nature portfolio

https://doi.org/10.1038/s41596-021-00675-2

## Supplementary information

# Genetic circuit design automation with Cello 2.0

In the format provided by the authors and unedited

# Supplementary Manual for ‘Genetic Circuit Design Automation with Cello 2.0’

Timothy S. Jones[1,2], Samuel M.D. Oliveira[1,2], Chris J. Myers[3], Christopher A. Voigt[4], and Douglas Densmore[1,2,†]

[1]Biological Design Center, Boston University, Boston MA, USA

[2]Department of Electrical and Computer Engineering, Boston University, Boston, MA, USA

[3]Electrical, Computer & Energy Engineering, University of Colorado Boulder, Boulder, CO, USA

[4]Synthetic Biology Center, Department of Biological Engineering, Massachusetts Institute of Technology, Cambridge, MA, USA

[†]Corresponding author: dougd@bu.edu

# Table of Contents

# 1 Software modules

The Cello 2.0 code base is divided into three separate projects: a Spring Boot web application, an Angular 9 GUI for that application, and a command line tool that executes the design process. These three projects exist in three separate repositories:

- Web application: https://github.com/CIDARLAB/Cello-v2-webapp
- Angular GUI: https://github.com/CIDARLAB/Cello-v2-webapp-gui
- Core command line tool: https://github.com/CIDARLAB/Cello-v2

A Docker container exists that can be used to serve a complete instance of the web application, available at: https://hub.docker.com/r/cidarlab/cello-webapp. This includes the GUI and the core command line tool, everything required to serve a private instance of Cello 2.0. A Docker container with only the core command line tool is also available: https://hub.docker.com/r/cidarlab/cello-dnacompiler.

# 2 User constraints file format

This section details the user constraints file (UCF), a JavaScript Object Notation (JSON) file that describes a part and gate library for a particular organism. The UCF format used in Cello 2.0 is very similar to the format used in Cello 1.0[1]. Note that the input sensor and output device files use the same basic JSON format but contain different collections.

All of the currently available UCFs and input sensor files and output device files are available at https://github.com/CIDARLAB/Cello-UCF. All the remaining information in this section is encoded in JSON Schema files in the same repository.

Note that the sections marked as "Unmodified from Cello 1.0" provide explanations of the same or similar UCF sections that are already documented by Nielsen et al. in the Supplementary Materials section of the Cello 1.0 publication.[1] The UCF, input, and output files designed for the case study are based on the library of gates presented in Voigt et al.[2] Also note that, except where noted, the UCF fragments appearing in each section below are taken from the Eco1C2G2T2 UCF.[3]

The following sections describe the collections of the UCF along with some examples written in JSON format (Supplementary Listings 1-14).

## 2.1 header

(Unmodified from Cello 1.0)

The header section of the UCF contains information on versioning, authorship, and descriptions of the target organism and its operating conditions.


```
{
   "collection": "header",
   "description": "TetR homologs: PhlF, SrpR, BM3R1, HlyIIR, BetI, AmtR, AmeR, QacR, IcaRA",
   "version": "Eco1C2G2T2",
   "date": "Mon Jun 17 14:13:20 EDT 2019",
   "author": [
      "Jonghyeon Shin",
      "Shuyi Zhang",
      "Timothy Jones"
   ],
   "organism": "Escherichia coli NEB 10-beta",
   "genome": "NEB 10 Δ(ara-leu) 7697 araD139 fhuA ΔlacX74 galK16 galE15 e14- φ80dlacZΔM15  recA1 relA1 endA1 nupG  rpsL (StrR) rph spoT1 Δ(mrr-hsdRMS-mcrBC)",
   "media": "M9 minimal media composed of M9 media salts (6.78 g/L Na2HPO4, 3 g/L KH2PO4, 1 g/L NH4Cl, 0.5 g/L NaCl, 0.34 g/L thiamine hydrochloride, 0.4% D-glucose, 0.2% Casamino acids, 2 mM MgSO4, and 0.1 mM CaCl2; kanamycin (50 µg/ml), spectinomycin (50 µg/ml)",
   "temperature": "37",
   "growth": "Inoculation: Individual colonies into M9 media, 16 hours overnight in plate shaker.  Dilution: Next day, cells dilute ~200-fold into M9 media with antibiotics, growth for 3 hours.  Induction: Cells diluted ~650-fold into M9 media with antibiotics.  Growth: shaking incubator for 5 hours.  Arrest protein production: PBS and 2 mg/ml kanamycin.  Measurement: flow cytometry, data processing for RPU normalization."
}
```


**Supplementary Listing 1**: Example header section of a UCF.

**description**: A prose description of the UCF.

**version**: The iteration of the UCF.

Example:

```
"version": "Eco1C2G2T2"
```

Required: yes

Allowed values: Any string, though a standard naming convention is encouraged.

| | |
|---|---|
| <organism id> | Organism identifier, e.g., Eco for *Eschericia coli* |
| <number> | Strain identifier, counting from 1 |
| C<number> | Experimental conditions identifier |
| G<number> | Genetic gates and insertion location identifier |
| T<number> | Technology mapping and motifs identifier |

**date**: The date the UCF was authored.

Required: no

Allowed values: Any string.

**author**: A list of authors of the UCF.

Required: no

Allowed values: An array of strings.

**organism**: Defines the organism, species, and strain for which the circuits are compiled.

Example:

**"organism"**: "Escherichia coli NEB 10-beta"

Required: yes

Allowed values: Any string.

**genome**: The genotype of the target organism.

Example:

**"genome"**: "NEB 10 Δ(ara-leu) 7697 araD139 fhuA ΔlacX74 galK16 galE15 e14- φ80dlacZΔM15 recA1 relA1 endA1 nupG rpsL (StrR) rph spoT1 Δ(mrr-hsdRMS-mcrBC)"

Required: yes

Allowed values: Any string.

**media**: The media in which the cells are expected to grow.

Example:

**"media"**: "M9 minimal media composed of M9 media salts (6.78 g/L Na2HPO4, 3 g/L KH2PO4, 1 g/L NH4Cl, 0.5 g/L NaCl, 0.34 g/L thiamine hydrochloride, 0.4% D-glucose, 0.2% Casamino acids, 2 mM MgSO4, and 0.1 mM CaCl2; kanamycin (50 µg/ml), spectinomycin (50 µg/ml)"

Required: yes

Allowed values: Any string.

**temperature**: The temperature at which the circuits are expected to perform.

Example:

**"temperature"**: "37"

Required: yes

Allowed values: Any string.

**growth**: The conditions for cell growth and measurement.

Example:

**"growth"**: "Inoculation: Individual colonies into M9 media, 16 hours overnight in plate shaker. Dilution: Next day, cells dilute ~200-fold into M9 media with antibiotics, growth for 3 hours. Induction: Cells diluted ~650-fold into M9 media with antibiotics. Growth: shaking incubator for 5 hours. Arrest protein production: PBS and 2 mg/ml kanamycin. Measurement: flow cytometry, data processing for RPU normalization."

Required: yes

Allowed values: Any string.

## 2.2 measurement_std

(Unmodified from Cello 1.0)

A description of the measurement and data normalization standard used to map fluorescent output to a common unit.

```
{
  "collection": "measurement_std",
  "signal_carrier_units": "RPU",
  "normalization_instructions": "The following equation converts the median YFP fluorescence to RPU. RPU = (YFP – YFP0)/(YFPRPU – YFP0), where YFP is the median fluorescence of the cells of interest, YFP0 is the median autofluorescence, and YFPRPU is the median fluorescence of the cells containing the measurement standard plasmid",
  "plasmid_description": "p15A plasmid backbone with kanamycin resistance and a YFP expression cassette. Upstream insulation by terminator L3S3P21
and a 5'-promoter spacer. Promoter BBa_J23101, ribozyme RiboJ, RBS BBa_B0064 drives constitutive YFP expression, with transcriptional termination by L3S3P21.",
  "plasmid_sequence": [
    "<Lines of the GenBank file (not shown) for the measurement standard plasmid.>"
  ]
}
```

**Supplementary Listing 2**: Example measurement standard section of UCF.

**signal_carrier_units**: The unit of the signal carrier for all response functions.

Example:

"signal_carrier_units": "RPU"

Required: yes

Allowed values: Any string.

**normalization_instructions**: A description of how measured data are normalized using the measurement standard.

Example:

"normalization_instructions": "The following equation converts the median YFP fluorescence to RPU. RPU = (YFP – YFP0)/(YFPRPU – YFP0), where YFP is the median fluorescence of the cells of interest, YFP0 is the median autofluorescence, and YFPRPU is the median fluorescence of the cells containing the measurement standard plasmid"

Required: yes

Allowed values: Any string.

**plasmid_description**: A structural description of the plasmid used to normalize data.

Example:

"plasmid_description": "p15A plasmid backbone with kanamycin resistance and a YFP expression cassette. Upstream insulation by terminator L3S3P21 and a 5'-promoter spacer. Promoter BBa_J23101, ribozyme RiboJ, RBS BBa_B0064 drives constitutive YFP expression, with transcriptional termination by L3S3P21."

Required: yes

Allowed values: Any string.

**plasmid_sequence**: The complete sequence of the measurement standard.

Required: yes

Allowed values: An array of strings. GenBank format is encouraged. Use one JSON string in the UCF per line of the GenBank file.

## 2.3 logic_constraints

(Unmodified from Cello 1.0)

Constraints on the allowed gate types in the circuit design.

```
{
  "collection": "logic_constraints",
  "available_gates": [
    {
      "type": "NOR",
      "max_instances": 9
    }
  ]
}
```

**Supplementary Listing 3**: Example header section of UCF.

**available_gates**: The allowed Boolean gate types and the maximum allowed instances of each.
Required: yes
Allowed values: An array of JSON objects. See below.

**type**: A Boolean gate type to be allowed in circuit designs.
Example:
"type": "NOR"
Required: yes
Allowed values: Any string. Should correspond to available gates in the gates section of the UCF. If NOR is included, NOT may be omitted.

**max_instances**: The maximum allowed instances of the associated gate type.
Example:
"max_instances": 9
Required: yes
Allowed values: Any number.

## 2.4 motif_library

(Unmodified from Cello 1.0)

A section that allows the library designer to specify logic motifs that can be interchanged with logically equivalent subcircuits. Each JSON object with this collection type defines one motif.

```
{
  "collection": "motif_library",
  "outputs": [ "y" ],
  "inputs": [ "a", "b", "c" ],
  "netlist": [
    "NOT(0Wire44,b)",
```

```
    "NOT(0Wire45,c)",
    "NOR(n4,0Wire44,0Wire45)",
    "NOT(0Wire46,a)",
    "NOT(0Wire47,n4)",
    "NOR(y,0Wire46,0Wire47)"
  ]
}
```

---

**Supplementary Listing 4**: An example logic motif section of the UCF.

**outputs**: The list of outputs for the netlist specified in this motif.

Example:

```
"outputs": [ "y" ]
```

Required: yes

Allowed values: An array of strings. Each string should appear in the netlist.

