Genetic Circuit Design Automation with Cello 2.0

Timothy S. Jones
Biological Design Center, Boston University, Boston MA, USA
Department of Electrical and Computer Engineering, Boston University, Boston, MA, USA
tjones01@gmail.com
https://orcid.org/0000-0002-0960-2441

Samuel M.D. Oliveira
Biological Design Center, Boston University, Boston MA, USA
Department of Electrical and Computer Engineering, Boston University, Boston, MA, USA
smdo@bu.edu
https://orcid.org/0000-0002-6914-5529

William Jackson
Biological Design Center, Boston University, Boston MA, USA
Department of Electrical and Computer Engineering, Boston University, Boston, MA, USA
wrjackso@bu.edu
https://orcid.org/0000-0002-1359-1730

Chris J. Myers
Electrical, Computer & Energy Engineering, University of Colorado Boulder, Boulder, CO, USA
chris.myers@colorado.edu
https://orcid.org/0000-0002-8762-8444

Christopher A. Voigt
Synthetic Biology Center, Department of Biological Engineering, Massachusetts Institute of Technology, Cambridge, MA, USA
cavoigt@gmail.com
https://orcid.org/0000-0003-0844-4776

Douglas Densmore
Biological Design Center, Boston University, Boston MA, USA
Department of Electrical and Computer Engineering, Boston University, Boston, MA, USA
dougd@bu.edu
https://orcid.org/0000-0002-7666-6808

Abstract

Cello 2.0 generates a DNA sequence for a “genetic circuit” based on a high-level software description and a library of characterized DNA parts representing Boolean logic gates. The software has been re-designed to enable flexible descriptions of the logic gates' structure and their mathematical models representing dynamic behavior. There are new formal rules for describing the placement of gates in a genome, a new graphical user interface, support for Verilog 2005 syntax, and a connection to the SynBioHub parts repository software environment. Collectively, these features expand Cello's capabilities beyond *Escherichia coli* plasmids to new organisms and broader genetic contexts, including the genome. The design process of Cello 2.0 is divided into distinct stages: logic synthesis, technology mapping, placing, and export. These stages produce an abstract Boolean network from a Verilog file, assign biological parts to each node in the Boolean network, construct a DNA sequence, and generate highly structured and annotated sequence representations suitable for downstream processing and fabrication, respectively. The final result is a sequence implementing the specified Boolean function in the organism and predictions of circuit performance. Experimentalists may take several hours to learn how to configure the software, e.g., develop a familiarity with Verilog and the syntax for gate libraries, while those with computational backgrounds can submit designs minutes. Depending on the size of the design space, jobs may take seconds or hours to complete. Cello 2.0 is cross-platform software written in Java, freely available as open-source code, and deployed in the browser at http://cellocad.org/.

Introduction

Principles of engineering electronic systems and circuits can be extended to design *genetic circuits* – artificial genetic regulatory networks that can perform computation – for use in living cells.[1] Useful principles in this regard include abstraction, modularity, standards, and modeling. A particular design approach is to use composable DNA elements to design sensors, therapeutics, and materials.[2] In the design of electronic systems, digital circuits have matured to the point where high-level functional descriptions can be transformed automatically into semiconductor-based circuits.[3,4] This transformation process is quite powerful not only because it is efficient but also because it is provably functionally correct, removing many errors and sub-optimizations that would have resulted in a manual process.

The digital logic design abstraction can be adapted to transcriptional genetic circuits[5] with two primary modifications. First, since signals are mediated by genetic products that diffuse throughout a shared space in the cell, each biological gate must have a unique genetic product that occurs only once in the circuit to provide signal orthogonality. Second, as each biological gate is constructed around a specific genetic product, the binary on and off states are not uniform over all gates. A software design framework called Cello has used this modified digital logic design paradigm to design genetic circuits in *E. coli*.[6] These circuits respond to specific inputs and use “NOR” primitives to create combinational logic circuits. The gate primitives are defined in a user constraints file (UCF), a JSON file that encodes gate characterization data, part sequences, rules for gate placement, and specifications of the integration sites in the host organism. Sequential logic circuits (those where the output is a function of both input and state) have also been constructed using a similar logic design paradigm.[7]

