start ()

Get the upstream overhang of the target sequence.

Returns the downstream overhang.

Return type `Seq`

classmethod **structure** ()

Get the module structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The module target sequence
3. The downstream (3') overhang sequence

target_sequence ()

Get the target sequence of the module.

Modules are often stored in a standardized way, and contain more than the sequence of interest: for instance they can contain an antibiotic marker, that will not be part of the assembly when that module is assembled into a vector; only the target sequence is inserted.

Returns the target sequence with annotations.

Return type `SeqRecord`

Note: Depending on the cutting direction of the restriction enzyme used during assembly, the overhang will be left at the beginning or at the end, so the obtained record is exactly the sequence the enzyme created during restriction.

Level -1

class `moclo.core.modules.Product` (*AbstractModule*)

A level -1 module, often obtained as a PCR product.

Modules of this level are the lowest components of the MoClo system, but are not practical to work with until they are assembled in a standard vector to obtain *entries*.

Level 0

class `moclo.core.modules.Entry` (*AbstractModule*)

A level 0 module, often obtained from the official toolkits plasmids.

Entries are assembled from products into a standard vector suitable for selection and storage.

Level 1

class `moclo.core.modules.Cassette` (*AbstractModule*)

A level 1 module, also referred as a Transcriptional Unit.

Cassettes can either express genes in their target organism, or be assembled into *multigene* modules for expressing many genes at once, depending on the chosen cassette vector during level 0 assembly.

Level 2

class `moclo.core.modules.Device` (*AbstractModule*)

A level 2 module, also referred as a Multigene plasmid.

Modules of this level are assembled from several transcriptional units so that they contain several genes that can be expressed all at once. Most of the MoClo implementations are designed so that multiple devices can be assembled into a module that is also a valid level 1 module, as does the **Golden Braid** system with its α and Ω plasmids.

2.3.4 Vectors

MoClo vector classes.

A vector is a plasmidic DNA sequence that can hold a combination of modules of the same level to create a single module of the following level. Vectors contain a placeholder sequence that is replaced by the concatenation of the modules during the Golden Gate assembly.

Abstract

class `moclo.core.vectors.AbstractVector` (*object*)

An abstract modular cloning vector.

assemble (*module*, **modules*, ***kwargs*)

Assemble the provided modules into the vector.

Parameters

- **module** (*AbstractModule*) – a module to insert in the vector.

- **modules** (*AbstractModule*, optional) – additional modules to insert in the vector. The order of the parameters is not important, since modules will be sorted by their start overhang in the function.

Returns the assembled sequence with sequence annotations inherited from the vector and the modules.

Return type *SeqRecord*

Raises

- *DuplicateModules* – when two different modules share the same start overhang, leading in possibly non-deterministic constructs.
- *MissingModule* – when a module has an end overhang that is not shared by any other module, leading to a partial construct only
- *InvalidSequence* – when one of the modules does not match the required module structure (missing site, wrong overhang, etc.).
- *UnusedModules* – when some modules were not used during the assembly (mostly caused by duplicate parts).

overhang_end()

Get the downstream overhang of the vector sequence.

overhang_start()

Get the upstream overhang of the vector sequence.

placeholder_sequence()

Get the placeholder sequence in the vector.

The placeholder sequence is replaced by the concatenation of modules during the assembly. It often contains a dropout sequence, such as a GFP expression cassette that can be used to measure the progress of the assembly.

classmethod structure()

Get the vector structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The downstream (3') overhang sequence
2. The vector placeholder sequence
3. The upstream (5') overhang sequence

target_sequence()

Get the target sequence in the vector.

The target sequence if the part of the plasmid that is not discarded during the assembly (everything except the placeholder sequence).

Level -1

class `moclo.core.vectors.EntryVector` (*AbstractVector*)

Level 0 vector.

Level 0

class `moclo.core.vectors.CassetteVector` (*AbstractVector*)
 Level 1 vector.

Level 1

class `moclo.core.vectors.DeviceVector` (*AbstractVector*)
 Level 2 vector.

2.3.5 Parts

Moclo part classes.

Abstract

class `moclo.core.parts.AbstractPart` (*object*)
 An abstract modular cloning part.

Parts can be either modules or vectors, but are determined by their flanking overhangs sequences, declared in the `signature` class attribute. The part structure is derived from the part class (module of vector), signature, and restriction enzyme.

Example

```
>>> class ExamplePart (AbstractPart, Entry):
...     cutter = BsaI
...     signature = ('ATGC', 'ATTC')
...
>>> ExamplePart.structure()
'GGTCTCN(ATGC)(NN*N)(ATTC)NGAGACC'
```

__init__ (*record*)
 Initialize self. See `help(type(self))` for accurate signature.

classmethod characterize (*record*)
 Load the record in a concrete subclass of this type.

is_valid ()
 Check if the wrapped record follows the required class structure.
Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

classmethod structure ()
 Get the part structure, as a DNA regex pattern.
 The structure of most parts can be obtained automatically from the part signature and the restriction enzyme used in the Golden Gate assembly.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The vector placeholder sequence
3. The downstream (3') overhang sequence

2.3.6 Errors

Base classes

class `moclo.errors.MocloError` (*Exception*)
Base class for all MoClo-related exceptions.

class `moclo.errors.AssemblyError` (*MocloError, RuntimeError*)
Assembly-specific run-time error.

class `moclo.errors.AssemblyWarning` (*MocloError, Warning*)
Assembly-specific run-time warning.

Warnings can be turned into errors using the `warnings.catch_warnings` decorator combined to `warnings.simplefilter` with action set to "error".

Errors

class `moclo.errors.DuplicateModules` (*AssemblyError*)
Several modules share the same overhangs.

class `moclo.errors.InvalidSequence` (*MocloError, ValueError*)
Invalid sequence provided.

class `moclo.errors.IllegalSite` (*InvalidSequence*)
Sequence with illegal site provided.

class `moclo.errors.MissingModule` (*AssemblyError*)
A module is missing in the assembly.

Warnings

class `moclo.errors.UnusedModules` (*AssemblyWarning*)
Not all modules were used during assembly.

2.3.7 Record (`moclo.record`)

CircularRecord

A derived `SeqRecord` that contains a circular DNA sequence.

2.3.8 Registry (`moclo.registry.base`)

Item

A uniquely identified record in a registry.

AbstractRegistry

An abstract registry holding MoClo plasmids.

Continued on next page

Table 2 – continued from previous page

<i>CombinedRegistry</i>	A registry combining several registries into a single collection.
<i>EmbeddedRegistry</i>	An embedded registry, distributed with the library source code.

2.3.9 Modules (`moclo.core.modules`)

<i>AbstractModule</i>	An abstract modular cloning module.
<i>Entry</i>	A level 0 module, often obtained from the official toolkits plasmids.
<i>Cassette</i>	A level 1 module, also referred as a Transcriptional Unit.
<i>Device</i>	A level 2 module, also referred as a Multigene plasmid.

2.3.10 Vectors (`moclo.core.vectors`)

<i>AbstractVector</i>	An abstract modular cloning vector.
<i>EntryVector</i>	Level 0 vector.
<i>CassetteVector</i>	Level 1 vector.
<i>DeviceVector</i>	Level 2 vector.

2.3.11 Parts (`moclo.core.parts`)

<i>AbstractPart</i>	An abstract modular cloning part.
---------------------	-----------------------------------

2.3.12 Errors (`moclo.errors`)

Base classes

<i>MocloError</i>	Base class for all MoClo-related exceptions.
<i>AssemblyError</i>	Assembly-specific run-time error.
<i>AssemblyWarning</i>	Assembly-specific run-time warning.

Errors

<i>DuplicateModules</i>	Several modules share the same overhangs.
<i>InvalidSequence</i>	Invalid sequence provided.
<i>IllegalSite</i>	Sequence with illegal site provided.
<i>MissingModule</i>	A module is missing in the assembly.

Warnings

<i>UnusedModules</i>	Not all modules were used during assembly.
----------------------	--

2.4 Changelogs

2.4.1 moclo

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#) and this project adheres to [Semantic Versioning](#).

Unreleased

v0.4.5 - 2019-02-22

Fixed

- Support all `fs` versions under `3.0`.

v0.4.4 - 2019-02-11

Changed

- Add `2.3.0` to the supported `fs` versions.

v0.4.3 - 2019-01-06

Changed

- Add `2.2.0` to the supported `fs` versions.

Added

- Add `Item.record shortcut` to `Item.entity.record` in `moclo.registry`.
- Make `moclo.core` abstract classes check for illegal sites in sequence to be identified as *valid*.
- This *CHANGELOG* file.

Documented

- Fix typos.

v0.4.2 - 2018-08-16

Fixed

- Some registries not loading `CircularRecord` instances.

v0.4.1 - 2018-08-16

Changed

- Bump required `fs` version to `2.1.0`.

v0.4.0 - 2018-08-10

Added

- `AbstractPart.characterize` to load a record into a part instance.
- Option to include / exclude `ELabFTWRegistry` items using tags.

v0.3.0 - 2018-08-07

Added

- Annotate assembled vectors as *circular* in `AbstractVector.assemble`.
- *eLabFTW* registry connector in `moclo.registry.elabftw`.

Changed

- Move `Item._find_type` to public function `moclo.registry.utils.find_type`.
- Improve annotation generated in `AbstractVector.assemble`.

Fixed

- `AbstractPart` subclasses not being recognized as abstract.

v0.2.1 - 2018-07-27

Added

- `moclo.registry.utils` module with resistance identification function.
- Make `AbstractVector.assemble` add an alphabet to the generated sequence.