**inputs**: The list of inputs for the netlist specified in this motif.

Example:

```
"inputs": [ "a", "b", "c" ]
```

Required: yes

Allowed values: An array of strings. Each string should appear in the netlist.

**netlist**: The list of gates specifying the subcircuit to be swapped. Given in a structural Verilog format.

Example: A combination of 2-input NOT and NOR gates:

```
"netlist": [
  "NOT(0Wire44,b)",
  "NOT(0Wire45,c)",
  "NOR(n4,0Wire44,0Wire45)",
  "NOT(0Wire46,a)",
  "NOT(0Wire47,n4)",
  "NOR(y,0Wire46,0Wire47)"
]
```

Example: A three-input NOR gate:

```
"netlist": [
  "NOR(y,a,b,c)"
]
```

Example: A three-input AND gate:

```
"netlist": [
  "AND(y,a,b,c)"
]
```

Required: yes

Allowed values: Any string. The gate type can be one of: NOT, NOR, AND, OR, OUTPUT_OR, NAND, XOR, XNOR. The OUTPUT_OR is not a gate but a motif, indicating that multiple inputs to an output gate may simply be summed. The types of gates implemented in Cello 2.0 libraries are all inverting (NOT and NOR), but the final output gate does not repress a downstream promoter.

## 2.5 genetic_locations

(Modified from Cello 1.0)

A specification of the locations available for circuit integration. Integration sites can be either on plasmids or in a genome.

---


```
{
  "collection": "genetic_locations",
  "locations": [
    {
      "symbol": "L1",
      "locus": {
        "start": 2343,
        "end": 2343
      },
      "sequence": {
        "type": "file",
        "ref": "JS_BB_2"
      }
    },
    {
      "symbol": "L2",
      "locus": {
        "start": 1876,
        "end": 2301
      },
      "unit_conversion": 1.0,
      "sequence": {
        "type": "file",
        "ref": "JS_BB_2"
      }
    },
    {
      "symbol": "L3",
      "locus": {
        "start": 34,
        "end": 459
      },
      "unit_conversion": 1.0,
      "sequence": {
        "type": "file",
        "ref": "JS_BB_3"
      }
    }
  ],
  "sequences": [
    {
      "name": "JS_BB_2",
      "type": "file",
      "data": [
            "<Lines of the GenBank file (not shown) for the plasmid.>"
          ]
    },
```

```
  {
    "name": "JS_BB_3",
    "type": "file",
    "data": [
          "<Lines of the GenBank file (not shown) for the plasmid.>"
    ]
  }
 ]
}
```

**Supplementary Listing 5**: Example header section of UCF.

**locations**: Specifications of the locations available for integration.

Required: yes

Allowed values: An array of JSON objects, see below.

**symbol**: The name of the insert location. It can be used in Eugene placement rules (see the circuit_rules section below).

Example:

```
"symbol": "L1"
```

Required: yes

Allowed values: Any string.

**locus**: A specification of the base pair offsets of the start and end of the integration site.

Required: yes

Allowed values: See below.

**start**: The base pair offset in the sequence associated with this location corresponding to the start of the integration site.

Example:

```
"start": 1876
```

Required: yes

Allowed values: Any positive integer.

**end**: The base pair offset in the sequence associated with this location corresponding to the end of the integration site.

Example:

```
"end": 2301
```

Required: yes

Allowed values: Any positive integer.

**sequence**: A specification of the sequence in which this integration site exists.

Required: yes

Allowed values: Any string.

**type**: The type of the sequence reference, i.e., whether the sequence is specified as a GenBank file or as a pointer to an NCBI entry.

Example:

**"type"**: "ncbi"

Example:

**"type"**: "file"

Required: yes

Allowed values: "file" | "ncbi"

**ref**: Either a reference to a file defined in the sequences section, or an NCBI identifier.

Example:

**"ref"**: "U0096.3"

Example:

**"ref"**: "my_sequence_name"

Required: yes

Allowed values: Any string. Either a reference to an NCBI sequence or a reference to a file.

**sequences**: Specifications of the sequences (plasmids or host genomes) in which the integration sites lie. Only required if any of the locations specified use file in the *type* field.

Required: no

Allowed values: An array of JSON objects, see below.

**name**: The name of the sequence.

Example:

**"name"**: "JS_BB_2"

Required: yes

Allowed values: Any string.

**type**: The type of this sequence.

Example:

**"type"**: "file"

Required: yes

Allowed values: "file"

**data**: The DNA sequence.

Required: yes

Allowed values: An array of strings, one string per line of a GenBank file.

## 2.6 gates

(Modified from Cello 1.0)


```
{
  "collection": "gates",
  "name": "P3_PhlF",
  "system": "TetR",
  "group": "PhlF",
  "regulator": "PhlF",
  "gate_type": "NOR",
  "color": "F9A427",
  "model": "P3_PhlF_model",
  "structure": "P3_PhlF_structure"
}
```


**Supplementary Listing 6**: Example header section of UCF.

**name**: The name of the gate.

Example:

"name": "P3_PhlF"

Required: yes

Allowed values: Any string.

**system**: The biochemistry of the gate.

Example:

"system": "TetR"

Example:

"system": "TetR"

Required: yes

Allowed values: Any string.

**group**: Used to group a set of gate variants, only one of which can be used in a circuit to preserve signal orthogonality. For example, if P1_PhlF and P2_PhlF produce an identical protein, only one should appear in a single cell, and thus both should be assigned the same group property.

Example:

"group": "PhlF"

Required: yes

Allowed values: Any string.

**regulator**: The regulator of this gate.

Example:

"regulator": "PhlF"

Required: yes

Allowed values: Any string.

**gate_type**: The Boolean operation this gate performs. A NOR gate can implement a NOT gate.

Example:

"gate_type": "NOR"

Required: yes

Allowed values: A string. The gate type can be one of: NOR, AND, OR, OUTPUT_OR, NAND, XOR, XNOR.

**color**: The color to shade the gate in output plots.

Example:

"color": "F9A427"

Required: yes

Allowed values: An RGB hexcode. Should match the regular expression [A-Fa-f0-9]{6}.

**model**: The name of the model definition of this gate.

Example:

"model": "P3_PhlF_model"

Required: yes

Allowed values: Any string. Should correspond to the name of a model appearing in the models section.

**structure**: The name of the structure definition of this gate.

Example:

"structure": "P3_PhlF_structure"

Required: yes

Allowed values: Any string.

## 2.7 models

(New in Cello 2.0)

---


```
{
  "collection": "models",
  "name": "P3_PhlF_model",
  "functions": {
    "response_function": "Hill_response",
    "input_composition": "tandem_input_composition",
    "tandem_interference_factor": "tandem_interference_factor",
    "toxicity": "P3_PhlF_toxicity",
    "cytometry": "P3_PhlF_cytometry"
  },
  "parameters": [
    {
      "name": "ymax",
      "value": 7.088098911,
      "description": "Maximal transcription"
    },
    {
      "name": "ymin",
      "value": 0.004656815,
      "description": "Minimal transcription"
```

```
    },
    {
      "name": "K",
      "value": 0.116044457,
      "description": "Half-maximum"
    },
    {
      "name": "n",
      "value": 3.342819362,
      "description": "Cooperativety"
    },
    {
      "name": "alpha",
      "value": 0.240700115,
      "description": "Tandem factor"
    },
    {
      "name": "beta",
      "value": 0.061908386,
      "description": "Tandem factor"
    }
  ]
}
```


**Supplementary Listing 7**: Example header section of UCF.

**name**: The name of this model.

Example:

Required: yes

Allowed values: Any string.

**functions**: A dictionary of functions used by this gate. An entry in the dictionary with key response_function is required. Keys cytometry and toxicity have special significance: if either is present in all gates in the UCF, this will cause Cello to compute the predicted cytometry profile or predicted optical density of the circuit.

Example:


```
"functions": {
  "response_function": "Hill_response",
  "input_composition": "tandem_input_composition",
  "tandem_interference_factor": "tandem_interference_factor",
  "toxicity": "P3_PhlF_toxicity",
  "cytometry": "P3_PhlF_cytometry"
}
```


Required: yes

Allowed values: A dictionary. Any string is allowed for the key, with response_function being required and cytometry and toxicity having special meaning.

**parameters**: The numerical parameters that may be referenced in any function specified in this gate model.

Required: yes

Allowed values: An array of JSON objects, see below.

**name**: The name of the parameter.
Example:
**"name"**: "n"
Required: yes
Allowed values: Any string.

**value**: The numerical value of the parameter.
Example:
**"value"**: 3.342819362
Required: yes
Allowed values: Any number.

**description**: A description of the parameter.
Example:
**"description"**: "Cooperativety"
Required: no
Allowed values: Any string.

## 2.8 functions

(New in Cello 2.0)

---


```
{
  "collection": "functions",
  "name": "Hill_response",
  "equation": "ymin + (ymax - ymin) / (1.0 + (x / K)^n)",
  "variables": [
    {
      "name": "x",
      "map": "#//model/functions/input_composition"
    }
  ],
  "parameters": [
    {
      "name": "ymax",
      "map": "#//model/parameters/ymax"
    },
    {
      "name": "ymin",
      "map": "#//model/parameters/ymin"
    },
    {
      "name": "K",
      "map": "#//model/parameters/K"
    },
    {
      "name": "n",
      "map": "#//model/parameters/n"
    }
```

```
  ]
},
{
  "collection": "functions",
  "name": "tandem_input_composition",
  "equation": "t1 * x2 + x1",
  "variables": [
    {
      "name": "x1",
      "map": "#//structure/inputs/in1/model/functions/response_function"
    },
    {
      "name": "x2",
      "map": "#//structure/inputs/in2/model/functions/response_function"
    },
    {
      "name": "t1",
      "map": "#//structure/inputs/in1/model/functions/tandem_interference_factor"
    }
  ]
},
{
  "collection": "functions",
  "name": "tandem_interference_factor",
  "equation": "alpha * (K^n + beta * x^n) / (K^n + x^n)",
  "variables": [
    {
      "name": "x",
      "map": "#//model/functions/input_composition"
    }
  ],
  "parameters": [
    {
      "name": "alpha",
      "map": "#//model/parameters/alpha"
    },
    {
      "name": "beta",
      "map": "#//model/parameters/beta"
    },
    {
      "name": "K",
      "map": "#//model/parameters/K"
    },
    {
      "name": "n",
      "map": "#//model/parameters/n"
    }
  ]
}
```

**Supplementary Listing 8**: Example header section of UCF.

**name**: The name of the function.

Example:

Required: yes
Allowed values: Any string.

**equation**: The equation of this function.
Example:
"equation": "ymin + (ymax - ymin) / (1.0 + (x / K)^n)"
Required: yes
Allowed values: Any valid equation. Variables or parameters can be contiguous strings of characters matching regular expression [A-Za-z_]+, and should correspond to a named parameter defined in the parameters section or a named variable defined in the variables section.

**variables**: The independent variables appearing in the equation.
Example:
Required: yes
Allowed values: An array of JSON objects, see below.