While Cello version 1.0 was able to design circuits reliably, it also had notable limitations. Gate libraries were limited to a single gate type (NOR) with a fixed architecture: two input promoters in a tandem arrangement. Arbitrary gate structures, regardless of the resulting logic function, were not supported. This first version was also limited in its modeling abilities, in that it always assumed the total input to a gate was a simple linear combination of the gate's inputs, making no attempt to quantify possible "roadblocking" effects of promoters in sequence. The design rules[8] that Cello version 1.0 uses to order parts and gates in a circuit could only be provided as a logical conjunction of constraints with no alternatives. The support for Verilog, the hardware description language used to describe circuit behavior, was minimal. Users could only write either purely "structural" Verilog specifying the exact layout of the desired Boolean circuit or explicitly provide a truth table in a limited "behavioral" Verilog format. Finally, while Cello version 1.0 could accept user-created libraries, only one was officially provided with the software. The Supplementary Material provides a detailed overview of these libraries' syntax and semantics along with sample files and case studies regarding the preparation of libraries.

This work outlines Cello version 2.0. The rest of the Introduction section describes the new features of Cello. The Materials and Procedure sections provide pragmatic instructions on using the software and its resources, including a case study on designing a circuit in *S. cerevisiae*. The case study leads the user through the circuit design process in the graphical user interface from start to finish. All input files are provided, including a Verilog file that specifies the circuit behavior, as well as library files that encode the input & output parts and the biological gates that implement the circuit. A collection of expected results of the case study is included and discussed. The Troubleshooting section lists a few error messages that may appear and suggestions on how to resolve them. The Anticipated results section describes sample outputs. A Supplementary manual is also included and provides descriptions of the UCF structure, input sensor, output device file structures, sample Verilog files, a case study on designing the UCF for *S. cerevisiae*, and a complete list of the results generated by Cello.

General introduction to circuit design

Synthetic biology and rapid, automated design of genetic circuits promise to produce advances in biosensing[9], smart therapeutics[10], and biofuels[11] & biomaterials.[12] Several approaches to the automated design of multiple genetic regulatory networks, for exploring the design space[13] and engineering stable systems[14], are documented in the literature. A few of these are SBROME[15], MatchMaker[16], Proto & BioCompiler[17], GenoCAD[18], Device Editor[19], iBioSim[20], RBS Calculator[21], and Genetdes.[22] Other example tools include the development of novel programming languages, such as GSL[23] and GEC[24]. Each of these explored different constraints of the genetic circuit design process. For example, while in electrical engineering transistors are highly uniform and share common signal levels for "on" and "off" states, biological parts are more variable in chemistry, structure, and signal levels. Not only does each part or assemblage of parts into a functional unit need to be independently characterized to be *reusable* in many designs, but the design process must also solve the *signal-matching problem* to ensure that an upstream module has a sufficient output signal range to actuate a downstream module. Like in electrical circuits, minimization of circuit size (the total number of DNA base pairs of the circuit or the longest path length from input to output) is desirable. A highly descriptive output format is also important for design verification.

We briefly review the advantages and disadvantages of the Cello 2.0 design approach. Cello is distinct from the tools listed in the previous paragraph in that there is a separation between the gate technology-independent phases in the design process and the technology-dependent phases. A technology-independent Boolean network is optimized first, and then biological gates are mapped to that network. Cello 2.0 has a UCF library format that captures highly specific constraints on gate architecture and parts placement, making circuits designed for a particular gate technology likely to function. Cello has five curated libraries, but the UCF format is also documented so that users can design their own libraries. Though the characterization process presents a non-negligible upfront cost to design circuits in a new host or new gate technology, once it is completed, arbitrary circuits can be designed quickly, giving a distinct advantage over a *de novo* circuit design process. Once the library characterization is complete, Cello users write code in the Verilog hardware description language to compile arbitrary combinational logic into genetic circuits. Verilog is quite flexible, though it requires the user to be familiar with its syntax. The digital logic paradigm for circuit design is also a simplification that may ignore biological phenomena potentially useful for computation.