Documented

- Improved `README.rst` file.

v0.2.0 - 2018-07-24

Added

- Use `AbstractModule.cutter` and `AbstractVector.cutter` to deduce the required structure for modules and vectors.
- `AbstractPart` class to generate sequence structure based on part signature.
- Add registry API in `moclo.registry` module.

Changed

- Make `StructuredRecord` convert `SeqRecord` to `CircularRecord` on instantiation if needed.
- Use `target_sequence` method in `AbstractVector.assemble`.
- Make modules and vectors add sources to their target sequences when assembled.
- Patch `CircularRecord.reverse_complement` to return a `CircularRecord`.

Documented

- Add `moclo.base.parts` to documentation.
- Add example in `AbstractPart` docstring.
- Fix documentation of `moclo.base`

Fixed

- Fix `AbstractModule.target_sequence` and `AbstractVector.target_sequence` to take into account cutter overhand position.

v0.1.0 - 2018-07-12

Initial public release.

2.4.2 moclo-cidar

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#) and this project adheres to [Semantic Versioning](#).

Unreleased

Added

- This *CHANGELOG* file.

Changed

- Update CIDAR sequences to latest AddGene data update (1.6.2).

v0.4.0 - 2018-08-16

Changed

- Bumped `moclo` minimal required version to `v0.4.0`.

Documented

- Add *SVG* images illustrating CIDAR parts to the API documentation.
- Fixed class hierarchy in API documentation.

v0.3.0 - 2018-08-07

Changed

- Bumped `moclo` minimal required version to `v0.3.0`.

Removed

- Location attribute handler from `CIDARRegistry`.
- *DVA* and *DVK* sequences from the registry as they are not MoClo elements.

v0.2.0 - 2018-07-25

Added

- Partial reference CIDAR sequences in `moclo.registry.cidar.CIDARRegistry`.

Changed

- Use signature and cutter to generate structures of `moclo.kits.cidar.CIDARPart` subclasses.
- Bumped `moclo` minimal required version to `v0.2.0`.

Documented

- Fixed link to documentation in `README.rst`.

v0.1.0 - 2018-07-12

Initial public release.

2.4.3 moclo-ecoflex

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#) and this project adheres to [Semantic Versioning](#).

Unreleased

Fixed

- Annotations of CmR cassette in pBP-BBa_B0034.
- Add missing sequences to the EcoFlex registry:
 - Promoters: pBP-SJM9** series.

v0.3.1 - 2018-11-19

Added

- This *CHANGELOG* file.

Fixed

- Wheel distribution not embedding the `moclo.registry.ecoflex` module.
- Add missing sequences to the EcoFlex registry:
 - Promoters: pBP-BBa_B0012, pBP-BBa_B0015, pBP-BBa_B0034,
 - Tags: pBP-HexHis
 - CDS: pBP-eCFP, pBP-eGFP
 - Promoter + RBS: pBP-T7-RBS-His6
 - Device Vectors: pTU2-a-RFP, pTU2-b-RFP

v0.3.0 - 2018-08-16

Changed

- Bumped `moclo` minimal required version to `v0.4.0`.

Documented

- Fixed class hierarchy in API documentation.

v0.2.0 - 2018-08-07

Added

- Partial reference EcoFlex sequences in `moclo.registry.ecoflex.EcoFlexRegistry`.

Changed

- Use signature and cutter to generate structures of `moclo.kits.ecoflex.EcoFlexPart` subclasses.
- Bumped `moclo` minimal required version to `v0.3.0`.

v0.1.0 - 2018-07-12

Initial public release.

2.4.4 moclo-gb3

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#) and this project adheres to [Semantic Versioning](#).

v0.1.0 - 2018-07-12

Initial public release.

2.4.5 moclo-ig

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#) and this project adheres to [Semantic Versioning](#).

v0.1.0 - 2018-07-12

Initial public release.

2.4.6 moclo-ytk

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#) and this project adheres to [Semantic Versioning](#).

Unreleased

Changed

- Update Pichia ToolKit sequences to latest AddGene data update (1.6.2).

Added

- This *CHANGELOG* file.

v0.4.0 - 2018-08-16

Changed

- Bumped `moclo` minimal required version to `v0.4.0`.

Documented

- Fixed class hierarchy in API documentation.

v0.3.0 - 2018-08-07

Changed

- Bumped `moclo` minimal required version to `v0.3.0`.

Documented

- Fix links to documentation in `README.rst`.
- Add YTK specific notebook in a Docker image.

v0.2.0 - 2018-07-24

Added

- Reference Yeast ToolKit sequences in `moclo.registry.ytk.YTKRegistry`.
- Reference Pichia ToolKit sequences in `moclo.registry.ytk.PTKRegistry`.

Changed

- Redefined `YTKProduct._structure` as a public static method.

v0.1.0 - 2018-07-12

Initial public release.

2.5 About

2.5.1 Authors

moclo is developed and maintained by:

Martin Larralde

Graduate student, Biology department
École Normale Supérieure Paris Saclay
martin.larralde@ens-paris-saclay.fr

This library was developed during a summer internship at **Institut Pasteur**, under the supervision of:

François Bertaux

Research Engineer, InBio Unit
Inria / Institut Pasteur
francois.bertaux@pasteur.fr

Grégory Batt

Senior Scientist, Head of InBio Unit
Inria / Institut Pasteur
gregory.batt@inria.fr

2.5.2 License

This project is licensed under the [MIT License](#).

3.1 CIDAR Kit

An implementation of the CIDAR ToolKit for the Python MoClo library.

References

1. Iverson, S. V., Haddock, T. L., Beal, J., & Densmore, D. M. (2016). CIDAR MoClo: Improved MoClo Assembly Standard and New E. coli Part Library Enable Rapid Combinatorial Design for Synthetic and Traditional Biology. *ACS Synthetic Biology*, 5(1), 99–103.
 2. Weber, E., Engler, C., Gruetzner, R., Werner, S., Marillonnet, S. (2011). A Modular Cloning System for Standardized Assembly of Multigene Constructs. *PLOS ONE*, 6(2), e16765.
-

3.1.1 Level -1

Module

```
class moclo.kits.cidar.CIDARProduct (Product)
    A CIDAR MoClo product.

    __init__ (record)
        Initialize self. See help(type(self)) for accurate signature.

    cutter
        alias of Bio.Restriction.Restriction.BbsI

    is_valid()
        Check if the wrapped record follows the required class structure.

        Returns True if the record is valid, False otherwise.

        Return type bool
```

overhang_end()

Get the downstream overhang of the target sequence.

Returns the downstream overhang.

Return type Seq

overhang_start()

Get the upstream overhang of the target sequence.

Returns the downstream overhang.

Return type Seq

classmethod structure()

Get the module structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The module target sequence
3. The downstream (3') overhang sequence

target_sequence()

Get the target sequence of the module.

Modules are often stored in a standardized way, and contain more than the sequence of interest: for instance they can contain an antibiotic marker, that will not be part of the assembly when that module is assembled into a vector; only the target sequence is inserted.

Returns the target sequence with annotations.

Return type SeqRecord

Note: Depending on the cutting direction of the restriction enzyme used during assembly, the overhang will be left at the beginning or at the end, so the obtained record is exactly the sequence the enzyme created during restriction.

Vector

class moclo.kits.cidar.CIDAREntryVector(*EntryVector*)

A CIDAR MoClo entry vector.

__init__(*record*)

Initialize self. See help(type(self)) for accurate signature.

assemble(*module*, **modules*, ***kwargs*)

Assemble the provided modules into the vector.

Parameters

- **module** (*AbstractModule*) – a module to insert in the vector.
- **modules** (*AbstractModule*, optional) – additional modules to insert in the vector. The order of the parameters is not important, since modules will be sorted by their start overhang in the function.

Returns the assembled sequence with sequence annotations inherited from the vector and the modules.

Return type `SeqRecord`

Raises

- *DuplicateModules* – when two different modules share the same start overhang, leading in possibly non-deterministic constructs.
- *MissingModule* – when a module has an end overhang that is not shared by any other module, leading to a partial construct only
- *InvalidSequence* – when one of the modules does not match the required module structure (missing site, wrong overhang, etc.).
- *UnusedModules* – when some modules were not used during the assembly (mostly caused by duplicate parts).

cutter

alias of `Bio.Restriction.Restriction.BbsI`

is_valid()

Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end()

Get the downstream overhang of the vector sequence.

overhang_start()

Get the upstream overhang of the vector sequence.

placeholder_sequence()

Get the placeholder sequence in the vector.

The placeholder sequence is replaced by the concatenation of modules during the assembly. It often contains a dropout sequence, such as a GFP expression cassette that can be used to measure the progress of the assembly.

static structure()

Get the vector structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The downstream (3') overhang sequence
2. The vector placeholder sequence
3. The upstream (5') overhang sequence

target_sequence()

Get the target sequence in the vector.

The target sequence if the part of the plasmid that is not discarded during the assembly (everything except the placeholder sequence).

3.1.2 Level 0

Module

class `moclo.kits.cidar.CIDAREntry` (*Entry*)

A CIDAR MoClo entry.

__init__ (*record*)

Initialize self. See `help(type(self))` for accurate signature.

cutter

alias of `Bio.Restriction.Restriction.BsaI`

is_valid ()

Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end ()

Get the downstream overhang of the target sequence.

Returns the downstream overhang.

Return type `Seq`

overhang_start ()

Get the upstream overhang of the target sequence.

Returns the downstream overhang.