**name**: The name of the variable.
Example:
"name": "x"
Required: yes
Allowed values: Any string matching regular expression [A-Za-z_]+.

**map**: A pointer to an equation.
Example:
"map": "#//model/functions/input_composition"
Example:
"map": "#//structure/inputs/in1/model/functions/response_function"

Required: yes
Allowed values: A string in a syntax similar to JSON Pointer. The two characters should be #/, followed by a concatenation of object references beginning with a forward slash /. The root context #/ is of the gate. From the gate, objects in the gate's model or structure can be referenced. Within a model, functions can be referenced. Within a structure, inputs can be referenced by name, leading the evaluation context to the gate assigned to that input. Once the evaluation context is in another gate, the process can continue to reference the model or structure of that gate.

**parameters**: The fixed parameters appearing in the equation.
Example:
Required: yes
Allowed values: Any string.

**name**: The parameter name.
Example:
"name": "ymax"
Required: yes
Allowed values: Any string.

**map**: A pointer to a numerical parameter defined on a gate's model.
Example:
"map": "#//model/parameters/ymax"
Required: yes
Allowed values: A string in a syntax similar to JSON Pointer. See the description above in the variables section.

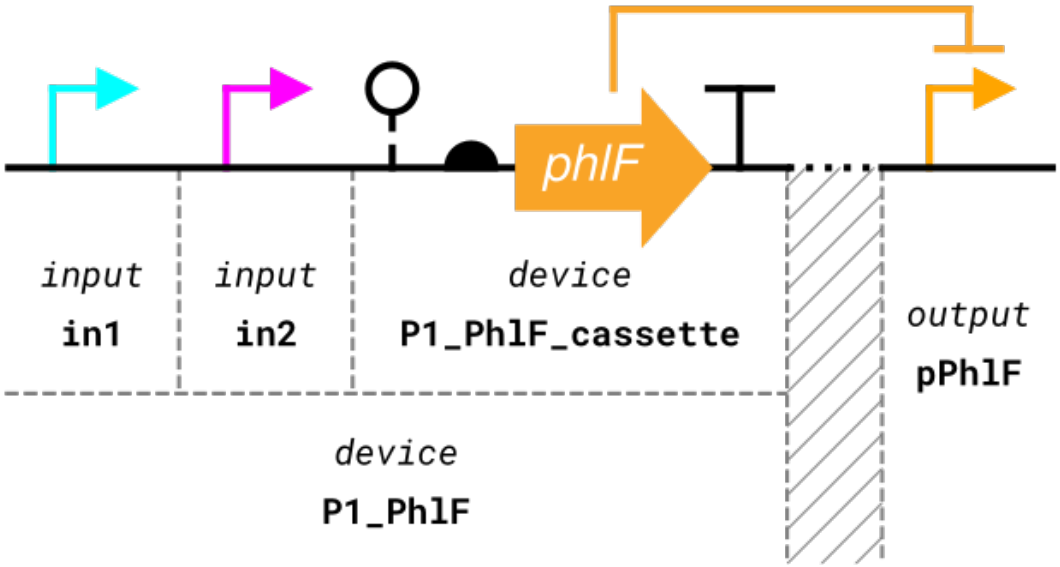
## 2.9 structures

(New in Cello 2.0)

This section describes the new structures field of the UCF, the schematics of different promoter structures that are compatible with Cello 2.0 (Supplementary Figures 1-3) and examples from multiple UCFs encoding different structures are given (Supplementary Listings 9-11).

### 2.9.1 Tandem promoters

This section gives an encoding of a gate with tandem promoters from the Eco1C2G2T2 UCF.




**Supplementary Figure 1:** A NOR gate with tandem promoters and its encoding in the UCF. The JSON encoding is given in Supplementary Listing 9.

---


```
{
  "collection": "structures",
  "name": "P3_PhlF_structure",
```

```
 "inputs": [
  {
   "name": "in1",
   "part_type": "promoter"
  },
  {
   "name": "in2",
   "part_type": "promoter"
  }
 ],
 "outputs": [
  "pPhlF"
 ],
 "devices": [
  {
   "name": "P3_PhlF",
   "components": [
    "#in2",
    "#in1",
    "P3_PhlF_cassette"
   ]
  },
  {
   "name": "P3_PhlF_cassette",
   "components": [
    "RiboJ53",
    "P3",
    "PhlF",
    "ECK120033737"
   ]
  }
 ]
}
```


**Supplementary Listing 9**: An example entry in the structures collection of the Eco1C2G2T2 UCF. This entry encodes a gate with tandem promoters.

**name**: The name of this structure object.

Example:

"name": "P3_PhlF_structure"

Required: yes

Allowed values: Any string.

**inputs**: A set of JSON objects specifying the parts that act as inputs to this gate.

Required: yes

Allowed values: An array of JSON objects, see below.

**name**: The name of the input part. This name is available for use in the devices section of the structure object.

Example:

"name": "in1"

Required: yes
Allowed values: Any string.

**part_type**:
Example:
**"part_type"**: "promoter"
Required: yes
Allowed values: Any of the allowed values for the type field of the parts collection below. The value of this field should likely always be "promoter".

**outputs**: A list of the parts that carry the output signal from this gate. Each part, if instantiated in the circuit, will carry the same output signal.
Example:
**"outputs"**: [ "pPhlF" ]
Required: yes
Allowed values: An array of strings. Each string should correspond to the name of a part in the parts section of the UCF.

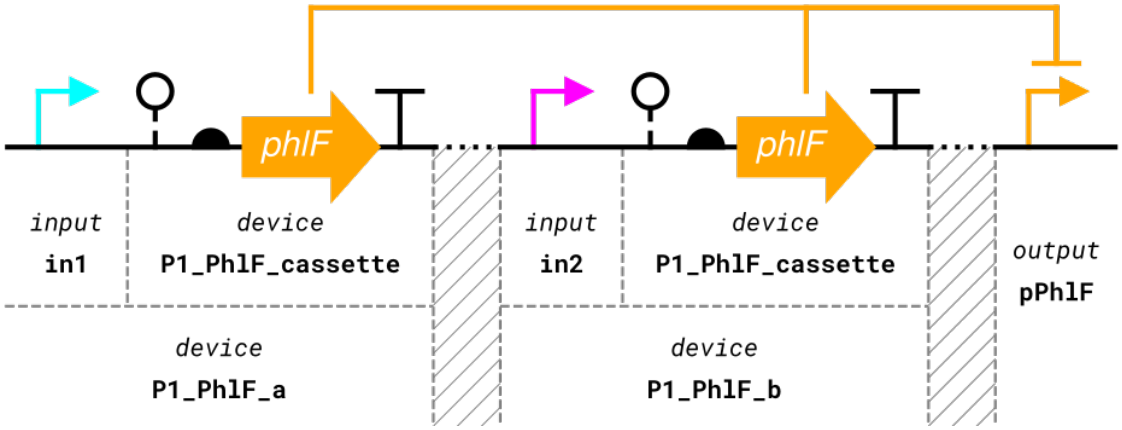
**devices**: The specifications for the structures that make up the gate. Each device is a named list of parts, and devices can be nested.
Required: yes
Allowed values: An array of JSON objects, see below.

**name**: The name of the device.
Example:
**"name"**: "P3_PhlF"
Required: yes
Allowed values: Any string.

**components**: The list of discrete parts that make up the gate.
Example:
**"components"**: [
"#in2",
"#in1",
"P3_PhlF_cassette"
]
Required: yes
Allowed values: An array of strings. To reference one of the inputs defined in this gate structure, prepend the name of the input with a hash (#) sign. Otherwise, valid strings are either other device names, or names of parts defined in the parts collection of the UCF.

### 2.9.2 Split transcriptional units

This section gives an encoding of a gate with split transcriptional units from the Eco2C1G3T1 UCF. See section 2.9.1 for an enumeration of the accepted fields in the structures collection of the UCF.




**Supplementary Figure 2:** A NOR gate split into two transcriptional units and its encoding in the Eco2C1G3T1 UCF. The JSON encoding is given in Listing 10.

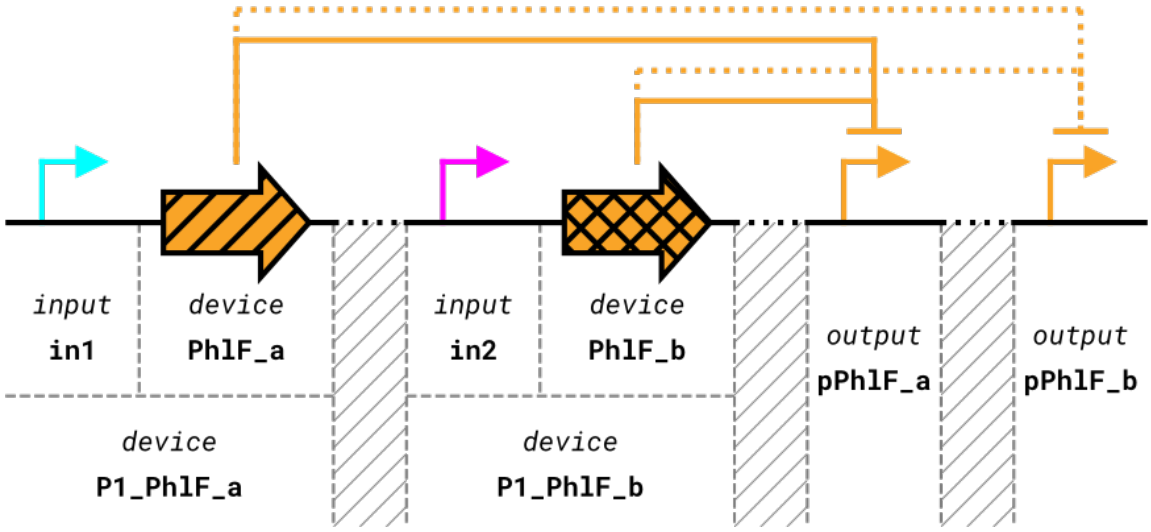
```
{
  "collection": "structures",
  "name": "P1_PhlF_structure",
  "inputs": [
    {
      "name": "in1",
      "part_type": "promoter"
    },
    {
      "name": "in2",
      "part_type": "promoter"
    }
  ],
  "outputs": [
    "pPhlF"
  ],
  "devices": [
    {
      "name": "P1_PhlF_a",
      "components": [
        "#in1",
        "P1_PhlF_a_cassette"
      ]
    },
    {
      "name": "P1_PhlF_a_cassette",
      "components": [
        "RiboJ53",
        "P1",
        "PhlF",
        "DT5"
      ]
    },
```

```
    {
      "name": "P1_PhlF_b",
      "components": [
        "#in2",
        "P1_PhlF_b_cassette"
      ]
    },
    {
      "name": "P1_PhlF_b_cassette",
      "components": [
        "RiboJ53",
        "P1",
        "PhlF",
        "DT5"
      ]
    }
  ]
}
```

**Supplementary Listing 10:** An example entry in the structures collection of the Eco2C1G3T1 UCF. This entry encodes a gate with split transcriptional units.