Cello 2.0 Experimental Design

Overview of the software architecture

Cello 2.0 is an open-source software tool written in the Java programming language. Cello accepts a Verilog file that describes the function of a circuit to be designed, as well as a library of gates and parts and an auxiliary configuration file. The process of converting the Verilog description to a genetic circuit is divided into four stages, see Figure 1. Each stage is intended to perform a specific task, such as synthesizing a Boolean gate diagram (or netlist) from the Verilog description or creating an ordered, genetic sequence from an unordered network of biological gates. Each stage may have many algorithms or implementations, different ways of performing the task associated with it. The auxiliary configuration file picks a particular implementation for each stage and specifies topical parameters, such as the number of sequence variants to produce, that affect the output files that are generated but not the design itself. The library of parts and gates is a composition of three files describing the parts and gates available in a particular organism (the "target"): a user constraints file (UCF) which describes the structure and function of the available gates and their constituent DNA parts, an input sensor file describing the input sensors to the circuit, and the output device file describing the circuit actuators, e.g., fluorescent output reporters. The standard user will interact with Cello as a web application in the browser, available at http://cellocad.org. The web application includes a limited Verilog editor as well as a set of different user constraints files and input and output files that can be selected. More ambitious or advanced users may prepare their own UCFs and load them into the web application, utilize the RESTful HTTP API to submit jobs programmatically, run their instance of the Cello web application, or use Cello locally from the command line.

The Cello workflow is divided into four stages (Figure 1): the logic synthesis (LS) stage, the technology mapping (TM) stage, the placing (PL) stage, and the export (EX) stage. Each stage is equipped with one or more algorithms – different implementations of the tasks associated with that stage. The result is the sequence of a genetic circuit that implements the behavior specified in the Verilog file and can be integrated into the target organism.

Logic synthesis (LS)

In the logic synthesis stage, Cello 2.0 uses an open-source logic synthesis framework called Yosys[25] to produce a Boolean gate network (netlist) based on a Verilog description of the desired circuit function. Verilog is a hardware description language typically used to specify the behavior of field-programmable gate arrays (FPGAs) and other integrated electrical circuits, and the process of compiling Verilog to a gate-level network has been studied at length.[3,4]

Technology mapping (TM)

In the technology mapping stage, biological gates that have been encoded in a user constraints file (UCF) and that implement the Boolean operations in the result of the logic synthesis stage are assigned to every node in the netlist. The gate assignment process is equivalent to a graph coloring problem[26] and is guided by a circuit score (effectively a ratio of the transcription in the ON and OFF states for every output of the circuit) that is optimized by an implementation of the Simulated Annealing algorithm.[6] After the technology mapping stage, all the biological parts that implement the desired circuit function are determined.

Placing (PL)

This stage arranges these parts into a linear DNA sequence. This process is constrained by a set of genetic insert locations or plasmids and rules for the relative positioning of parts. Certain parts may be forbidden by the gate technology to be adjacent to one another—a promoter that roadblocks its upstream neighbor, for example. Rules are specified in the Eugene language[8]. Many examples are shown in Figure 4.

Export (EX)

Finally, in the export stage, a circuit description file is created. The file is encoded using the Synthetic Biology Open Language (SBOL) data standard[27], which allows for extensive annotation of DNA constructs as well as encoding of functional relationships between components using the Sequence Ontology or Systems Biology Ontology, for example. SBOL files also provide provenance information and can be uploaded to an instance of a SynBioHub[28] parts repository where they can be assigned version numbers and grouped into collections.