Return type `Seq`

classmethod structure ()

Get the module structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The module target sequence
3. The downstream (3') overhang sequence

target_sequence ()

Get the target sequence of the module.

Modules are often stored in a standardized way, and contain more than the sequence of interest: for instance they can contain an antibiotic marker, that will not be part of the assembly when that module is assembled into a vector; only the target sequence is inserted.

Returns the target sequence with annotations.

Return type `SeqRecord`

Note: Depending on the cutting direction of the restriction enzyme used during assembly, the overhang will be left at the beginning or at the end, so the obtained record is exactly the sequence the enzyme created during restriction.

Vector

class `moclo.kits.cidar.CIDARCassetteVector` (*CassetteVector*)
 A CIDAR Moclo cassette vector.

References

Iverson et al., Figure 1.

__init__ (*record*)
 Initialize self. See `help(type(self))` for accurate signature.

assemble (*module*, **modules*, ***kwargs*)
 Assemble the provided modules into the vector.

Parameters

- **module** (*AbstractModule*) – a module to insert in the vector.
- **modules** (*AbstractModule*, optional) – additional modules to insert in the vector. The order of the parameters is not important, since modules will be sorted by their start overhang in the function.

Returns the assembled sequence with sequence annotations inherited from the vector and the modules.

Return type `SeqRecord`

Raises

- *DuplicateModules* – when two different modules share the same start overhang, leading in possibly non-deterministic constructs.
- *MissingModule* – when a module has an end overhang that is not shared by any other module, leading to a partial construct only
- *InvalidSequence* – when one of the modules does not match the required module structure (missing site, wrong overhang, etc.).
- *UnusedModules* – when some modules were not used during the assembly (mostly caused by duplicate parts).

cutter
 alias of `Bio.Restriction.Restriction.BsaI`

is_valid ()
 Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end ()
 Get the downstream overhang of the vector sequence.

overhang_start ()
 Get the upstream overhang of the vector sequence.

placeholder_sequence ()
 Get the placeholder sequence in the vector.

The placeholder sequence is replaced by the concatenation of modules during the assembly. It often contains a dropout sequence, such as a GFP expression cassette that can be used to measure the progress of the assembly.

static structure()

Get the vector structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The downstream (3') overhang sequence
2. The vector placeholder sequence
3. The upstream (5') overhang sequence

target_sequence()

Get the target sequence in the vector.

The target sequence is the part of the plasmid that is not discarded during the assembly (everything except the placeholder sequence).

Parts

class `moclo.kits.cidar.CIDARPromoter` (*CIDARPart*, *CIDAREntry*)

A CIDAR Promoter part.

Parts of this type contain a promoter. The upstream overhangs can be changed to amend the order of assembly of a circuit from different cassettes.

Note: The CIDAR toolkit parts provide 4 different upstream overhangs: *GGAG*, *GCTT*, *CGCT*, and *TGCC*. These are not enforced in this module, and any upstream sequence will be accepted. The downstream sequence however is always *TACT*.

class `moclo.kits.cidar.CIDARRibosomeBindingSite` (*CIDARPart*, *CIDAREntry*)

A CIDAR ribosome binding site.

Parts of this type contain a ribosome binding site (RBS). The downstream overhang doubles as the start codon for the subsequent coding sequence.

class `moclo.kits.cidar.CIDARCodingSequence` (*CIDARPart*, *CIDAREntry*)

A CIDAR coding sequence.

Parts of this type contain a coding sequence, with the start codon located on the upstream overhang.

Caution: Although the start codon is located on the upstream overhang, a STOP codon is expected to be found within this part target sequence before the downstream overhang.

```
class moclo.kits.cidar.CIDARTerminator (CIDARPart, CIDAREntry)
    A CIDAR terminator.
```

Parts of this type contain a terminator. The upstream overhang is always the same for the terminator to directly follow the coding sequence, but the downstream overhang can vary to specify an order for a following multigenic assembly within a device.

Note: The CIDAR toolkit parts provide 4 different downstream overhangs: *GCTT*, *CGCT*, *TGCC*, and *ACTA*. These are not enforced in this module, and any downstream sequence will be accepted. The upstream sequence however is always *AGGT*.

3.1.3 Level 1

Module

```
class moclo.kits.cidar.CIDARCassette (Cassette)
    A CIDAR MoClo cassette.

cutter
    alias of Bio.Restriction.Restriction.BbsI
```

Vector

```
class moclo.kits.cidar.CIDARDeviceVector (DeviceVector)
    A CIDAR Moclo device vector.
```

References

Iverson et al., Figure 1.

```
cutter
    alias of Bio.Restriction.Restriction.BbsI

static structure ()
    Get the vector structure, as a DNA regex pattern.
```

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The downstream (3') overhang sequence
2. The vector placeholder sequence
3. The upstream (5') overhang sequence

3.1.4 Level 2

Module

class `moclo.kits.cidar.CIDARDevice` (*Device*)
A CIDAR MoClo device.

cutter
alias of `Bio.Restriction.Restriction.BsaI`

3.2 EcoFlex Kit

An implementation of the EcoFlex ToolKit for the Python MoClo library.

References

1. Moore, S. J., Lai, H.-E., Kelwick, R. J. R., Chee, S. M., Bell, D. J., Polizzi, K. M., Freemont, P. S. (2016). EcoFlex: A Multifunctional MoClo Kit for E. coli Synthetic Biology. *ACS Synthetic Biology*, 5(10), 1059–1069.
 2. Weber, E., Engler, C., Gruetzner, R., Werner, S., Marillonnet, S. (2011). A Modular Cloning System for Standardized Assembly of Multigene Constructs. *PLOS ONE*, 6(2), e16765.
-

3.2.1 Level 0

Module

class `moclo.kits.ecoflex.EcoFlexEntry` (*Entry*)
An EcoFlex MoClo entry.

EcoFlex entries are stored and shared as plasmids flanked by *BsaI* binding sites at both ends of the target sequence.

__init__ (*record*)
Initialize self. See `help(type(self))` for accurate signature.

cutter
alias of `Bio.Restriction.Restriction.BsaI`

is_valid ()
Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end ()
Get the downstream overhang of the target sequence.

Returns the downstream overhang.

Return type `Seq`

overhang_start ()
Get the upstream overhang of the target sequence.

Returns the downstream overhang.

Return type `Seq`

classmethod `structure()`

Get the module structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The module target sequence
3. The downstream (3') overhang sequence

target_sequence()

Get the target sequence of the module.

Modules are often stored in a standardized way, and contain more than the sequence of interest: for instance they can contain an antibiotic marker, that will not be part of the assembly when that module is assembled into a vector; only the target sequence is inserted.

Returns the target sequence with annotations.

Return type `SeqRecord`

Note: Depending on the cutting direction of the restriction enzyme used during assembly, the overhang will be left at the beginning or at the end, so the obtained record is exactly the sequence the enzyme created during restriction.

Vector

class `moclo.kits.ecoflex.EcoFlexCassetteVector` (*CassetteVector*)

An EcoFlex MoClo cassette vector.

__init__ (*record*)

Initialize self. See `help(type(self))` for accurate signature.

assemble (*module*, **modules*, ***kwargs*)

Assemble the provided modules into the vector.

Parameters

- **module** (`AbstractModule`) – a module to insert in the vector.
- **modules** (`AbstractModule`, optional) – additional modules to insert in the vector. The order of the parameters is not important, since modules will be sorted by their start overhang in the function.

Returns the assembled sequence with sequence annotations inherited from the vector and the modules.

Return type `SeqRecord`

Raises

- *DuplicateModules* – when two different modules share the same start overhang, leading in possibly non-deterministic constructs.
- *MissingModule* – when a module has an end overhang that is not shared by any other module, leading to a partial construct only

- *InvalidSequence* – when one of the modules does not match the required module structure (missing site, wrong overhang, etc.).
- *UnusedModules* – when some modules were not used during the assembly (mostly caused by duplicate parts).

cutter

alias of `Bio.Restriction.Restriction.BsaI`

is_valid()

Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end()

Get the downstream overhang of the vector sequence.

overhang_start()

Get the upstream overhang of the vector sequence.

placeholder_sequence()

Get the placeholder sequence in the vector.

The placeholder sequence is replaced by the concatenation of modules during the assembly. It often contains a dropout sequence, such as a GFP expression cassette that can be used to measure the progress of the assembly.

static structure()

Get the vector structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The downstream (3') overhang sequence
2. The vector placeholder sequence
3. The upstream (5') overhang sequence

target_sequence()

Get the target sequence in the vector.

The target sequence if the part of the plasmid that is not discarded during the assembly (everything except the placeholder sequence).

Parts

class `moclo.kits.ecoflex.EcoFlexPromoter` (*EcoFlexPart*, *EcoFlexEntry*)

An EcoFlex MoClo promoter.

class `moclo.kits.ecoflex.EcoFlexRBS` (*EcoFlexPart*, *EcoFlexEntry*)

An EcoFlex MoClo ribosome binding site.

Parts of this type contain a ribosome binding site (RBS). The last adenosine serves as the beginning of the start codon of the following CDS.

class moclo.kits.ecoflex.**EcoFlexTagLinker** (*EcoFlexPart*, *EcoFlexEntry*)
 An EcoFlex MoClo tag linker.

Parts of this type also contain a RBS, but they allow adding a N-terminal tag sequence before the CDS.

class moclo.kits.ecoflex.**EcoFlexTag** (*EcoFlexPart*, *EcoFlexEntry*)
 An EcoFlex MoClo N-terminal tag.