### 2.9.3 Split transcriptional units with gene variants and multiple outputs

This section gives an encoding of a gate with split transcriptional units from the SC1C1G1T1 UCF. See section 2.9.1 for an enumeration of the accepted fields in the structures collection of the UCF.




**Supplementary Figure 3:** A NOR gate split into two transcriptional units, each with a variant of the gene. The gene variants produce the same transcription factor, which regulates two promoter variants. A visualization of the encoding in the SC1C1G1T1 UCF is drawn underneath the sequence diagram. The exact JSON that encodes the gate is given in Listing 11.

```
{
  "collection": "structures",
  "name": "P1_PhlF_structure",
  "inputs": [
```

```
    {
      "name": "in1",
      "part_type": "promoter"
    },
    {
      "name": "in2",
      "part_type": "promoter"
    }
  ],
  "outputs": [
    "pPhlF1_a",
    "pPhlF1_b"
  ],
  "devices": [
    {
      "name": "P1_PhlF_a",
      "components": [
        "#in1",
        "P1_PhlF_a_cassette"
      ]
    },
    {
      "name": "P1_PhlF_a_cassette",
      "components": [
        "Kozak1",
        "PhlF_a"
      ]
    },
    {
      "name": "P1_PhlF_b",
      "components": [
        "#in2",
        "P1_PhlF_b_cassette"
      ]
    },
    {
      "name": "P1_PhlF_b_cassette",
      "components": [
        "Kozak1",
        "PhlF_b"
      ]
    }
  ]
}
```


**Supplementary Listing 11:** An example entry in the structures collection of the SC1G1T1T1 UCF. This entry encodes a gate with split transcriptional units, with a gene variant in each transcriptional unit, and multiple output promoters.

## 2.10 parts

(Unmodified from Cello 1.0)


```
{
```

```
  "collection": "parts",
  "type": "ribozyme",
  "name": "RiboJ53",
  "dnasequence":
"AGCGGTCAACGCATGTGCTTTGCGTTCTGATGAGACAGTGATGTCGAAACCGCCTCTACAAATAATT
TTGTTTAA"
}
```


**Supplementary Listing 12**: Example header section of UCF.

**type**: The part type.

Example:

"type": "ribozyme"

Required: yes

Allowed values: "promoter" | "ribozyme" | "rbs" | "cds" | "terminator" | "spacer" | "scar"

**name**: The part name.

Example:

"name": "RiboJ53"

Required: yes

Allowed values: Any string.

**dnasequence**: The complete DNA sequence of the part.

Example:

"dnasequence":
"AGCGGTCAACGCATGTGCTTTGCGTTCTGATGAGACAGTGATGTCGAAACC
GCCTCTACAAATAATTTTGTTTAA"

Required: yes

Allowed values: Any contiguous string of characters from the set {A,T,G,C,a,t,g,c}.

## 2.11 device_rules

(New in Cello 2.0)


```
{
  "collection": "device_rules",
  "rules": {
    "function": "AND",
    "rules": [
      "pTac BEFORE pTet",
      "ALL_FORWARD"
    ]
  }
}
```


**Supplementary Listing 13**: An example of the device rules section of a UCF.

## 2.12 circuit_rules

(New in Cello 2.0)

```
{
  "collection": "circuit_rules",
  "rules": {
    "function": "AND",
    "rules": [
      {
      "V1_VanR BEFORE P1_PhlF",
      "ALL_FORWARD"
      },
      {
      "V1_VanR BEFORE P1_PhlF",
      "ALL_FORWARD"
      },
    ]
  }
}
```

**Supplementary Listing 14**: An example of the circuit rules section of a UCF.

# 3 Input sensor file format

This section describes the input sensor file, a JSON file that describes a set of characterized input sensors available for use in a circuit design. The input sensor file is written in a similar format as the UCF (a list of JSON objects, each with a type specified by a collection field), but only certain collection types are allowed. One of these, *input_sensors*, is described below. The others are models, functions, structures, and parts. The following sections describe the collections of the input sensor file along with some examples written in JSON format (Supplementary Listings 15-18).

## 3.1 input_sensors

(New in Cello 2.0)

```
{
  "collection": "input_sensors",
  "name": "LacI_sensor",
  "model": "LacI_sensor_model",
  "structure": "LacI_sensor_structure"
}
```

**Supplementary Listing 15**: Example input sensors section of the input sensor file.

**name**: The name of the input sensor.

Example:

```
"name": "LacI_sensor"
```

Required: yes
Allowed values: Any string.

**model**: The name of the model associated with the input sensor. See the §2.7 on models.
Example:
"model": "LacI_sensor_model"
Required: yes
Allowed values: Any string. Should correspond to the name of an object defined in the models collection.

**structure**: The name of the model associated with the input sensor. See the §2.9 on structures.
Example:
"structure": "LacI_sensor_structure"
Required: yes
Allowed values: Any string. Should correspond to the name of an object defined in the structures collection.

## 3.2 models

(New in Cello 2.0)

Identical to the models collection of the UCF, see section 2.7 for a complete description of all fields. An example entry for an input sensor in this collection is shown below.

---


```
{
  "collection": "models",
  "name": "LacI_sensor_model",
  "functions": {
    "response_function": "sensor_response",
    "tandem_interference_factor": "sensor_tandem_interference_factor"
  },
  "parameters": [
    {
      "name": "ymax",
      "value": 1.686,
      "description": "Maximal transcription"
    },
    {
      "name": "ymin",
      "value": 0.008,
      "description": "Minimal transcription"
    },
    {
      "name": "alpha",
      "value": 0.73,
      "description": "Tandem parameter"
    },
    {
      "name": "beta",
```

```
    "value": 0.04,
    "description": "Tandem parameter"
   }
  ]
}
```

**Supplementary Listing 16:** An example model for the LacI sensor. Sensors still specify a response function

## 3.3 structures

(New in Cello 2.0)

Identical to the structures collection of the UCF, see section 2.9 for a complete description of all fields. An example entry for an input sensor in this collection is shown below.

```
{
 "collection": "structures",
 "name": "LacI_sensor_structure",
 "outputs": [
  "pTac"
 ]
}
```

**Supplementary Listing 17:** An example structure for the LacI sensor. Only one output part is specified.

## 3.4 functions

(New in Cello 2.0)

Identical to the functions collection of the UCF, see section 2.8 for a complete description of all fields. An example entry in this collection in an input sensor file is shown below. The example demonstrates the special symbol $STATE that maps the Boolean output value of the node in which the function is being evaluated to an integer: True ↦ 1, False ↦ 0.

```
{
 "collection": "functions",
 "name": "sensor_tandem_interference_factor",
 "equation": "alpha * beta^(1 - $STATE)",
 "parameters": [
  {
   "name": "alpha",
   "map": "#//model/parameters/alpha"
  },
  {
   "name": "beta",
   "map": "#//model/parameters/beta"
  }
```

```
  ]
}
```

**Supplementary Listing 18:** An example function specified in an input sensor file. The equation uses the special symbol $STATE that maps the Boolean output value of the node in which the function is being evaluated to an integer: True ↦ 1, False ↦ 0.

## 3.5 parts

(Unmodified from Cello 1.0)

Identical to the parts collection of the UCF, see section 2.10 for an example and a complete description of all fields.

# 4 Output device file format

This section describes the output device file, a JSON file that describes a set of characterized output actuators available for use in a circuit design. The output device file is written in a similar format as the UCF (a list of JSON objects, each with a type specified by a collection field), but only certain collection types are allowed. One of these, output_devices, is described below. The others are models, functions, structures, and parts. The following sections describe the collections of the output device file along with some examples written in JSON format (Supplementary Listings 19-22).

## 4.1 output_devices

(New in Cello 2.0)

```
{
  "collection": "output_devices",
  "name": "YFP_reporter",
  "model": "YFP_reporter_model",
  "structure": "YFP_reporter_structure"
}
```

**Supplementary Listing 19**: Example header section of UCF.

**name**: The name of the output device.

Example:

"name": "YFP_reporter"

Required: yes

Allowed values: Any string.

**model**: The name of the model describing the behavior of this output device.

Example:

"model": "YFP_reporter_model"

Required: yes
Allowed values: Any string. Should correspond to the name of an object defined in the models collection.

**structure**: The name of the architecture definition of this output device.
Example:
**"structure"**: "YFP_reporter_structure"
Required: yes
Allowed values: Any string. Should correspond to the name of an object defined in the structures collection.

## 4.2 models

(New in Cello 2.0)

Identical to the models collection of the UCF, see section 2.7 for a complete description of all fields. An example entry for an output device in this collection is shown below.

```
{
  "collection": "models",
  "name": "YFP_reporter_model",
  "functions": {
    "response_function": "linear_response",
    "input_composition": "tandem_input_composition"
  },
  "parameters": [
    {
      "name": "unit_conversion",
      "value": 1.0
    }
  ]
}
```

**Supplementary Listing 20:** An example model for the YFP output reporter. Output devices still specify a response function.

## 4.3 structures

(New in Cello 2.0)

Identical to the structures collection of the UCF, see section 2.9 for a complete description of all fields. An example entry for an output device in this collection is shown below.

```
{
  "collection": "structures",
  "name": "YFP_reporter_structure",
  "inputs": [
    {
      "name": "in1",
      "part_type": "promoter"
```

```
    },
    {
      "name": "in2",
      "part_type": "promoter"
    }
  ],
  "devices": [
    {
      "name": "YFP_reporter",
      "components": [
        "#in2",
        "#in1",
        "YFP_cassette"
      ]
    }
  ]
}
```

**Supplementary Listing 21:** An example structure for the YFP output device.

## 4.4 functions

(New in Cello 2.0)

Identical to the functions collection of the UCF, see section 2.8 for a complete description of all fields. An example entry for an output device in this collection is shown below.

```
{
  "collection": "functions",
  "name": "linear_response",
  "equation": "c * x",
  "variables": [
    {
      "name": "x",
      "map": "#//model/functions/input_composition"
        }
  ],
  "parameters": [
    {
      "name": "c",
      "map": "#//model/parameters/unit_conversion"
    }
  ]
}
```

**Supplementary Listing 22:** An example function specified in an output device file. The function simply specifies an output proportional to the input.

## 4.5 parts

(Unmodified from Cello 1.0)

Identical to the parts collection of the UCF, see section 2.10 for an example and a complete description of all fields.

# 5 Verilog

## 5.1 A short guide to preparing a Verilog file

Numerous guides on understanding and writing Verilog files can be found in textbooks or online. Here we briefly explain the preparation of Verilog files in one of two modes commonly used in for genetic circuits, i.e., Boolean logic of very low complexity. Files that use different Verilog syntax are certainly acceptable, here we merely explain very simple templates (Supplementary Listings 23-26).

### 5.1.1 Truth table

As an example, suppose we wish to implement some logic of two inputs *a* and *b* and a single output *y with t*he following truth table:

| *a* | *b* | *y* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The truth table is the XOR function for which there is a built-in Verilog operation, but suppose this is not the case and we wish to implement the truth table directly.