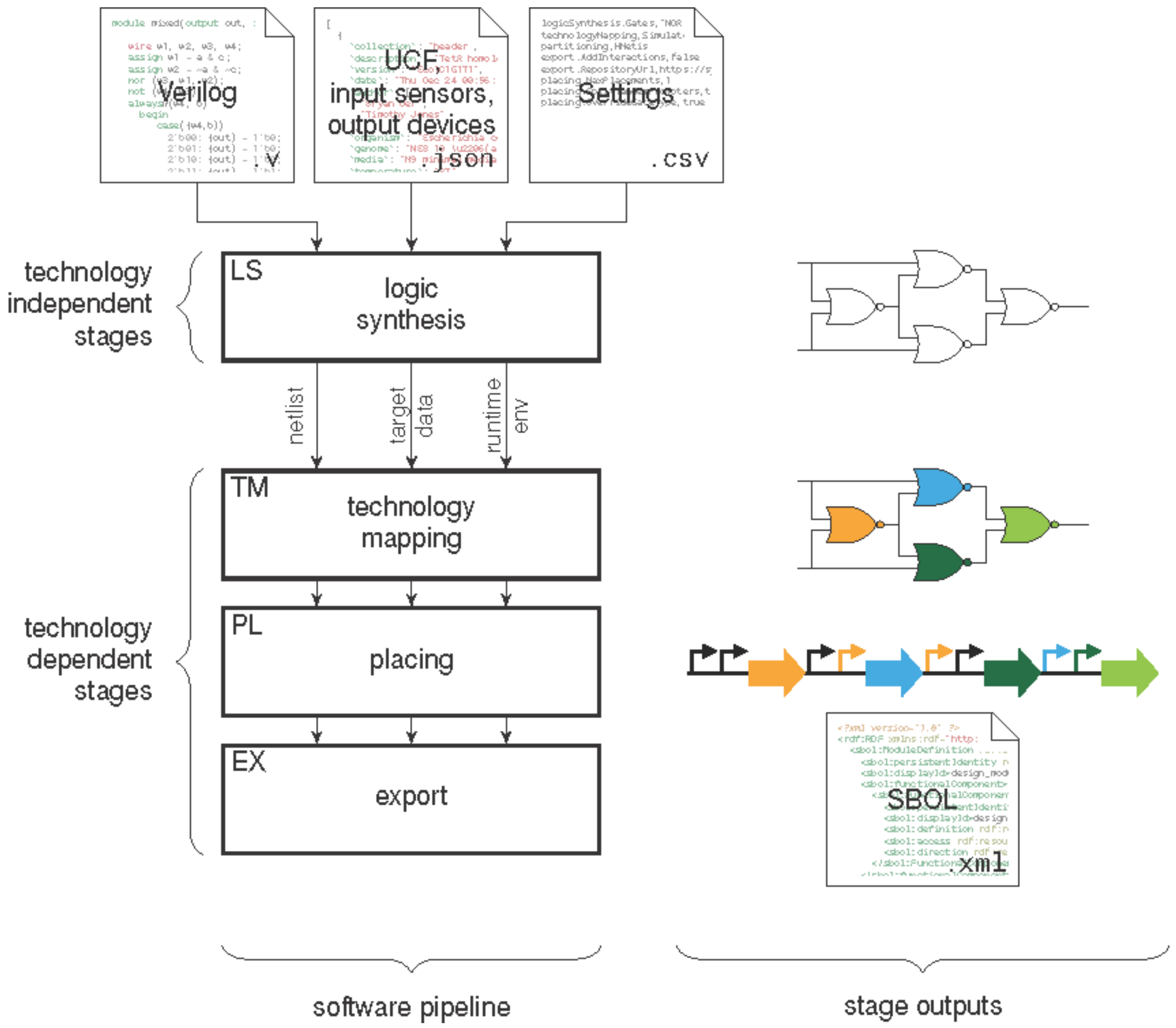
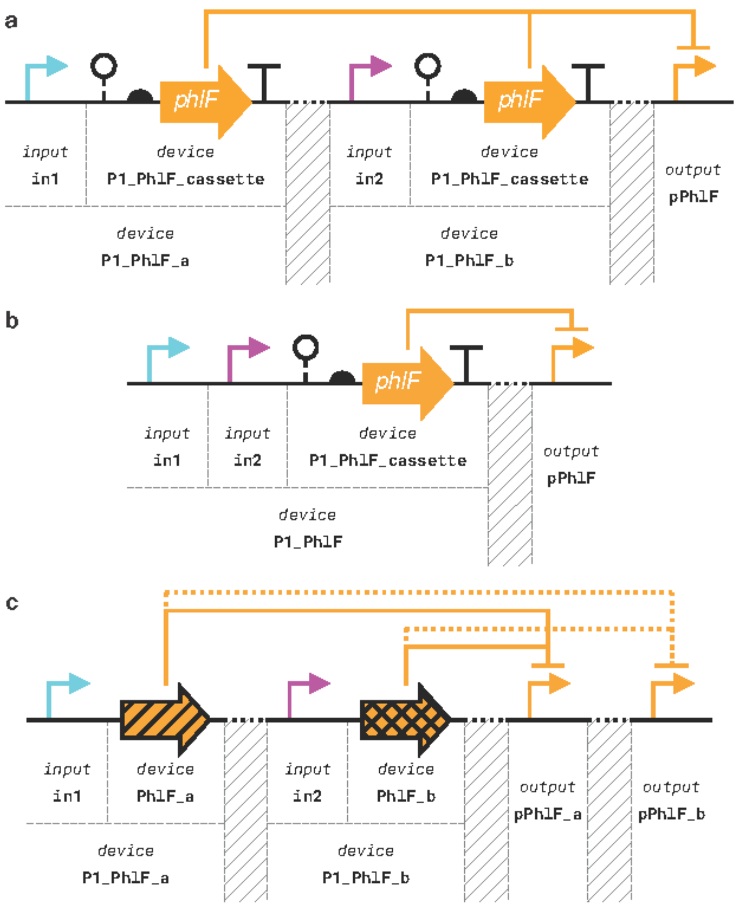