Parts of this type typically contain tags that are added to the N-terminus of the translated protein, such as a *hexa histidine* or a *Strep(II)* tag.

class moclo.kits.ecoflex.**EcoFlexCodingSequence** (*EcoFlexPart*, *EcoFlexEntry*)
 An EcoFlex MoClo coding sequence.

Parts of this type contain a coding sequence (CDS), with the start codon beginning on the upstream overhang.

Caution: Although the start codon is located on the upstream overhang, a STOP codon is expected to be found within this part target sequence before the downstream overhang.

class moclo.kits.ecoflex.**EcoFlexTerminator** (*EcoFlexPart*, *EcoFlexEntry*)
 An EcoFlex MoClo terminator.

3.2.2 Level 1

Module

class moclo.kits.ecoflex.**EcoFlexCassette** (*Cassette*)
 An EcoFlex MoClo cassette.

cutter
 alias of Bio.Restriction.Restriction.BsmBI

Vector

class moclo.kits.ecoflex.**EcoFlexDeviceVector** (*DeviceVector*)
 An EcoFlex MoClo device vector.

cutter
 alias of Bio.Restriction.Restriction.BsmBI

static structure ()
 Get the vector structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The downstream (3') overhang sequence
2. The vector placeholder sequence

3. The upstream (5') overhang sequence

3.2.3 Level 2

Module

```
class moclo.kits.ecoflex.EcoFlexDevice (Device)
    An EcoFlex MoClo device.

    cutter
        alias of Bio.Restriction.Restriction.BsaI
```

3.3 Icon Genetics Kit

An implementation of the Icon Genetics ToolKit for the Python MoClo library.

References

1. Weber, E., Engler, C., Gruetzner, R., Werner, S., Marillonnet, S. (2011). A Modular Cloning System for Standardized Assembly of Multigene Constructs. PLOS ONE, 6(2), e16765.
 2. Werner, S., Engler, C., Weber, E., Gruetzner, R., & Marillonnet, S. (2012). Fast track assembly of multigene constructs using Golden Gate cloning and the MoClo system. Bioengineered, 3(1), 38–43.
-

3.3.1 Level -1

Module

```
class moclo.kits.ig.IGProduct (Product)
    An Icon Genetics MoClo product.

    __init__ (record)
        Initialize self. See help(type(self)) for accurate signature.

    cutter
        alias of Bio.Restriction.Restriction.BpiI

    is_valid ()
        Check if the wrapped record follows the required class structure.

        Returns True if the record is valid, False otherwise.

        Return type bool

    overhang_end ()
        Get the downstream overhang of the target sequence.

        Returns the downstream overhang.

        Return type Seq

    overhang_start ()
        Get the upstream overhang of the target sequence.
```

Returns the downstream overhang.

Return type `Seq`

classmethod `structure()`

Get the module structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The module target sequence
3. The downstream (3') overhang sequence

target_sequence()

Get the target sequence of the module.

Modules are often stored in a standardized way, and contain more than the sequence of interest: for instance they can contain an antibiotic marker, that will not be part of the assembly when that module is assembled into a vector; only the target sequence is inserted.

Returns the target sequence with annotations.

Return type `SeqRecord`

Note: Depending on the cutting direction of the restriction enzyme used during assembly, the overhang will be left at the beginning or at the end, so the obtained record is exactly the sequence the enzyme created during restriction.

Vector

class `moclo.kits.ig.IGEntryVector(EntryVector)`

An Icon Genetics entry vector.

References

Weber *et al.*, Figure 2A.

__init__(*record*)

Initialize self. See `help(type(self))` for accurate signature.

assemble(*module*, **modules*, ***kwargs*)

Assemble the provided modules into the vector.

Parameters

- **module** (`AbstractModule`) – a module to insert in the vector.
- **modules** (`AbstractModule`, optional) – additional modules to insert in the vector. The order of the parameters is not important, since modules will be sorted by their start overhang in the function.

Returns the assembled sequence with sequence annotations inherited from the vector and the modules.

Return type SeqRecord

Raises

- *DuplicateModules* – when two different modules share the same start overhang, leading in possibly non-deterministic constructs.
- *MissingModule* – when a module has an end overhang that is not shared by any other module, leading to a partial construct only
- *InvalidSequence* – when one of the modules does not match the required module structure (missing site, wrong overhang, etc.).
- *UnusedModules* – when some modules were not used during the assembly (mostly caused by duplicate parts).

cutter

alias of `Bio.Restriction.Restriction.BpiI`

is_valid()

Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end()

Get the downstream overhang of the vector sequence.

overhang_start()

Get the upstream overhang of the vector sequence.

placeholder_sequence()

Get the placeholder sequence in the vector.

The placeholder sequence is replaced by the concatenation of modules during the assembly. It often contains a dropout sequence, such as a GFP expression cassette that can be used to measure the progress of the assembly.

classmethod structure()

Get the vector structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The downstream (3') overhang sequence
2. The vector placeholder sequence
3. The upstream (5') overhang sequence

target_sequence()

Get the target sequence in the vector.

The target sequence is the part of the plasmid that is not discarded during the assembly (everything except the placeholder sequence).

3.3.2 Level 0

Module

class `moclo.kits.ig.IGEntry` (*Entry*)

An Icon Genetics MoClo entry.

__init__ (*record*)

Initialize self. See `help(type(self))` for accurate signature.

cutter

alias of `Bio.Restriction.Restriction.BsaI`

is_valid ()

Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end ()

Get the downstream overhang of the target sequence.

Returns the downstream overhang.

Return type `Seq`

overhang_start ()

Get the upstream overhang of the target sequence.

Returns the downstream overhang.

Return type `Seq`

classmethod **structure** ()

Get the module structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The module target sequence
3. The downstream (3') overhang sequence

target_sequence ()

Get the target sequence of the module.

Modules are often stored in a standardized way, and contain more than the sequence of interest: for instance they can contain an antibiotic marker, that will not be part of the assembly when that module is assembled into a vector; only the target sequence is inserted.

Returns the target sequence with annotations.

Return type `SeqRecord`

Note: Depending on the cutting direction of the restriction enzyme used during assembly, the overhang will be left at the beginning or at the end, so the obtained record is exactly the sequence the enzyme created during restriction.

Vector

class `moclo.kits.ig.IGCassetteVector` (*CassetteVector*)
An Icon Genetics cassette vector.

References

Weber et al., Figure 4A.

__init__ (*record*)
Initialize self. See `help(type(self))` for accurate signature.

assemble (*module*, **modules*, ***kwargs*)
Assemble the provided modules into the vector.

Parameters

- **module** (*AbstractModule*) – a module to insert in the vector.
- **modules** (*AbstractModule*, optional) – additional modules to insert in the vector. The order of the parameters is not important, since modules will be sorted by their start overhang in the function.

Returns the assembled sequence with sequence annotations inherited from the vector and the modules.

Return type `SeqRecord`

Raises

- *DuplicateModules* – when two different modules share the same start overhang, leading in possibly non-deterministic constructs.
- *MissingModule* – when a module has an end overhang that is not shared by any other module, leading to a partial construct only
- *InvalidSequence* – when one of the modules does not match the required module structure (missing site, wrong overhang, etc.).
- *UnusedModules* – when some modules were not used during the assembly (mostly caused by duplicate parts).

cutter
alias of `Bio.Restriction.Restriction.BsaI`

is_valid ()
Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end ()
Get the downstream overhang of the vector sequence.

overhang_start ()
Get the upstream overhang of the vector sequence.

placeholder_sequence ()
Get the placeholder sequence in the vector.

The placeholder sequence is replaced by the concatenation of modules during the assembly. It often contains a dropout sequence, such as a GFP expression cassette that can be used to measure the progress of the assembly.

classmethod structure()

Get the vector structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The downstream (3') overhang sequence
2. The vector placeholder sequence
3. The upstream (5') overhang sequence

target_sequence()

Get the target sequence in the vector.

The target sequence is the part of the plasmid that is not discarded during the assembly (everything except the placeholder sequence).

Parts

class moclo.kits.ig.IGPromoter (IGPart, IEntry)

An Icon Genetics promoter part.

class moclo.kits.ig.IGUntranslatedRegion (IGPart, IEntry)

An Icon Genetics 5' UTR part.

class moclo.kits.ig.IGSignalPeptide (IGPart, IEntry)

An Icon Genetics signal peptide part.

class moclo.kits.ig.IGCodingSequence (IGPart, IEntry)

An Icon Genetics CDS part.

class moclo.kits.ig.IGTerminator (IGPart, IEntry)

An Icon Genetics terminator part.

3.3.3 Level 1

Module

class moclo.kits.ig.IGCassette (Cassette)

An Icon Genetics MoClo cassette.

cutter

alias of Bio.Restriction.Restriction.BpiI

Vector

class moclo.kits.ig.IGDeviceVector (DeviceVector)

An Icon Genetics device vector.

References

Weber *et al.*, Figure 4A.

cutter

alias of `Bio.Restriction.Restriction.BpiI`

Parts

class `moclo.kits.ig.IGEndLinker` (*IGPart*, *IGCassette*)
An Icon Genetic end linker part.

References

Weber *et al.*, Figure 5.

3.3.4 Level M

Parts

class `moclo.kits.ig.IGLevelMVector` (*IGPart*, *IGDeviceVector*)

cutter

alias of `Bio.Restriction.Restriction.BpiI`

classmethod structure ()

Get the part structure, as a DNA regex pattern.

The structure of most parts can be obtained automatically from the part signature and the restriction enzyme used in the Golden Gate assembly.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The vector placeholder sequence
3. The downstream (3') overhang sequence

class `moclo.kits.ig.IGLevelMEndLinker` (*IGPart*, *IGCassette*)

classmethod structure ()

Get the part structure, as a DNA regex pattern.