All Verilog files in Cello will begin with a module statement that names the device to be constructed and specifies inputs and outputs. For the above truth table, we should write something like

```
module my_xor(output y, input a, b);
```

This is self-explanatory, indicating a module called "my_xor" with inputs *a* and *b* and output *y*. In the body of the module we write the truth table. For those familiar with basic programming concepts, this is done using a case statement. As Verilog is a language that describes synchronous, clocked logic, it is necessary to wrap the case statement in a block that indicates to what variables (inputs in this case) the block is sensitive. If we follow the most basic template for a truth table implemented in Verilog, simply write the names of the inputs, followed by the begin keyword, as

```
always@(a, b)
  begin
```

Next is the beginning of the case statement. The individual input states are denoted as binary literals, and the first line of the case statement indicates the encoding.

```
    case({a, b})
```

The meaning of this will become clear as we write the rows of the truth table.

```
      2'b00: {y} = 1'b0;
      2'b01: {y} = 1'b1;
      2'b10: {y} = 1'b1;
      2'b11: {y} = 1'b0;
```

We have four cases for the inputs (four lines above) and four rows of the truth table. The first row begins with 2'b00: that indicates the first input state where *a* = 0 and *b* = 0. The prefix 2'b indicates that the following symbols are to be interpreted as 2-bit binary data. We indicated with the statement **case**({a, b}) that the first bit in 2'b00 corresponds to the input *a* and the second bit corresponds to the input *b.* Everything after the colon is executed if the inputs are in the given state. In our truth table construction, we execute an assignment of the output *y* for every input state. In the first row, we have {y} = 1'b0; which simply assigns a 1-bit binary number (0) to the output. The remaining rows of the truth table are encoded in an identical manner.

Finally, we close the case statement, the "always" block, and the module, as

```
    endcase
  end

endmodule
```

See Supplementary Listing 23 for a similar example with three inputs.

## 5.1.2 Boolean function

Another common mode of describing logic is by writing the Boolean function explicitly. For a truth table, we described the input-output relationship as a lookup table. If we write the Boolean function explicitly, we indicate the intermediate logic operations between input and output. Implement an AND gate using the NOT and NOR primitive operations. We know from DeMorgan's laws that

NOT(A AND B) = NOT(A) OR NOT(B)

Applying NOT to both sides, we have

A AND B = NOT(A) NOR NOT(B)

So to implement an AND gate, we invert each input and then apply a NOR operation. We proceed by writing a module definition as in Section 5.1.1.

```
module my_and(output y, input a, b);
```

Next, we need to define two variables to store the outputs of the initial inversion of the inputs *a* and *b*. For simple combinational logic, we will always use the "wire" type.

```
  wire w1, w2;
```

We have now declared two "wire" type variables, and can implement the Boolean function with primitive operations.

```
  not(w1, a);
  not(w2, b);
  nor(y, w1, w2);
```

In these primitive functions, the first argument is always the variable name to which to assign the output of the operation. So w1 receives the inversion of a and w2 the inversion of b. Finally, we apply a NOR operation to the intermediate wires and then assign the result to y. We then close the module block as before.

```
endmodule
```

We implemented an AND operation with the NOT and NOR primitives. Though the libraries that are included with Cello use NOT and NOR primitives exclusively, Verilog includes an and function that we could have used in this example. Cello would still understand to synthesize the AND operation using the NOT and NOR biological primitives such that the resulting circuit would use three gates, just as we wrote the Verilog above.

## 5.2 Sample files

This section includes a few sample Verilog files that illustrated different modes of Verilog input that are accepted by Cello.

```
module x01(output out, input a, b, c);

  always@(c, b, a)
    begin
      case({c, b, a})
        3'b000: {out} = 1'b0;
        3'b001: {out} = 1'b0;
        3'b010: {out} = 1'b0;
        3'b011: {out} = 1'b0;
        3'b100: {out} = 1'b0;
        3'b101: {out} = 1'b0;
        3'b110: {out} = 1'b0;
        3'b111: {out} = 1'b1;
      endcase
    end

endmodule
```

**Supplementary Listing 23**: Verilog for the 0x01 circuit. The Verilog file uses a case statement to encode the complete truth table.

```
module x01(output out, input a, b, c);
```

```
  and(out, a, b, c);

endmodule
```

**Supplementary Listing 24:** An alternate Verilog specification of the 0x01 circuit. The case statement encoding of the truth table has been condensed into a purely structural form, a single AND operation.

```
// N.B. - Place module definitions above those in which they are
// instantiated. Module `mysub` is instantiated in module `main`, so
// `mysub` is defined before `main`.

// SUB module
module mysub(output out, input a, b);

  and(out, a, b);

endmodule

// MAIN module
module main(output out, input a, b, c, d);

  wire w1, w2;

  mysub s1(w1, a, b);
  mysub s2(w2, c, d);
  and(out, w1, w2);

endmodule
```

**Supplementary Listing 25:** Example usage of submodules in Verilog. The module *mysub* is instantiated twice in the module main. Note that a module definition should appear before the module(s) in which it is instantiated, so in the listing, module *mysub* appears before main.

```
module struct(output x, input a, b, c);

  wire w1, w2, w3, w4, w5;
  not (w1, c);
  not (w5, b);
  not (w4, a);
  nor (w3, w4, w5);
  not (w2, w3);
  nor (x, w1, w2);

endmodule
```

**Supplementary Listing 26:** An example of a circuit specification written in purely structural Verilog.

# 6 Case study: Library preparation

In this section we consider the design of the SC1C1G1T1 UCF library of NOT and 2-input NOR gates for *S. cerevisiae* developed by Voigt et al.[2] We choose to demonstrate the preparation of the UCF using the *S. cerevisiae* gates as this system illustrates most of the new features of the UCF format available for the description of new and idiosyncratic gate libraries.

In more detail, the *S. cerevisiae* gates are structurally different from the first *E. coli* gates that shipped with Cello. In the *E. coli* system, the simplest gate is a NOT gate, composed of one promoter followed by a gene expression cassette. A two-input NOR gate in the *E. coli* system is two promoters in series formation, i.e., tandem promoters, followed by an expression cassette. The expression cassette produces a transcription factor that regulates a single promoter (an input to a gate elsewhere in the circuit). The gates can go anywhere in their target location; for *E. coli*, the target is a plasmid. In the *S. cerevisiae* system, a two-input NOR gate in yeast is composed of two gene expression cassettes with one input promoter each. Both genes in the cassette produce the same transcription factor, but the genes themselves are sequence variants of one another. Furthermore, the output transcription factor regulates both of two available sequence-variant promoters. Finally, each transcriptional unit is placed in the host genome and each placement is constrained by rules different than those for plasmids. Examples of both strategies are shown in Figure 2 in the main manuscript and an example of genome-integrated promoters in Supplementary Figure 2.

## 6.1 Preparation of Cello Input-files

In the previous sections, we described the formats for designing the three independent input files required for a Cello 2.0 run: (i) UCF library (described in §2), (ii) input-sensors file (see §3), and (iii) output-device file (see §4). This section describes the steps involved in designing such input-files from scratch, i.e., whether users cannot make use of the existing UCFs and input sensor files and output device files currently available at https://github.com/CIDARLAB/Cello-UCF.

In general, the files' preparation starts with the decision for gates, sensors, outputs, strains (i.e., chassis), and experimental conditions to be used for genetic circuit design. Namely, it first starts with collecting empirical metrics for each transcriptional unit (e.g., a NOT gate or an Input-sensor) and output signal (e.g., YFP fluorescence signal) in each experimental condition. These metrics, such as cell growth optical density and flow-cytometer fluorescence measurements, are usually obtained from their experiments. If characterized gates generate low cell density for toxicity measurements or poor dynamic range for gene activity measurements, we recommend users define a criterion for selecting the optimal gates to be included in the input files. Find more information about customizing gates and models in the section' Custom gate models' and experimental timings in 'General Procedures' in the main Manuscript.

Then, while the UCF libraries should contain metrics for multiple gates and the input-sensor files should contain characterized data for sensors, output-device files should have characterized data for actuators or signals used as final outputs of designed circuits. Note that Cello input-files should

be in JSON format. To start, we recommend users find the closest set of files to the one being design following the criteria of:

- Promoter arrangement (i.e., tandem, or split)
- Gene location (i.e., plasmid or genome-integrated)
- Chassis-type (e.g., bacteria or yeast)

Then, the user can download their template files from the link (https://github.com/CIDARLAB/Cello-UCF). Next, in each input file, use a text editor to change the information that already exists to the new one following the steps described in §2, §3, and §4. We recommend the users edit each file and each collection within at the time to avoid mistakes. To summarize, the main 'collections' inside the library to be modified are:

- Header: add information from the project and authors
- Measurement_std: add information from experiments and measurements
- Logic_constraints: add information from genetic circuits to be designed
- Genetic_locations: add DNA sequences and information from previously constructed plasmid backbones and strains
- Gates: add information from previously constructed genetic parts and components
- Structures: add information of promoter type (tandem or split) per gate. Follow the instructions presented in §2.9 in this manual
- Models: add parameters from mathematical fittings to the raw data collected per gate
- Components: add DNA sequence information for genetic components
- Toxicity: add normalized data (to first time-point) from the cell growth curves tested in plate-readers (for example)
- Cytometry: add normalized data (to RNAFlux or RPUs) from fluorescence measurements corresponding to gate activity

We recommend users keep the library of collections for types of motifs with EUGENE rules and circuits intact (see examples in Figure 4 in the main Manuscript).

When working with toxicity or cytometry data, convert integer values (e.g., '1') to a float data type (e.g., '1.00'). Otherwise, significant failures when running the UCF library may occur (see 'Troubleshooting' table in the main Manuscript). Also, the number of data points considered in the template files may be larger than the number obtained by users for their new files. Cello 2.0 has no constraints for a minimum number of data points allowed.

Up to this point, all required modifications should be complete. Next steps involve users uploading their files onto the Cello 2.0 available at http://cellocad.org, selecting the 'Settings' for the run, and writing (or uploading) a new (an existing) Verilog file.

In more detail, first, if users opt to run a test in Cello using existing input-files, they can select a (1) UCF library from a list of available options (as in Table 2 in the main Manuscript), then select its corresponding (2) input-sensor file and (3) output-device file, as shown in Figure 5 of the main Manuscript. On the other hand, if users opt to run tests using newly designed files, they can click on a filled-blue-cloud icon button (located on the top-right corner of the browser) to upload the (1)

UCF library. Then, following the exact instructions presented in Figure 5 (Manuscript), users can click on an empty-blue-cloud icon button (located in the top-right corner of the Input- and Output-boxes) to upload the newly designed (2) input-sensor and (3) out-device files. The steps for choosing or uploading existing or newly designed files are described in the section 'GUI' and exemplified in the sections' General Procedures' and 'Case Study of the main Manuscript.

Next, in the 'Settings' tab of Cello 2.0 (see Figure 5 in the main Manuscript), while each stage has a single implementation, some stages may have multiple in the future. Also, while all information affecting the gate architecture is specified in the Cello input files, there may be additional parameters controlling each stage's output. For example, as the number of valid gate orderings returned by the placing stage can be large, a parameter exists to limit that stage's number of results.