Figure 1: Basic software architecture of Cello 2.0. The workflow is divided into four stages, each with one or more algorithms to perform that stage's tasks. Cello ingests different input files: a Verilog file, the "target data" (a composition of the User Constraints File (UCF), the input sensor file, and the output device file), and a stage configuration file. Each stage produces some characteristic output: the logic synthesis (LS) stage produces a Boolean gate-level network that implements the circuit behavior described by the Verilog code, the technology mapping (TM) stage assigns biological gates to each Boolean gate from the previous stage, the placing (PL) stage orders gates and parts into a linear DNA sequence, and the export (EX) stage produces interchange files (SBOL).

Gate architecture specification

The new UCF format has a structured JSON object type which allows library designers to specify gate architectures using arbitrary hierarchies of parts. This is critical for most of the new libraries, including two-input NOR gates where each input promoter may drive its own sequence variant of the repressor gene and where terminators and insulators may not be a part of the gate architecture. See Figure 2 for example gate architectures and their encodings. The UCF gate architecture description can also accommodate Boolean gate types other than NOR, such as AND, NAND, etc.

The NOR operation is the standard primitive in the UCFs as it is the computational operation implemented by a repressor with two inputs, and NOR logic is "universal" or "functionally complete," meaning that any Boolean function can be implemented exclusively with NOR gates, as can be shown with simple applications of De Morgan's Law.



```
{
  "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_cassette"
      ]
    }, {
      "name": "P1_PhlF_b",
      "components": [
        "#in2",
        "P1_PhlF_cassette"
      ]
    }, {
      "name": "P1_PhlF_cassette",
      "components": [
        "RiboJ53",
        "P1",
        "PhlF",
        "DT5"
      ]
    } ]
}
```

Figure 2: Different NOR gate architectures. **a** A visualization of the gate structure in **d**. **b** A tandem promoter type gate architecture used in libraries Eco1C1G1T1 and Eco1C2G2T2. An example JSON UCF encoding of this gate architecture is given in the Supplementary Manual §2.9.1. **c** A modified split-gate architecture used in the SC1C1G1T1 library. This architecture uses gene and output promoter variants. The genes and promoters produce and respond to the same protein, but the parts are sequence variants of one another to avoid recombination effects. An example JSON UCF encoding of this gate architecture is given in the Supplementary Manual §2.9.3. **d** A UCF encoding of a split-transcriptional-unit type gate architecture, used in the Eco2C1G3T1 and Bth1C1G1T1 libraries. The JSON UCF encoding of this gate architecture is reproduced in the Supplementary Manual §2.9.2.