The structure of most parts can be obtained automatically from the part signature and the restriction enzyme used in the Golden Gate assembly.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence

2. The vector placeholder sequence
3. The downstream (3') overhang sequence

3.3.5 Level P

Parts

class moclo.kits.ig.IGLevelPVector (*IGPart*, *IGCassetteVector*)

cutter

alias of Bio.Restriction.Restriction.BsaI

classmethod structure()

Get the part structure, as a DNA regex pattern.

The structure of most parts can be obtained automatically from the part signature and the restriction enzyme used in the Golden Gate assembly.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The vector placeholder sequence
3. The downstream (3') overhang sequence

class moclo.kits.ig.IGLevelPEndLinker (*IGPart*, *IGEntry*)

cutter

alias of Bio.Restriction.Restriction.BsaI

classmethod structure()

Get the part structure, as a DNA regex pattern.

The structure of most parts can be obtained automatically from the part signature and the restriction enzyme used in the Golden Gate assembly.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The vector placeholder sequence
3. The downstream (3') overhang sequence

3.4 Yeast ToolKit (YTK) / Pichia ToolKit (PTK)

An implementation of the Yeast ToolKit for the Python MoClo library.

This module is tested against the official parts available in the Yeast ToolKit (YTK), and also against the Pichia ToolKit (PTK) parts since they were designed to be compatible with each other.

The documentation of this module is mostly adapted from the *Lee et al.* supplementary data. Each item also has specific sections that are organized as follow:

Note: this section describes a behaviour that is not part of the YTK standard, but that is implemented in all YTK official parts, and encouraged to follow by the YTK authors.

Caution this section describes a behaviour that goes against the MoClo standard, but which you are entitled to follow for your parts to be valid YTK parts.

Danger this section describes a quirk specific to the `moclo-ytk` library.

References

1. Lee, M. E., DeLoache, W. C., Cervantes, B., Dueber, J. E. (2015). A Highly Characterized Yeast Toolkit for Modular, Multipart Assembly. *ACS Synthetic Biology*, 4(9), 975–986.
 2. Obst, U., Lu, T. K., Sieber, V. (2017). A Modular Toolkit for Generating Pichia pastoris Secretion Libraries. *ACS Synthetic Biology*, 6(6), 1016–1025
 3. Weber, E., Engler, C., Gruetzner, R., Werner, S., Marillonnet, S. (2011). A Modular Cloning System for Standardized Assembly of Multigene Constructs. *PLOS ONE*, 6(2), e16765.
-

3.4.1 Level -1

Module

class `moclo.kits.ytk.YTKProduct` (*Product*)
A MoClo Yeast ToolKit product.

As the YTK entry vector does not contain the required *BsaI* restriction site, the site must be contained in the product sequence.

Caution: The standard construction describe in the *Lee et al.* paper directly inserts the beginning of the *BsaI* recognition site inside of the two *BsmBI* overhangs at both ends of the product. Other valid constructs that do not proceed like so won't be considered a valid product, although they contain the required *BsaI* site.

References

Lee et al., Supplementary Figure S19.

__init__ (*record*)
Initialize self. See `help(type(self))` for accurate signature.

cutter
alias of `Bio.Restriction.Restriction.BsmBI`

is_valid ()
Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end()

Get the downstream overhang of the target sequence.

Returns the downstream overhang.

Return type Seq

overhang_start()

Get the upstream overhang of the target sequence.

Returns the downstream overhang.

Return type Seq

static structure()

Get the module structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence
2. The module target sequence
3. The downstream (3') overhang sequence

target_sequence()

Get the target sequence of the module.

Modules are often stored in a standardized way, and contain more than the sequence of interest: for instance they can contain an antibiotic marker, that will not be part of the assembly when that module is assembled into a vector; only the target sequence is inserted.

Returns the target sequence with annotations.

Return type SeqRecord

Note: Depending on the cutting direction of the restriction enzyme used during assembly, the overhang will be left at the beginning or at the end, so the obtained record is exactly the sequence the enzyme created during restriction.

Vector

class moclo.kits.ytk.YTKEntryVector (EntryVector)

A MoClo Yeast ToolKit entry vector.

Any plasmid with two *BsmBI* restriction sites can be used to create a YTK entry, although the toolkit-provided entry vector (*pYTK001*) is probably the most appropriate plasmid to use.

Caution: To the contrary of the usual MoClo entry vectors described in the *Weber et al.* paper, the YTK entry vectors do not provide another *BsaI* restriction site enclosing the placeholder sequence. As such, YTK Level -1 modules must embed the *BsaI* binding site.

__init__(record)

Initialize self. See help(type(self)) for accurate signature.

assemble (*module*, **modules*, ***kwargs*)

Assemble the provided modules into the vector.

Parameters

- **module** (*AbstractModule*) – a module to insert in the vector.
- **modules** (*AbstractModule*, optional) – additional modules to insert in the vector. The order of the parameters is not important, since modules will be sorted by their start overhang in the function.

Returns the assembled sequence with sequence annotations inherited from the vector and the modules.

Return type *SeqRecord*

Raises

- *DuplicateModules* – when two different modules share the same start overhang, leading in possibly non-deterministic constructs.
- *MissingModule* – when a module has an end overhang that is not shared by any other module, leading to a partial construct only
- *InvalidSequence* – when one of the modules does not match the required module structure (missing site, wrong overhang, etc.).
- *UnusedModules* – when some modules were not used during the assembly (mostly caused by duplicate parts).

cutter

alias of `Bio.Restriction.Restriction.BsmBI`

is_valid()

Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end()

Get the downstream overhang of the vector sequence.

overhang_start()

Get the upstream overhang of the vector sequence.

placeholder_sequence()

Get the placeholder sequence in the vector.

The placeholder sequence is replaced by the concatenation of modules during the assembly. It often contains a dropout sequence, such as a GFP expression cassette that can be used to measure the progress of the assembly.

classmethod structure()

Get the vector structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The downstream (3') overhang sequence
2. The vector placeholder sequence
3. The upstream (5') overhang sequence

target_sequence()

Get the target sequence in the vector.

The target sequence is the part of the plasmid that is not discarded during the assembly (everything except the placeholder sequence).

3.4.2 Level 0

Module

class `moclo.kits.ytk.YTKEntry` (*Entry*)

A MoClo Yeast ToolKit entry.

YTK entries are stored and shared as plasmids flanked by *BsaI* binding sites at both ends of the target sequence.

Danger: Although the *BsaI* binding sites are not located within the target sequence for almost all the standard toolkit parts, special Type 234r parts have these sites reversed, because these parts are used to assemble cassette vectors and require the final construct to contain a *BsaI* site to allow assembly with other parts. **Those parts will not match** the default *YTKEntry*, and must be used as *YTKPart234r* instances for the assembly logic to work as expected.

__init__ (*record*)

Initialize self. See `help(type(self))` for accurate signature.

cutter

alias of `Bio.Restriction.Restriction.BsaI`

is_valid()

Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end()

Get the downstream overhang of the target sequence.

Returns the downstream overhang.

Return type `Seq`

overhang_start()

Get the upstream overhang of the target sequence.

Returns the downstream overhang.

Return type `Seq`

classmethod structure()

Get the module structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The upstream (5') overhang sequence

2. The module target sequence
3. The downstream (3') overhang sequence

target_sequence ()

Get the target sequence of the module.

Modules are often stored in a standardized way, and contain more than the sequence of interest: for instance they can contain an antibiotic marker, that will not be part of the assembly when that module is assembled into a vector; only the target sequence is inserted.

Returns the target sequence with annotations.

Return type SeqRecord

Note: Depending on the cutting direction of the restriction enzyme used during assembly, the overhang will be left at the beginning or at the end, so the obtained record is exactly the sequence the enzyme created during restriction.

Vector

class moclo.kits.ytk.YTKCassetteVector (*CassetteVector*)

A MoClo Yeast ToolKit cassette vector.

The YTK provides a canonical integration plasmid, preassembled from several other parts, that can be used as a cassette vector for an assembly of Type 2, 3 and 4 parts. Type 8, 8a and 678 parts are also considered as cassette vectors.

References

Lee et al., Figure 2.

__init__ (record)

Initialize self. See help(type(self)) for accurate signature.

assemble (module, *modules, **kwargs)

Assemble the provided modules into the vector.

Parameters

- **module** (*AbstractModule*) – a module to insert in the vector.
- **modules** (*AbstractModule*, optional) – additional modules to insert in the vector. The order of the parameters is not important, since modules will be sorted by their start overhang in the function.

Returns the assembled sequence with sequence annotations inherited from the vector and the modules.

Return type SeqRecord

Raises

- *DuplicateModules* – when two different modules share the same start overhang, leading in possibly non-deterministic constructs.

- *MissingModule* – when a module has an end overhang that is not shared by any other module, leading to a partial construct only
- *InvalidSequence* – when one of the modules does not match the required module structure (missing site, wrong overhang, etc.).
- *UnusedModules* – when some modules were not used during the assembly (mostly caused by duplicate parts).

cutter

alias of `Bio.Restriction.Restriction.BsaI`

is_valid()

Check if the wrapped record follows the required class structure.

Returns `True` if the record is valid, `False` otherwise.

Return type `bool`

overhang_end()

Get the downstream overhang of the vector sequence.

overhang_start()

Get the upstream overhang of the vector sequence.

placeholder_sequence()

Get the placeholder sequence in the vector.