Finally, as for Verilog files, users can opt to select an existing file (see Figure 5 in the main Manuscript) or write their files. For the latter, see §5 in this manual and section 'Verilog' of the main Manuscript to find out more about writing Verilog files starting from a few template files.

In the following sections, we present the preparation of the commonly modified collections (Supplementary Listings 27-30) in more detail.

## 6.2 Gate

Each gate is first defined along with some metadata.

```
{
  "collection": "gates",
  "name": "P1_PhlF",
  "system": "TetR",
  "group": "PhlF",
  "regulator": "PhlF",
  "gate_type": "NOR",
  "color": "F9A427",
  "model": "P1_PhlF_model",
  "structure": "P1_PhlF_structure"
}
```

**Supplementary Listing 27**: Gate definition.

The first field collection is mandatory and simply specifies that the object being defined is a gate. The name field should be unique to this gate, duplicate names are not allowed. The system field is required, though not currently used by Cello. The regulator field nominally specifies the name of the gene that regulates transcription of the gate's output and is also not used. The group field is used to group variants of a gate together. For example, gates P1_PhlF and P2_PhlF should be assigned the same group, indicating that only one of them should appear in the circuit because they have the same regulator. The gate_type field can be one of NOR, AND, OR, OUTPUT_OR,

NAND, XOR, or XNOR. The color field is optional and defines the color of the gate when it appears in output drawings. The last two fields, model and structure reference two new objects in the UCF. The model field names an object appearing elsewhere in the UCF (in the models collection) that specifies the quantitative behavior of the gate; for example, the gate's input-output relationship in terms of relative promoter units (RPU). The structure field references an object in the structures collection that defines how to compose different parts to build the gate.

## 6.3 Model


```
{
  "collection": "models",
  "name": "P1_PhlF_model",
  "functions": {
    "response_function": "Hill_response",
    "input_composition": "linear_input_composition",
    "toxicity": "P1_PhlF_toxicity",
    "cytometry": "P1_PhlF_cytometry"
  },
  "parameters": [
    {
      "name": "ymax",
      "value": 3.21,
      "description": "Maximal transcription"
    },
    {
      "name": "ymin",
      "value": 0.006,
      "description": "Minimal transcription"
    },
    {
      "name": "K",
      "value": 0.211,
      "description": "Half-maximum"
    },
    {
      "name": "n",
      "value": 3.57,
      "description": "Cooperativety"
    }
  ]
}
```


**Supplementary Listing 28**: Gate model.

The gate model specifies quantitative behavior of the gate. This includes the response function and any auxiliary functions and their numerical parameters, gate toxicity, and gate cytometry. The first field collection of this object is mandatory and specifies that the object is a gate model. The name field must correspond to the model field for the associated gate object. The functions field is a dictionary that specifies all the quantitative behavior of the gate. The only required key is response_function. The next key/value pair, "input_composition": "linear_input_composition"

refers to a function that indicates how the gate's inputs are combined into a single free variable for the response function. The dependence of these functions will be shown below. The last two keys, toxicity and cytometry are optional. If toxicity is provided for all the gates in the library, a toxicity "score" will be evaluated during gate assignment, as in the standard Cello assignment flow[1]. Likewise, if cytometry is provided for each gate in the library, a prediction of the cytometry profile of the circuit design will be generated.

## 6.4 Structure


```
{
 "collection": "structures",
 "name": "P1_PhlF_structure",
 "inputs": [
  {
   "name": "in1",
   "part_type": "promoter"
  },
  {
   "name": "in2",
   "part_type": "promoter"
  }
 ],
 "outputs": [
  "pPhlF1_a",
  "pPhlF1_b"
 ],
 "devices": [
  {
   "name": "P1_PhlF_a",
   "components": [
    "#in1",
    "P1_PhlF_a_cassette"
   ]
  },
  {
   "name": "P1_PhlF_a_cassette",
   "components": [
    "Kozak1",
    "PhlF_a"
   ]
  },
  {
   "name": "P1_PhlF_b",
   "components": [
    "#in2",
    "P1_PhlF_b_cassette"
   ]
  },
  {
   "name": "P1_PhlF_b_cassette",
   "components": [
    "Kozak1",
```

```
      "PhlF_b"
    ]
   }
  ]
}
```

**Supplementary Listing 29**: Gate structure.

An object in the structures collection encapsulates the architecture of a gate. The inputs field enumerates input part "slots" of the gate. Each input slot is identified by name and must include a part type that will be accepted in the slot. The outputs field enumerates specific parts that carry the output signal of the gate. The structure is open to any type of part, but in all official Cello gate libraries, gates are transcriptional and based on repression; each output part is a promoter that is repressed by a transcription factor produced by the gate. Although there are physical limitations to the number of genes that can be driven by a given transcription factor, Cello does not implement any such restriction. Available output promoters will be assigned to downstream gates as many times as needed. In the example above, there are two available output promoters. Cello interprets the outputs field as a circular list and will iterate and possibly reuse output parts as needed. For example, if gates X, Y, and Z take as input the output of gate A, and gate A has two output promoters pA1 and pA2, Cello will assign these promoters as follows: X gets pA1, Y gets pA2, and Z gets pA1.

The final field of the gate structure object is the devices field. The devices define how input parts are placed with respect to the other parts (genes, terminators, etc.) that compose the gate. A single-input NOT gate in our yeast system is built with one input promoter that regulates a gene. A two-input NOR gate has two copies of this motif: two transcriptional units, each with one input promoter. The devices field in the above example reflects this in a nested structure. Each device in the list includes a name and an ordered list of components. The first device definition in the list, P1_PhlF_a, has two components, #in1 and P1_PhlF_a_cassette. Neither of these is an atomic part. The component #in1 refers to an object in the inputs field, and P1_PhlF_a_cassette is another device in the devices field. Any entry in the list of components must be either a reference to an input (prefixed with #), a part, or another device. The following device, P1_PhlF_a_cassette, includes the Kozak sequence and the gene.

The gate associated with this structure was defined as a NOR gate. Note that this gate may also serve as a "1-input NOR" (a NOT). The gate structure is adapted as follows. If all inputs of a device are unfilled, then the device will be omitted in the gate structure. If some of the inputs are unfiled, then those unfilled inputs will be omitted. For example, if the gate structure given above were utilized as a NOR gate, then only in1 would be filled. The device P1_PhlF_b would contain only unfilled inputs, and thus omitted from the gate structure. (P1_PhlF_b_cassette would also be ignored, as it is a dependency of the P1_PhlF_b device.)

Finally, note that each part, PhlF_a or Kozak1 for example, is defined by a name, sequence, and part type in the parts collection.

## 6.5 Gate placement rules


```
{
  "collection": "circuit_rules",
  "rules": {
    "function": "AND",
    "rules": [
      "[0] EQUALS L1",
      "[1] EQUALS TCYC7",
      "[2] EQUALS RiboJ00_S1",
      "[4] EQUALS TENO1",
      "[5] EQUALS ElvJ_S3",
      "[7] EQUALS TGND1",
      "[8] EQUALS RiboJ51_S5",
      "[10] EQUALS TTIP1",
      "[11] EQUALS RiboJ10_S7",
      "[13] EQUALS TMRP4",
      "[14] EQUALS RiboJ54_S9",
      "[16] EQUALS TGRE3",
      "[17] EQUALS RiboJ60_S11",
      "[19] EQUALS TSED1",
      "[20] EQUALS LtsvJ_S13",
      "[22] EQUALS THUG1",
      "[23] EQUALS S15",
      "[25] EQUALS TGAT2",
      "[26] EQUALS L2",
      "[27] EQUALS TADH1",
      "[28] EQUALS PlmJ_S2",
      "[30] EQUALS TTEF1",
      "[31] EQUALS AraJ_S4",
      "[33] EQUALS TGSY2",
      "[34] EQUALS ScmJ_S6",
      "[36] EQUALS TACT1",
      "[37] EQUALS RiboJ53_S8",
      "[39] EQUALS TPDC6",
      "[40] EQUALS RiboJ57_S10",
      "[42] EQUALS THSP26",
      "[43] EQUALS SarJ_S12",
      "[45] EQUALS TSPO1",
      "[46] EQUALS RiboJ64_S14",
      "[48] EQUALS TADH2",
      "[49] EQUALS S16",
      "[51] EQUALS TFUM1",
      "ALL_FORWARD"
    ]
  }
}
```


**Supplementary Listing 30**: Gate placement rules.

The yeast gate system has special constraints for the placement of gates. Terminators and spacers are at fixed locations, and gates are placed in between. The rules in the listing above are defined in the circuit_rules collection and define constraints on the placement of gates in the

plasmid or genome. There is a single rules field, which contains a list of rules the logical conjunction of which must be satisfied. Each rule in the list is written in the Eugene syntax. In the listing, all the rules use the EQUALS keyword. Rules of this form require that a particular object be placed at the given position in the DNA sequence. The rules given above specify the placement of spacers and terminators that are not a part of the gate structure. The indices excluded from the list are available for gate placement. Note that writing rules in this fashion, where only EQUALS rules are used (except for ALL_FORWARD), and where the parts specified are of type scar or of type spacer followed by terminator, triggers a special rule-processing routine in Cello. Cello will count the number of gates to be placed in the circuit, distribute them evenly amongst the different integration sites, and then omit all parts after the last gate in an integration site so that no unnecessary scars or spacers are included.

# 7 Case study: Results

## 7.1 results.json

Cello generates a metadata file named results.json that describes all the results that were generated for a particular project. We present the results in the Supplementary Listings 31-35 and Supplementary Figures 4-6.