Custom gate models

Each gate in a Cello library is equipped with a response function that maps the input transcriptional activity to some output activity. The Cello 2.0 UCF format includes a collection of models that can be used to define arbitrary auxiliary functions, e.g., to describe nonlinear input combinations and specify pointers to numerical parameters defined elsewhere. A gate is only required to have defined a response function. The response function can be any scalar-valued function that can be written as a closed-form expression, but in all libraries included with Cello 2.0, gates' responses are modeled by Hill functions. In the UCF, the standard Hill function response is defined only once (Figure 3), but parameters are resolved by pointers to values defined on the gate in which the Hill function is being evaluated. For example, the Hill function definition defines a parameter {"name": "ymin", "map": "#//model/parameters/ymin"} and a gate may define the parameter {"name": "ymin", "value": "0.04"}, the pointer will be resolved to the numeric value.

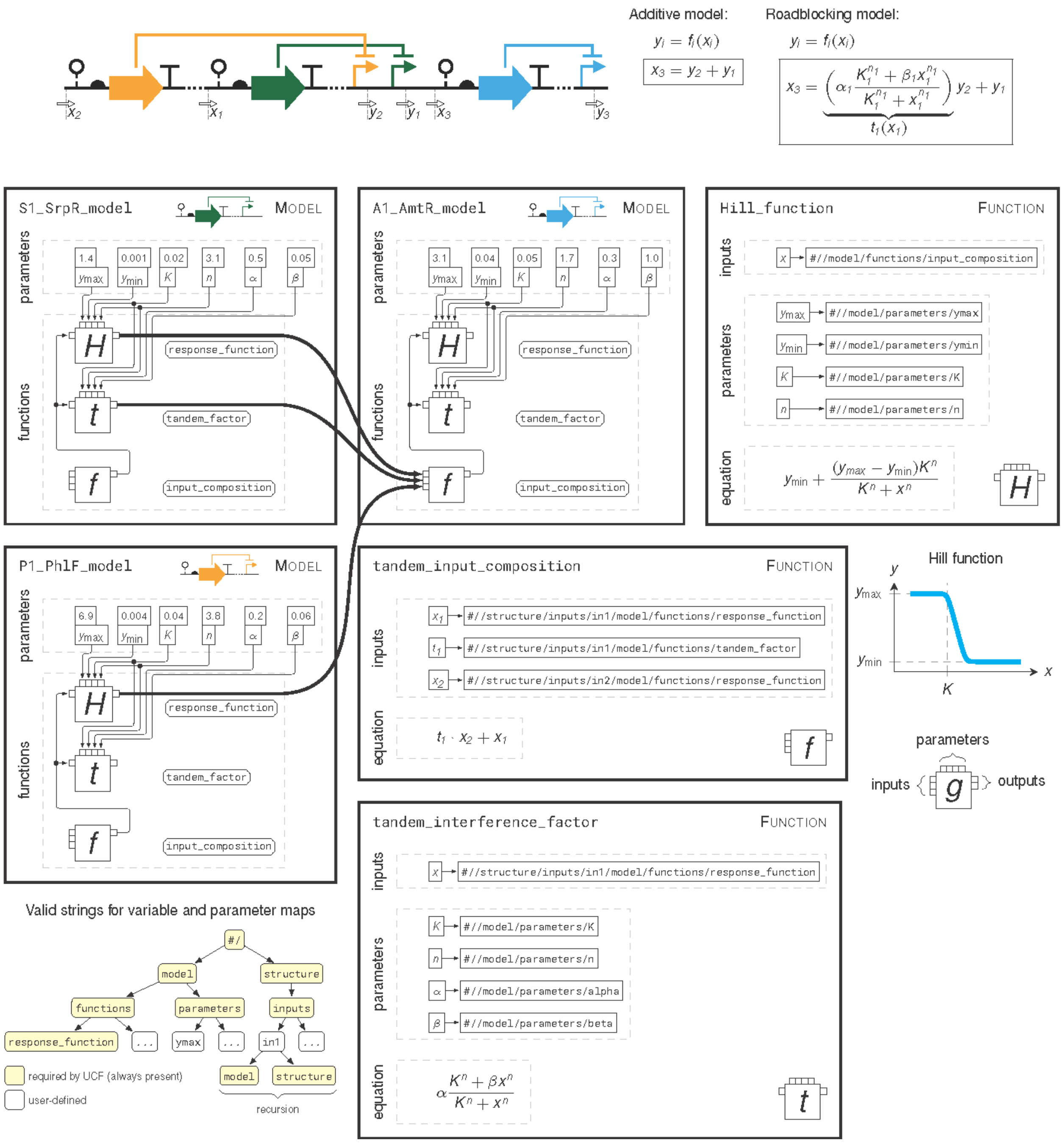
Cello only models transcriptional gates which are structured such that the input to a gate is a transcriptional activity over a regulator (due to some input promoters), and the output of a gate is the transcriptional activity due to a promoter controlled by the regulator. Any unit that measures transcriptional activity in this way can be used to characterize gates and specified in the UCF, but all libraries included with Cello 2.0 use relative promoter units (RPU) to quantify gate inputs and outputs. Note that, as RPU is always defined using an output marker, the definition may change depending on the marker that is used and the host context. The Bacteroides UCF Bth1C1G1T1, which uses a luciferase marker, has an RPU definition that is different from the Eco2C1G3T1 UCF which uses a fluorescent protein that is produced from the genome. The exact definition of the carrier unit is always provided in the UCF, and users that design their own UCFs are free to specify their own signal carrier unit, as described in the Supplementary Manual §2.1.