The placeholder sequence is replaced by the concatenation of modules during the assembly. It often contains a dropout sequence, such as a GFP expression cassette that can be used to measure the progress of the assembly.

classmethod structure()

Get the vector structure, as a DNA regex pattern.

Warning: If overloading this method, the returned pattern must include 3 capture groups to capture the following features:

1. The downstream (3') overhang sequence
2. The vector placeholder sequence
3. The upstream (5') overhang sequence

target_sequence()

Get the target sequence in the vector.

The target sequence is the part of the plasmid that is not discarded during the assembly (everything except the placeholder sequence).

Parts

Base Parts

class `moclo.kits.ytk.YTKPart1` (*YTKPart*, *YTKEntry*)

A YTK Type 1 part (**Upstream assembly connector**).

Parts of this type contain non-coding and non-regulatory sequences that are used to direct assembly of multigene plasmids, such as ligation sites for other Type IIS endonucleases (e.g. *BsmBI*).

Note: Official toolkit Type 1 parts also include a *EcoRI* and *XbaI* site just after the upstream overhang for BioBrick compatibility of the assembled cassettes and multi-gene plasmids.

class `moclo.kits.ytk.YTKPart2` (*YTKPart*, *YTKEntry*)
A YTK Type 2 part (**Promoter**).

Parts of this type contain a promoter. The downstream overhang doubles as the start codon for the subsequent Type 3 or Type 3a coding sequence.

Note: Official toolkit Type 2 parts also include a *BglIII* site immediately preceding the start codon (overlapping the downstream overhang) for BglBrick compatibility.

class `moclo.kits.ytk.YTKPart3` (*YTKPart*, *YTKEntry*)
A YTK Type 3 part (**Coding sequence**).

Parts of this type contain a coding sequence, with the start codon located on the upstream overhang. If a stop codon is omitted from the part, and two bases are added before the downstream overhang, the resulting site can be used as a two amino acid linker to a Type 4 or 4a C-terminal fusion.

Note: Official toolkit Type 3 parts also include a *BamHI* recognition site at the end of the included CDS (overlapping the downstream overhang) for BglBrick compatibility.

class `moclo.kits.ytk.YTKPart3a` (*YTKPart*, *YTKEntry*)
A YTK Type 3a part (**N-terminal coding sequence**).

class `moclo.kits.ytk.YTKPart3b` (*YTKPart*, *YTKEntry*)
A YTK Type 3b part (**C-terminal coding sequence**).

Note: As with Type 3 parts, official toolkits Type 3b parts also include a *BamHI* recognition site at the end of the included CDS (overlapping the downstream overhang) for BglBrick compatibility.

class `moclo.kits.ytk.YTKPart4` (*YTKPart*, *YTKEntry*)
A YTK Type 4 part (**Transcriptional terminator**).

As Type 3 parts do not include a stop codon, parts of this type should encode an in-frame stop codon before the transcriptional terminator. Commonly used C-terminal fusions, such as purification or epitope tags, but it is recommended to use *YTKPart4a* and *YTKPart4b* subtypes instead.

Note: Official toolkit Type 4 parts all start by a stop codon directly after the upstream overhang, followed by a *XhoI* recognition site which enables BglBrick compatibility, then followed by the terminator sequence itself.

```
class moclo.kits.ytk.YTKPart4a (YTKPart, YTKEntry)
    A YTK Type 4a part (C-terminal tag sequence).
```

Type 4a parts contain additional coding sequences that will be fused to the C-terminal extremity of the protein. These parts include, but are not limited to: localisation tags, purification tags, fluorescent proteins.

Caution: In contrast to the Type 3 and 3b parts, the convention for 4a parts is to include the stop codon rather than enable read-through of the downstream overhang, although that convention it is not enforced.

Note: Official toolkit Type 4a parts contain a stop codon after the CDS, itself immediately followed by a *XhoI* recognition site just before the downstream overhang, for BglBrick compatibility.

```
class moclo.kits.ytk.YTKPart4b (YTKPart, YTKEntry)
    A YTK Type 4b part (Terminator sequence).
```

Type 4b contain transcriptional terminators, but are not required to encode an in-frame start codon, as it should be located in the Type 4a part that precedes it.

```
class moclo.kits.ytk.YTKPart5 (YTKPart, YTKEntry)
    A YTK Type 5 part (Downstream assembly connector).
```

As with Type 1 parts, parts of this type provide sequences such as restriction enzymes recognition sites, for instance in order to direct multigene expression plasmids.

Note: Official toolkit parts also include a *SpeI* and *PstI* site at the end of the part sequence for BioBrick compatibility of the assembled cassettes and multi-gene plasmids.

```
class moclo.kits.ytk.YTKPart6 (YTKPart, YTKEntry)
    A YTK Type 6 part (Yeast marker).
```

Parts of this type contain a selectable marker for *S. cerevisiae*, as a full expression cassette (promoter, ORF, and terminal) for conferring the selectable phenotype (such as drug-resistance or bioluminescence).

```
class moclo.kits.ytk.YTKPart7 (YTKPart, YTKEntry)
    A YTK Part Type 7 part (Yeast origin / 3' homology).
```

Depending on the expression organism (*E.coli* or *S. cerevisiae*), this sequence will either hold a yeast origin of replication, or a 3' homology sequence for integration in the bacterial genome.

```
class moclo.kits.ytk.YTKPart8 (YTKPart, YTKCassetteVector)
    A YTK Type 8 part (Bacterial origin & marker).
```

Parts of this type contain a bacterial origin of replication, as well as an antibiotic resistance marker. They act as the Golden Gate Assembly vector when assembling a cassette, and as such should also embed a dropout sequence, such as a fluorescent protein expression cassette.

Note: Official toolkit parts use an mRFP coding sequence as the dropout, and also include *NotI* restriction site at each end of the part to allow the verification of new assemblies.

class moclo.kits.ytk.YTKPart8a (YTKPart, YTKCassetteVector)
A YTK Part 8a part (**Bacterial origin & marker**).

Parts of this type, like Type 8 parts, include a bacterial origin of replication and an antibiotic resistance marker, and act as Assembly vectors.

Note: Official toolkit parts use an mRFP coding sequence as the dropout, and also include *NotI* restriction site at each end of the part so the integration plasmid can be linearized prior to transformation into yeast.

class moclo.kits.ytk.YTKPart8b (YTKPart, YTKEntry)
A YTK Type 8b part (**5' homology**).

As with certain Type 7 parts, parts of this type contain long sequences of homology to the genome that is upstream of the target locus.

Composite

class moclo.kits.ytk.YTKPart234 (YTKPart, YTKEntry)
A YTK Type 234 part (**Composite 2, 3, 4**).

Type 234 parts are composed of a complete expression cassette (promoter, coding sequence, and terminator) fused into a single part, instead of separate Type 2, 3 and 4 parts.

class moclo.kits.ytk.YTKPart234r (YTKPart, YTKEntry)
A YTK Type 234 part (**Composite 2, 3, 4**) with reversed BsaI sites.

Type 234r parts are designed so that the BsaI sites are kept within the final cassette. They are used to assemble canonical integration vectors, where the Type 234 part acts as a placeholder until replaced by actual Type 2, 3 and 4 parts in the final construct.

class moclo.kits.ytk.YTKPart678 (YTKPart, YTKCassetteVector)
A YTK Type 678 part (**Composite 6, 7, 8**).

Type 678 parts are used when there is no requirement for yeast markers and origins to be included in the final assembly, for instance when assembling an intermediary plasmid acting as a vector for a multi-gene construct.

3.4.3 Level 1

Module

class moclo.kits.ytk.YTKCassette (Cassette)
A MoClo Yeast ToolKit cassette.

cutter

alias of `Bio.Restriction.Restriction.BsmBI`

Vector

class `moclo.kits.ytk.YTKDeviceVector` (*DeviceVector*)

A MoClo Yeast ToolKit multigene vector.

Parts of Type 1 and 5 are used to order the cassette plasmids within the multigene assembly. The vector always contains a ConLS and ConRE parts.

References

Lee et al., Supplementary Figure S21.

cutter

alias of `Bio.Restriction.Restriction.BsmBI`

CHAPTER 4

Indices and tables

- `genindex`
- `modindex`
- `search`

m

- `moclo.core.modules`, [22](#)
- `moclo.core.parts`, [26](#)
- `moclo.core.vectors`, [24](#)
- `moclo.errors`, [27](#)
- `moclo.kits.cidar`, [37](#)
- `moclo.kits.ecoflex`, [44](#)
- `moclo.kits.ig`, [48](#)
- `moclo.kits.ytk`, [55](#)
- `moclo.record`, [21](#)
- `moclo.registry.base`, [22](#)