---


```
[ {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_logicSynthesis.dot",
  "stage" : "logicSynthesis"
}, {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_logicSynthesis.pdf",
  "stage" : "logicSynthesis"
}, {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_logicSynthesis.png",
  "stage" : "logicSynthesis"
}, {
  "name" : "logic",
  "description" : "The truth table for the circuit.",
  "file" : "hhehe_logic.csv",
  "stage" : "technologyMapping"
}, {
  "name" : "toxicity",
  "description" : "The toxicity for the circuit.",
  "file" : "hhehe_toxicity.csv",
  "stage" : "technologyMapping"
}, {
  "name" : "activity",
  "description" : "The activity for the circuit.",
  "file" : "hhehe_activity.csv",
```

```
  "stage" : "technologyMapping"
}, {
  "name" : "response_plot",
  "description" : "The response plot for node $1.",
  "file" : "response_plot_$1_P1_BM3RI.png",
  "stage" : "technologyMapping"
}, {
  "name" : "response_plot",
  "description" : "The response plot for node $2.",
  "file" : "response_plot_$2_P1_PhlF.png",
  "stage" : "technologyMapping"
}, {
  "name" : "response_plot",
  "description" : "The response plot for node $3.",
  "file" : "response_plot_$3_P1_IcaR.png",
  "stage" : "technologyMapping"
}, {
  "name" : "response_plot",
  "description" : "The response plot for node $4.",
  "file" : "response_plot_$4_P2_LexA.png",
  "stage" : "technologyMapping"
}, {
  "name" : "response_plot",
  "description" : "The response plot for node $5.",
  "file" : "response_plot_$5_P1_CI434.png",
  "stage" : "technologyMapping"
}, {
  "name" : "response_plot",
  "description" : "The response plot for node $6.",
  "file" : "response_plot_$6_P1_PsrA.png",
  "stage" : "technologyMapping"
}, {
  "name" : "cytometry_plot",
  "description" : "The cytometry plot for node $1.",
  "file" : "cytometry_plot_$1_P1_BM3RI.png",
  "stage" : "technologyMapping"
}, {
  "name" : "cytometry_plot",
  "description" : "The cytometry plot for node $2.",
  "file" : "cytometry_plot_$2_P1_PhlF.png",
  "stage" : "technologyMapping"
}, {
  "name" : "cytometry_plot",
  "description" : "The cytometry plot for node $3.",
  "file" : "cytometry_plot_$3_P1_IcaR.png",
  "stage" : "technologyMapping"
}, {
  "name" : "cytometry_plot",
  "description" : "The cytometry plot for node $4.",
  "file" : "cytometry_plot_$4_P2_LexA.png",
  "stage" : "technologyMapping"
}, {
  "name" : "cytometry_plot",
  "description" : "The cytometry plot for node $5.",
  "file" : "cytometry_plot_$5_P1_CI434.png",
  "stage" : "technologyMapping"
```

```
}, {
  "name" : "cytometry_plot",
  "description" : "The cytometry plot for node $6.",
  "file" : "cytometry_plot_$6_P1_PsrA.png",
  "stage" : "technologyMapping"
}, {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_technologyMapping.dot",
  "stage" : "technologyMapping"
}, {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_technologyMapping.pdf",
  "stage" : "technologyMapping"
}, {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_technologyMapping.png",
  "stage" : "technologyMapping"
}, {
  "name" : "dnaplotlib",
  "description" : "The sequence diagram generated by dnaplotlib.",
  "file" : "x01_dpl.pdf",
  "stage" : "placing"
}, {
  "name" : "dnaplotlib",
  "description" : "The sequence diagram generated by dnaplotlib.",
  "file" : "x01_dpl.png",
  "stage" : "placing"
}, {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_placing.dot",
  "stage" : "placing"
}, {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_placing.pdf",
  "stage" : "placing"
}, {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_placing.png",
  "stage" : "placing"
}, {
  "name" : "sbol",
  "description" : "An SBOL representation of the design.",
  "file" : "hhehe.xml",
  "stage" : "export"
}, {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_export.dot",
  "stage" : "export"
}, {
```

```
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_export.pdf",
  "stage" : "export"
}, {
  "name" : "netlist",
  "description" : "The netlist.",
  "file" : "x01_export.png",
  "stage" : "export"
} ]
```


**Supplementary Listing 31**: A metadata file describing all the results generated by Cello. Each result is described by a name, description, file name, and the stage in which it was generated.

## 7.2 Netlist

For every successful circuit design, Cello will return a netlist in JSON format. The netlist is a specification of the Boolean graph along with the biological gate assignments. Cello includes the different sequence variants (placements) in the JSON netlist file as well. Below is an example netlist output for an implementation of the 0x01 (three-input AND gate) in *S. cerevisiae*.


```
{
  "name": "x01",
  "inputFilename": "x01.v",
  "placements":
  [
    [
      {
        "name": "",
        "components":
        [
          {
            "name": "Group0_Object1",
            "node": "null",
            "direction": 1,
            "parts":
            [
              "TCYC7",
              "RiboJ00_S1"
            ]
          },
          {
            "name": "Group0_Object2",
            "node": "$4",
            "direction": 1,
            "parts":
            [
              "pCI4341_a",
              "P2_LexA_a_cassette"
            ]
          },
          {
            "name": "Group0_Object4",
```

```
  "node": "null",
  "direction": 1,
  "parts":
  [
    "TENO1",
    "ElvJ_S3"
  ]
},
{
  "name": "Group0_Object5",
  "node": "$1",
  "direction": 1,
  "parts":
  [
    "pPhlF1_a",
    "P1_BM3RI_a_cassette"
  ]
},
{
  "name": "Group0_Object7",
  "node": "null",
  "direction": 1,
  "parts":
  [
    "TGND1",
    "RiboJ51_S5"
  ]
},
{
  "name": "Group0_Object8",
  "node": "$6",
  "direction": 1,
  "parts":
  [
    "Ptet",
    "P1_PsrA_a_cassette"
  ]
},
{
  "name": "Group0_Object10",
  "node": "null",
  "direction": 1,
  "parts":
  [
    "TTIP1",
    "RiboJ10_S7"
  ]
},
{
  "name": "Group0_Object11",
  "node": "$3",
  "direction": 1,
  "parts":
  [
    "pLexA2_a",
    "P1_IcaR_a_cassette"
```

```
      ]
    },
    {
      "name": "Group0_Object13",
      "node": "null",
      "direction": 1,
      "parts":
      [
        "TMRP4",
        "RiboJ54_S9"
      ]
    },
    {
      "name": "Group0_Object14",
      "node": "$5",
      "direction": 1,
      "parts":
      [
        "Pxyl",
        "P1_CI434_a_cassette"
      ]
    },
    {
      "name": "Group0_Object15",
      "node": "null",
      "direction": 1,
      "parts":
      [
        "TGRE3"
      ]
    }
  ]
},
{
  "name": "",
  "components":
  [
    {
      "name": "Group1_Object1",
      "node": "null",
      "direction": 1,
      "parts":
      [
        "TADH1",
        "PlmJ_S2"
      ]
    },
    {
      "name": "Group1_Object2",
      "node": "out",
      "direction": 1,
      "parts":
      [
        "pBm3RI1_a",
        "YFP_reporter_cassette"
      ]
```

```
},
{
  "name": "Group1_Object4",
  "node": "null",
  "direction": 1,
  "parts":
  [
    "TTEF1",
    "AraJ_S4"
  ]
},
{
  "name": "Group1_Object5",
  "node": "$2",
  "direction": 1,
  "parts":
  [
    "Plac",
    "P1_PhlF_a_cassette"
  ]
},
{
  "name": "Group1_Object7",
  "node": "null",
  "direction": 1,
  "parts":
  [
    "TGSY2",
    "ScmJ_S6"
  ]
},
{
  "name": "Group1_Object8",
  "node": "$4",
  "direction": 1,
  "parts":
  [
    "pPsrA1_a",
    "P2_LexA_b_cassette"
  ]
},
{
  "name": "Group1_Object10",
  "node": "null",
  "direction": 1,
  "parts":
  [
    "TACT1",
    "RiboJ53_S8"
  ]
},
{
  "name": "Group1_Object11",
  "node": "$1",
  "direction": 1,
  "parts":
```

```
          [
            "pIcaR1_a",
            "P1_BM3RI_b_cassette"
          ]
        },
        {
          "name": "Group1_Object12",
          "node": "null",
          "direction": 1,
          "parts":
          [
            "TPDC6"
          ]
        }
      ]
    }
  ]
],
"nodes":
[
  {
    "name": "$1",
    "nodeType": "NOR",
    "partitionID": -1,
    "deviceName": "P1_BM3RI"
  },
  {
    "name": "out",
    "nodeType": "PRIMARY_OUTPUT",
    "partitionID": -1,
    "deviceName": "YFP_reporter"
  },
  {
    "name": "$2",
    "nodeType": "NOT",
    "partitionID": -1,
    "deviceName": "P1_PhlF"
  },
  {
    "name": "$3",
    "nodeType": "NOT",
    "partitionID": -1,
    "deviceName": "P1_IcaR"
  },
  {
    "name": "$4",
    "nodeType": "NOR",
    "partitionID": -1,
    "deviceName": "P2_LexA"
  },
  {
    "name": "c",
    "nodeType": "PRIMARY_INPUT",
    "partitionID": -1,
    "deviceName": "IPTG_sensor"
  },
```

```
  {
    "name": "$5",
    "nodeType": "NOT",
    "partitionID": -1,
    "deviceName": "P1_CI434"
  },
  {
    "name": "$6",
    "nodeType": "NOT",
    "partitionID": -1,
    "deviceName": "P1_PsrA"
  },
  {
    "name": "a",
    "nodeType": "PRIMARY_INPUT",
    "partitionID": -1,
    "deviceName": "aTc_sensor"
  },
  {
    "name": "b",
    "nodeType": "PRIMARY_INPUT",
    "partitionID": -1,
    "deviceName": "Xylose_sensor"
  }
],
"edges":
[
  {
    "name": "e1__$1_out",
    "src": "$1",
    "dst": "out"
  },
  {
    "name": "e2__$2_$1",
    "src": "$2",
    "dst": "$1"
  },
  {
    "name": "e3__$3_$1",
    "src": "$3",
    "dst": "$1"
  },
  {
    "name": "e4__$4_$3",
    "src": "$4",
    "dst": "$3"
  },
  {
    "name": "e5__c_$2",
    "src": "c",
    "dst": "$2"
  },
  {
    "name": "e6__$5_$4",
    "src": "$5",
    "dst": "$4"
```

```
    },
    {
      "name": "e7__$6_$4",
      "src": "$6",
      "dst": "$4"
    },
    {
      "name": "e8__a_$6",
      "src": "a",
      "dst": "$6"
    },
    {
      "name": "e9__b_$5",
      "src": "b",
      "dst": "$5"
    }
  ]
}
```

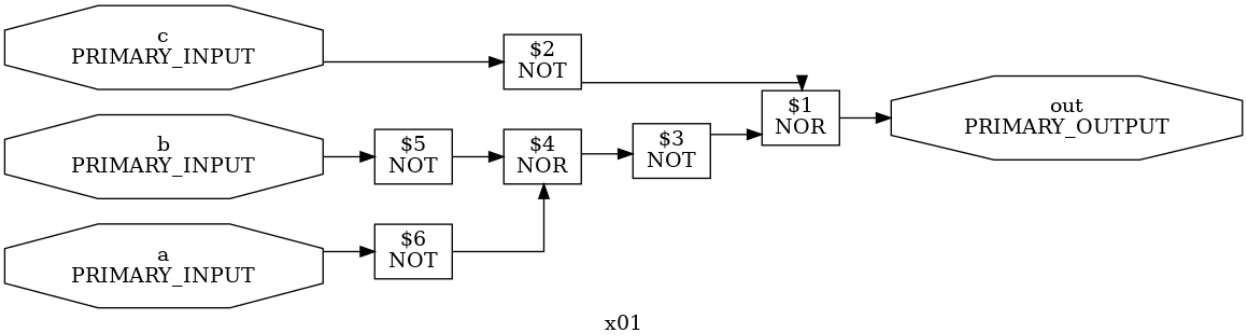

**Supplementary Listing 32**: The output netlist for the *S. cerevisiae* 0x01 case study.

The netlist has a name field as well as an inputFilename field that indicates the name of the Verilog file that the netlist was generated from. Afterward, the netlist has a placements field, a nodes field, and an edges field.