Symbols

[__add__\(\) \(moclo.record.CircularRecord method\), 21](#)
[__contains__\(\) \(moclo.record.CircularRecord method\), 21](#)
[__getitem__\(\) \(moclo.record.CircularRecord method\), 21](#)
[__init__\(\) \(moclo.core.modules.AbstractModule method\), 23](#)
[__init__\(\) \(moclo.core.parts.AbstractPart method\), 26](#)
[__init__\(\) \(moclo.kits.cidar.CIDARCassetteVector method\), 41](#)
[__init__\(\) \(moclo.kits.cidar.CIDAREntry method\), 40](#)
[__init__\(\) \(moclo.kits.cidar.CIDAREntryVector method\), 38](#)
[__init__\(\) \(moclo.kits.cidar.CIDARProduct method\), 37](#)
[__init__\(\) \(moclo.kits.ecoflex.EcoFlexCassetteVector method\), 45](#)
[__init__\(\) \(moclo.kits.ecoflex.EcoFlexEntry method\), 44](#)
[__init__\(\) \(moclo.kits.ig.IGCassetteVector method\), 52](#)
[__init__\(\) \(moclo.kits.ig.IGEntry method\), 51](#)
[__init__\(\) \(moclo.kits.ig.IGEntryVector method\), 49](#)
[__init__\(\) \(moclo.kits.ig.IGProduct method\), 48](#)
[__init__\(\) \(moclo.kits.ytk.YTKCassetteVector method\), 60](#)
[__init__\(\) \(moclo.kits.ytk.YTKEntry method\), 59](#)
[__init__\(\) \(moclo.kits.ytk.YTKEntryVector method\), 57](#)
[__init__\(\) \(moclo.kits.ytk.YTKProduct method\), 56](#)
[__init__\(\) \(moclo.record.CircularRecord method\), 21](#)
[__init__\(\) \(moclo.registry.base.CombinedRegistry method\), 22](#)
[__lshift__\(\) \(moclo.record.CircularRecord method\), 22](#)

[__radd__\(\) \(moclo.record.CircularRecord method\), 22](#)
[__rshift__\(\) \(moclo.record.CircularRecord method\), 22](#)

A

[AbstractModule \(class in moclo.core.modules\), 23](#)
[AbstractPart \(class in moclo.core.parts\), 26](#)
[AbstractRegistry \(class in moclo.registry.base\), 22](#)
[AbstractVector \(class in moclo.core.vectors\), 24](#)
[assemble\(\) \(moclo.core.vectors.AbstractVector method\), 24](#)
[assemble\(\) \(moclo.kits.cidar.CIDARCassetteVector method\), 41](#)
[assemble\(\) \(moclo.kits.cidar.CIDAREntryVector method\), 38](#)
[assemble\(\) \(moclo.kits.ecoflex.EcoFlexCassetteVector method\), 45](#)
[assemble\(\) \(moclo.kits.ig.IGCassetteVector method\), 52](#)
[assemble\(\) \(moclo.kits.ig.IGEntryVector method\), 49](#)
[assemble\(\) \(moclo.kits.ytk.YTKCassetteVector method\), 60](#)
[assemble\(\) \(moclo.kits.ytk.YTKEntryVector method\), 57](#)
[AssemblyError \(class in moclo.errors\), 27](#)
[AssemblyWarning \(class in moclo.errors\), 27](#)

C

[Cassette \(class in moclo.core.modules\), 24](#)
[CassetteVector \(class in moclo.core.vectors\), 26](#)
[characterize\(\) \(moclo.core.parts.AbstractPart class method\), 26](#)
[CIDARCassette \(class in moclo.kits.cidar\), 43](#)
[CIDARCassetteVector \(class in moclo.kits.cidar\), 41](#)
[CIDARCodingSequence \(class in moclo.kits.cidar\), 42](#)
[CIDARDevice \(class in moclo.kits.cidar\), 44](#)

[CIDARDeviceVector \(class in moclo.kits.cidar\)](#), [43](#)
[CIDAREntry \(class in moclo.kits.cidar\)](#), [40](#)
[CIDAREntryVector \(class in moclo.kits.cidar\)](#), [38](#)
[CIDARProduct \(class in moclo.kits.cidar\)](#), [37](#)
[CIDARPromoter \(class in moclo.kits.cidar\)](#), [42](#)
[CIDARRibosomeBindingSite \(class in moclo.kits.cidar\)](#), [42](#)
[CIDARTerminator \(class in moclo.kits.cidar\)](#), [42](#)
[CircularRecord \(class in moclo.record\)](#), [21](#)
[CombinedRegistry \(class in moclo.registry.base\)](#), [22](#)
[cutter \(moclo.core.modules.AbstractModule attribute\)](#), [23](#)
[cutter \(moclo.kits.cidar.CIDARCassette attribute\)](#), [43](#)
[cutter \(moclo.kits.cidar.CIDARCassetteVector attribute\)](#), [41](#)
[cutter \(moclo.kits.cidar.CIDARDevice attribute\)](#), [44](#)
[cutter \(moclo.kits.cidar.CIDARDeviceVector attribute\)](#), [43](#)
[cutter \(moclo.kits.cidar.CIDAREntry attribute\)](#), [40](#)
[cutter \(moclo.kits.cidar.CIDAREntryVector attribute\)](#), [39](#)
[cutter \(moclo.kits.cidar.CIDARProduct attribute\)](#), [37](#)
[cutter \(moclo.kits.ecoflex.EcoFlexCassette attribute\)](#), [47](#)
[cutter \(moclo.kits.ecoflex.EcoFlexCassetteVector attribute\)](#), [46](#)
[cutter \(moclo.kits.ecoflex.EcoFlexDevice attribute\)](#), [48](#)
[cutter \(moclo.kits.ecoflex.EcoFlexDeviceVector attribute\)](#), [47](#)
[cutter \(moclo.kits.ecoflex.EcoFlexEntry attribute\)](#), [44](#)
[cutter \(moclo.kits.ig.IGCassette attribute\)](#), [53](#)
[cutter \(moclo.kits.ig.IGCassetteVector attribute\)](#), [52](#)
[cutter \(moclo.kits.ig.IGDeviceVector attribute\)](#), [54](#)
[cutter \(moclo.kits.ig.IGEntry attribute\)](#), [51](#)
[cutter \(moclo.kits.ig.IGEntryVector attribute\)](#), [50](#)
[cutter \(moclo.kits.ig.IGLevelMVector attribute\)](#), [54](#)
[cutter \(moclo.kits.ig.IGLevelPEndLinker attribute\)](#), [55](#)
[cutter \(moclo.kits.ig.IGLevelPVector attribute\)](#), [55](#)
[cutter \(moclo.kits.ig.IGProduct attribute\)](#), [48](#)
[cutter \(moclo.kits.ytk.YTKCassette attribute\)](#), [64](#)
[cutter \(moclo.kits.ytk.YTKCassetteVector attribute\)](#), [61](#)
[cutter \(moclo.kits.ytk.YTKDeviceVector attribute\)](#), [65](#)
[cutter \(moclo.kits.ytk.YTKEntry attribute\)](#), [59](#)
[cutter \(moclo.kits.ytk.YTKEntryVector attribute\)](#), [58](#)
[cutter \(moclo.kits.ytk.YTKProduct attribute\)](#), [56](#)

D

[Device \(class in moclo.core.modules\)](#), [24](#)
[DeviceVector \(class in moclo.core.vectors\)](#), [26](#)
[DuplicateModules \(class in moclo.errors\)](#), [27](#)

E

[EcoFlexCassette \(class in moclo.kits.ecoflex\)](#), [47](#)

[EcoFlexCassetteVector \(class in moclo.kits.ecoflex\)](#), [45](#)
[EcoFlexCodingSequence \(class in moclo.kits.ecoflex\)](#), [47](#)
[EcoFlexDevice \(class in moclo.kits.ecoflex\)](#), [48](#)
[EcoFlexDeviceVector \(class in moclo.kits.ecoflex\)](#), [47](#)
[EcoFlexEntry \(class in moclo.kits.ecoflex\)](#), [44](#)
[EcoFlexPromoter \(class in moclo.kits.ecoflex\)](#), [46](#)
[EcoFlexRBS \(class in moclo.kits.ecoflex\)](#), [46](#)
[EcoFlexTag \(class in moclo.kits.ecoflex\)](#), [47](#)
[EcoFlexTagLinker \(class in moclo.kits.ecoflex\)](#), [46](#)
[EcoFlexTerminator \(class in moclo.kits.ecoflex\)](#), [47](#)
[EmbeddedRegistry \(class in moclo.registry.base\)](#), [22](#)
[Entry \(class in moclo.core.modules\)](#), [24](#)
[EntryVector \(class in moclo.core.vectors\)](#), [25](#)

I

[IGCassette \(class in moclo.kits.ig\)](#), [53](#)
[IGCassetteVector \(class in moclo.kits.ig\)](#), [52](#)
[IGCodingSequence \(class in moclo.kits.ig\)](#), [53](#)
[IGDeviceVector \(class in moclo.kits.ig\)](#), [53](#)
[IGEndLinker \(class in moclo.kits.ig\)](#), [54](#)
[IGEntry \(class in moclo.kits.ig\)](#), [51](#)
[IGEntryVector \(class in moclo.kits.ig\)](#), [49](#)
[IGLevelMEndLinker \(class in moclo.kits.ig\)](#), [54](#)
[IGLevelMVector \(class in moclo.kits.ig\)](#), [54](#)
[IGLevelPEndLinker \(class in moclo.kits.ig\)](#), [55](#)
[IGLevelPVector \(class in moclo.kits.ig\)](#), [55](#)
[IGProduct \(class in moclo.kits.ig\)](#), [48](#)
[IGPromoter \(class in moclo.kits.ig\)](#), [53](#)
[IGSignalPeptide \(class in moclo.kits.ig\)](#), [53](#)
[IGTerminator \(class in moclo.kits.ig\)](#), [53](#)
[IGUntranslatedRegion \(class in moclo.kits.ig\)](#), [53](#)
[IllegalSite \(class in moclo.errors\)](#), [27](#)
[InvalidSequence \(class in moclo.errors\)](#), [27](#)
[is_valid\(\) \(moclo.core.modules.AbstractModule method\)](#), [23](#)
[is_valid\(\) \(moclo.core.parts.AbstractPart method\)](#), [26](#)
[is_valid\(\) \(moclo.kits.cidar.CIDARCassetteVector method\)](#), [41](#)
[is_valid\(\) \(moclo.kits.cidar.CIDAREntry method\)](#), [40](#)
[is_valid\(\) \(moclo.kits.cidar.CIDAREntryVector method\)](#), [39](#)
[is_valid\(\) \(moclo.kits.cidar.CIDARProduct method\)](#), [37](#)
[is_valid\(\) \(moclo.kits.ecoflex.EcoFlexCassetteVector method\)](#), [46](#)
[is_valid\(\) \(moclo.kits.ecoflex.EcoFlexEntry method\)](#), [44](#)
[is_valid\(\) \(moclo.kits.ig.IGCassetteVector method\)](#), [52](#)

is_valid() (*moclo.kits.ig.IGEntry method*), 51
 is_valid() (*moclo.kits.ig.IGEntryVector method*), 50
 is_valid() (*moclo.kits.ig.IGProduct method*), 48
 is_valid() (*moclo.kits.ytk.YTKCassetteVector method*), 61
 is_valid() (*moclo.kits.ytk.YTKEntry method*), 59
 is_valid() (*moclo.kits.ytk.YTKEntryVector method*), 58
 is_valid() (*moclo.kits.ytk.YTKProduct method*), 56

M

MissingModule (*class in moclo.errors*), 27
 moclo.core.modules (*module*), 22
 moclo.core.parts (*module*), 26
 moclo.core.vectors (*module*), 24
 moclo.errors (*module*), 27
 moclo.kits.cidar (*module*), 37
 moclo.kits.ecoflex (*module*), 44
 moclo.kits.ig (*module*), 48
 moclo.kits.ytk (*module*), 55
 moclo.record (*module*), 21
 moclo.registry.base (*module*), 22
 MocloError (*class in moclo.errors*), 27

O

overhang_end() (*moclo.core.modules.AbstractModule method*), 23
 overhang_end() (*moclo.core.vectors.AbstractVector method*), 25
 overhang_end() (*moclo.kits.cidar.CIDARCassetteVector method*), 41
 overhang_end() (*moclo.kits.cidar.CIDAREntry method*), 40
 overhang_end() (*moclo.kits.cidar.CIDAREntryVector method*), 39
 overhang_end() (*moclo.kits.cidar.CIDARProduct method*), 37
 overhang_end() (*moclo.kits.ecoflex.EcoFlexCassetteVector method*), 46
 overhang_end() (*moclo.kits.ecoflex.EcoFlexEntry method*), 44
 overhang_end() (*moclo.kits.ig.IGCassetteVector method*), 52
 overhang_end() (*moclo.kits.ig.IGEntry method*), 51
 overhang_end() (*moclo.kits.ig.IGEntryVector method*), 50
 overhang_end() (*moclo.kits.ig.IGProduct method*), 48
 overhang_end() (*moclo.kits.ytk.YTKCassetteVector method*), 61

overhang_end() (*moclo.kits.ytk.YTKEntry method*), 59
 overhang_end() (*moclo.kits.ytk.YTKEntryVector method*), 58
 overhang_end() (*moclo.kits.ytk.YTKProduct method*), 56
 overhang_start() (*moclo.core.modules.AbstractModule method*), 23
 overhang_start() (*moclo.core.vectors.AbstractVector method*), 25
 overhang_start() (*moclo.kits.cidar.CIDARCassetteVector method*), 41
 overhang_start() (*moclo.kits.cidar.CIDAREntry method*), 40
 overhang_start() (*moclo.kits.cidar.CIDAREntryVector method*), 39
 overhang_start() (*moclo.kits.cidar.CIDARProduct method*), 38
 overhang_start() (*moclo.kits.ecoflex.EcoFlexCassetteVector method*), 46
 overhang_start() (*moclo.kits.ecoflex.EcoFlexEntry method*), 44
 overhang_start() (*moclo.kits.ig.IGCassetteVector method*), 52
 overhang_start() (*moclo.kits.ig.IGEntry method*), 51
 overhang_start() (*moclo.kits.ig.IGEntryVector method*), 50
 overhang_start() (*moclo.kits.ig.IGProduct method*), 48
 overhang_start() (*moclo.kits.ytk.YTKCassetteVector method*), 61
 overhang_start() (*moclo.kits.ytk.YTKEntry method*), 59
 overhang_start() (*moclo.kits.ytk.YTKEntryVector method*), 58
 overhang_start() (*moclo.kits.ytk.YTKProduct method*), 57

P

placeholder_sequence() (*moclo.core.vectors.AbstractVector method*), 25
 placeholder_sequence() (*moclo.kits.cidar.CIDARCassetteVector method*), 41
 placeholder_sequence() (*moclo.kits.cidar.CIDAREntryVector method*),

placeholder_sequence() (moclo.kits.ecoflex.EcoFlexCassetteVector method), 46

placeholder_sequence() (moclo.kits.ig.IGCassetteVector method), 52

placeholder_sequence() (moclo.kits.ig.IGEntryVector method), 50

placeholder_sequence() (moclo.kits.ytk.YTKCassetteVector method), 61

placeholder_sequence() (moclo.kits.ytk.YTKEntryVector method), 58

Product (class in moclo.core.modules), 24

R

reverse_complement() (moclo.record.CircularRecord method), 22

S

structure() (moclo.core.modules.AbstractModule class method), 23

structure() (moclo.core.parts.AbstractPart class method), 26

structure() (moclo.core.vectors.AbstractVector class method), 25

structure() (moclo.kits.cidar.CIDARCassetteVector static method), 42

structure() (moclo.kits.cidar.CIDARDeviceVector static method), 43

structure() (moclo.kits.cidar.CIDAREntry class method), 40

structure() (moclo.kits.cidar.CIDAREntryVector static method), 39

structure() (moclo.kits.cidar.CIDARProduct class method), 38

structure() (moclo.kits.ecoflex.EcoFlexCassetteVector static method), 46

structure() (moclo.kits.ecoflex.EcoFlexDeviceVector static method), 47

structure() (moclo.kits.ecoflex.EcoFlexEntry class method), 44

structure() (moclo.kits.ig.IGCassetteVector class method), 53

structure() (moclo.kits.ig.IGEntry class method), 51

structure() (moclo.kits.ig.IGEntryVector class method), 50

structure() (moclo.kits.ig.IGLevelMEndLinker class method), 54

structure() (moclo.kits.ig.IGLevelMVector class method), 54

structure() (moclo.kits.ig.IGLevelPEndLinker class method), 55

structure() (moclo.kits.ig.IGLevelPVector class method), 55

structure() (moclo.kits.ig.IGProduct class method), 49

structure() (moclo.kits.ytk.YTKCassetteVector class method), 61

structure() (moclo.kits.ytk.YTKEntry class method), 59

structure() (moclo.kits.ytk.YTKEntryVector class method), 58

structure() (moclo.kits.ytk.YTKProduct static method), 57

T

target_sequence() (moclo.core.modules.AbstractModule method), 23

target_sequence() (moclo.core.vectors.AbstractVector method), 25

target_sequence() (moclo.kits.cidar.CIDARCassetteVector method), 42

target_sequence() (moclo.kits.cidar.CIDAREntry method), 40

target_sequence() (moclo.kits.cidar.CIDAREntryVector method), 39

target_sequence() (moclo.kits.cidar.CIDARProduct method), 38

target_sequence() (moclo.kits.ecoflex.EcoFlexCassetteVector method), 46

target_sequence() (moclo.kits.ecoflex.EcoFlexEntry method), 45

target_sequence() (moclo.kits.ig.IGCassetteVector method), 53

target_sequence() (moclo.kits.ig.IGEntry method), 51

target_sequence() (moclo.kits.ig.IGEntryVector method), 50

target_sequence() (moclo.kits.ig.IGProduct method), 49

target_sequence() (moclo.kits.ytk.YTKCassetteVector method), 61

target_sequence() (moclo.kits.ytk.YTKEntry method), 60

target_sequence() (moclo.kits.ytk.YTKEntryVector method), 59

target_sequence() (moclo.kits.ytk.YTKProduct method), 57

U

UnusedModules (*class in moclo.errors*), 27

Y

YTKCassette (*class in moclo.kits.ytk*), 64
YTKCassetteVector (*class in moclo.kits.ytk*), 60
YTKDeviceVector (*class in moclo.kits.ytk*), 65
YTKEntry (*class in moclo.kits.ytk*), 59
YTKEntryVector (*class in moclo.kits.ytk*), 57
YTKPart1 (*class in moclo.kits.ytk*), 61
YTKPart2 (*class in moclo.kits.ytk*), 62
YTKPart234 (*class in moclo.kits.ytk*), 64
YTKPart234r (*class in moclo.kits.ytk*), 64
YTKPart3 (*class in moclo.kits.ytk*), 62
YTKPart3a (*class in moclo.kits.ytk*), 62
YTKPart3b (*class in moclo.kits.ytk*), 62
YTKPart4 (*class in moclo.kits.ytk*), 62
YTKPart4a (*class in moclo.kits.ytk*), 62
YTKPart4b (*class in moclo.kits.ytk*), 63
YTKPart5 (*class in moclo.kits.ytk*), 63
YTKPart6 (*class in moclo.kits.ytk*), 63
YTKPart678 (*class in moclo.kits.ytk*), 64
YTKPart7 (*class in moclo.kits.ytk*), 63
YTKPart8 (*class in moclo.kits.ytk*), 63
YTKPart8a (*class in moclo.kits.ytk*), 64
YTKPart8b (*class in moclo.kits.ytk*), 64
YTKProduct (*class in moclo.kits.ytk*), 56