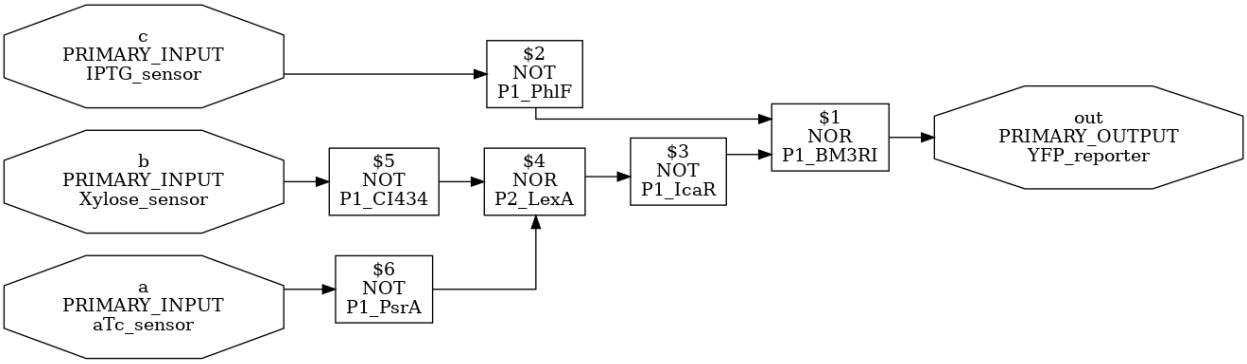
The placements field is a list of different sequence specifications. Each placement in the list is itself an array, and each object in the placement corresponds to a particular integration site. Each integration site has a name and a components field. Each object in the components field corresponds to a gate or some collection of non-gate separator parts.

The nodes and edges fields in the netlist encode the structure of the graph. Each node has a name, a Boolean gate type, and a biological gate assignment. Each edge simply has a name, and a source node and destination node.

## 7.3 Stage netlist diagrams




**Supplementary Figure 4**: A pictorial representation of the netlist for the 0x01 (three-input AND gate) circuit after the logic synthesis stage. Each node contains a node name, e.g. c or $5, and a node type, e.g. PRIMARY_INPUT or NOT. Primary inputs and outputs are drawn as octagons.




**Supplementary Figure 5**: A pictorial representation of the netlist for the 0x01 (three-input AND gate) circuit after the technology mapping stage. Each node now contains a biological device assignment, e.g. P2_LexA or IPTG_sensor.

## 7.4 SBOL file

### 7.4.1 Truth table

```
out,false,false,false,false,false,false,false,true
b,false,false,false,false,true,true,true,true
$3,true,true,true,true,true,true,false,false
$1,false,false,false,false,false,false,false,true
$6,true,true,false,false,true,true,false,false
$5,true,true,true,true,false,false,false,false
$2,true,false,true,false,true,false,true,false
a,false,false,true,true,false,false,true,true
$4,false,false,false,false,false,false,true,true
c,false,true,false,true,false,true,false,true
```

**Supplementary Listing 33**: The truth table associated with the 0x01 (three-input AND gate) encoded in a comma-separated value (CSV) file. If implemented in a cell, the circuit should be expected to satisfy this truth table. Each row corresponds to one node in the netlist. The first column is the node name, and every remaining column is a particular state of the netlist.

### 7.4.2 Activity table

```
out,4.03e-03,4.41e-03,4.03e-03,4.41e-03,4.03e-03,4.41e-03,4.22e-03,5.06e+00
b,3.00e-03,3.00e-03,3.00e-03,3.00e-03,1.80e+00,1.80e+00,1.80e+00,1.80e+00
$3,2.64e+00,2.64e+00,2.64e+00,2.64e+00,2.64e+00,2.64e+00,4.31e-03,4.31e-03
$1,4.03e-03,4.41e-03,4.03e-03,4.41e-03,4.03e-03,4.41e-03,4.22e-03,5.06e+00
$6,2.62e+00,2.62e+00,1.24e-02,1.24e-02,2.62e+00,2.62e+00,1.24e-02,1.24e-02
$5,3.57e+00,3.57e+00,3.57e+00,3.57e+00,2.28e-02,2.28e-02,2.28e-02,2.28e-02
$2,3.20e+00,6.47e-03,3.20e+00,6.47e-03,3.20e+00,6.47e-03,3.20e+00,6.47e-03
a,2.00e-03,2.00e-03,2.50e+00,2.50e+00,2.00e-03,2.00e-03,2.50e+00,2.50e+00
$4,1.10e-02,1.10e-02,1.11e-02,1.11e-02,1.14e-02,1.14e-02,2.22e+00,2.22e+00
c,8.20e-03,2.50e+00,8.20e-03,2.50e+00,8.20e-03,2.50e+00,8.20e-03,2.50e+00
```

**Supplementary Listing 34**: The predicted (not experimental) activity table associated with the 0x01 (three-input AND gate) encoded in a CSV file. Each row corresponds to one node in the netlist. The first column is the node name, and every remaining column contains a transcriptional activity value in the unit specified by the measurement standard section of the UCF (see §2.2). Each column in this table corresponds to the column in the same position in the truth table. For example, when the node out has Boolean value true, it has transcriptional activity 5.06. The accuracy of these predictions can only be tested by manufacturing the circuit in question. Nielsen *et al.* found 45 of 60 circuits tested exhibited a truth table matching the prediction.[1] In many but not all cases the predicted transcriptional activity deviated from the measured value significantly even in circuits with a correct truth table. In the gate characterization protocol, gates are measured independently from one another; unmodeled interactions between gates are likely a significant source of error in the predictions. The reader is referred to the study by Nielsen *et al.* of circuit design automation with Cello 1.0 for a comparison of predicted and expected values of transcriptional activity in *E. coli* circuits.[1]

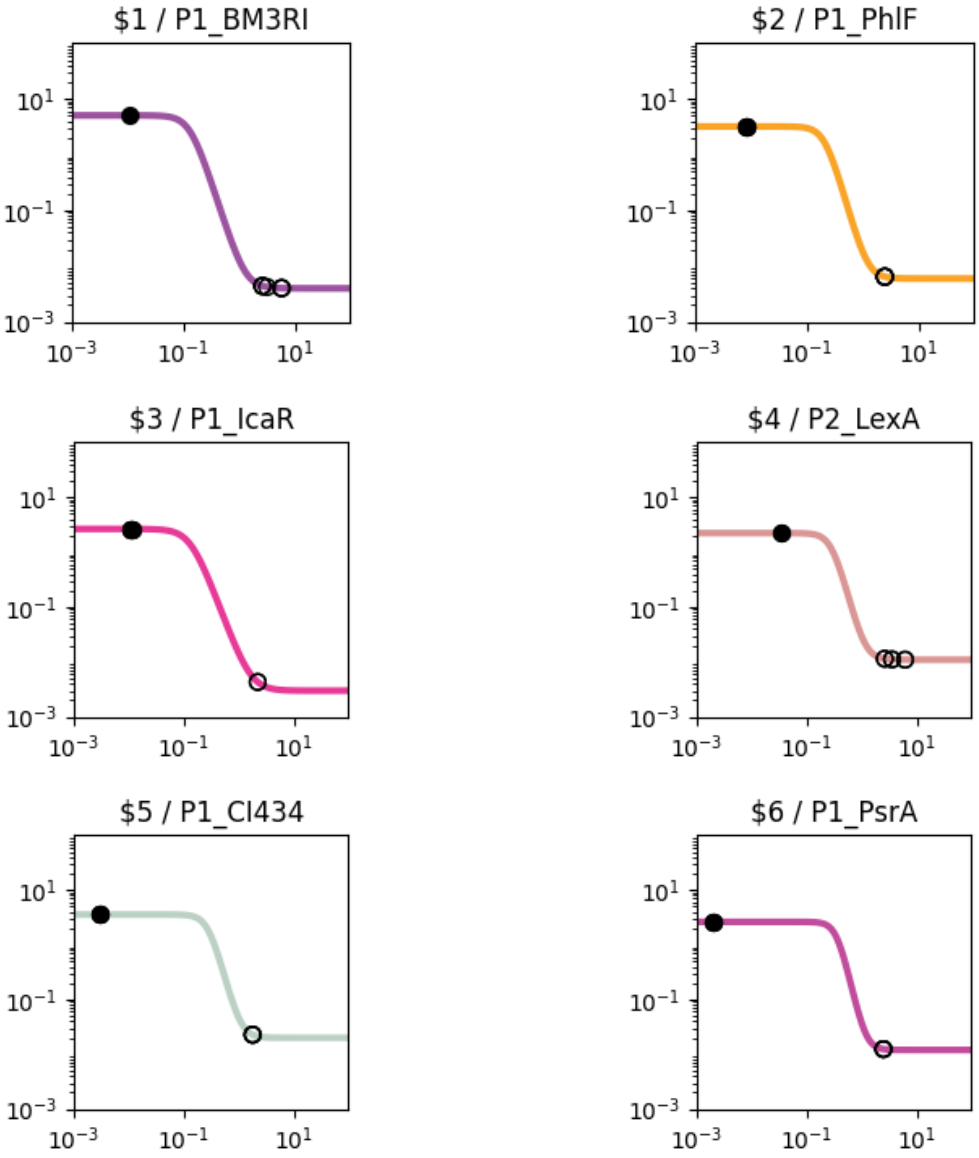
### 7.4.3 Toxicity table

```
$3,0.97,0.97,0.97,0.97,0.97,0.97,0.93,0.93
$1,0.98,1.00,0.98,1.00,0.98,1.00,1.00,1.00
$6,1.00,1.00,0.97,0.97,1.00,1.00,0.97,0.97
$5,1.00,1.00,1.00,1.00,0.90,0.90,0.90,0.90
$2,0.98,1.00,0.98,1.00,0.98,1.00,0.98,1.00
$4,0.83,0.83,1.00,1.00,1.00,1.00,1.00,1.00
```

**Supplementary Listing 35**: The predicted (not experimental) toxicity table associated with the 0x01 (three-input AND gate) encoded in a CSV file. Each row corresponds to one node in the

netlist. The first column is the node name, and every remaining column is a particular state of the netlist. These predictions can only be verified by building the associated circuit. See Supplementary Listing 34 for a discussion of the accuracy of such predictions.

### 7.4.4 Response plots




**Supplementary Figure 6**: The response plot of each logic node in the netlist. The title of each plot contains the node name and the assigned biological gate. The markers on the curves indicate the predicted (not experimental) states the node assumes. Filled markers indicate states with Boolean value true, unfilled markers indicate states with Boolean value false.

# Bibliography

1. Nielsen, A. A. *et al.* Genetic circuit design automation. *Science* **352**, aac7341 (2016).
2. Chen, Y. *et al.* Genetic circuit design automation for yeast. *Nat. Microbiol.* **5**, 1349-1360 (2020).
3. Shin, J. *et al.* Programming Escherichia coli to function as a digital display. *Mol. Syst. Biol.* **16**, e9401 (2020).