It should also be noted that this new version of Cello is intended to model combinational logic. These circuits will have outputs that are only a function of their current inputs. To avoid stateful logic (i.e., sequential logic), feedback and cycles are prevented in Cello 2.0 but are being considered for future releases.

We refer to auxiliary functions as functions other than the response function or the cytometry or toxicity functions (defined below). Auxiliary functions are used in the Cello UCFs to define "input composition" functions that indicate how to collect all a gate's inputs into one quantity to be used as the single free variable in the Hill function response. Most libraries use an additive model that assumes the input transcriptional activity to a gate to be the sum of the transcriptional activities of the gate's input promoters. However, if input promoters occur immediately after one another in a tandem arrangement, roadblocking effects can occur. Such effects are quantifiable, and the Eco1C2G2T2 UCF (containing tandem promoters) considers a gate's input to be a weighted sum of the input promoters' activities.[29] The weights in the sum depend on other functions defined by a gate, implemented in the UCF as a chaining of auxiliary functions, as shown in Figure 3.

Cytometry distributions at various input levels can also be included in a gate's model. While it is the Hill function, the parameters of which are extracted from the cytometry distributions, that is used in the process of finding the optimal gate assignment, the inclusion of cytometry distributions in the UCF will cause the creation of a predicted cytometry distribution at each output state of each gate. As a cytometry distribution will be included in the UCF for each of a finite set of input levels, Cello will interpolate a new distribution given the predicted input to that gate.

Finally, a gate "toxicity" measure can also be included in the gate model (see Supplementary Manual §2.7). If, for each gate in the library, a lookup table function is defined that maps some input value in RPU to a real number on the unit interval, taken to be a percentage of cell growth relative to an unloaded cell (a host cell without the gate in question), then Cello will predict net toxicity due to all gates in the circuit. In all five Cello libraries (see Table 2), cell growth is measured via optical density at 600 nm (OD_{600}). Note that gates are measured individually for their effect on cell growth. Cello 2.0 and the UCF format do not support encoding of non-additive or synergistic toxicity produced by specific combinations of gates, though support for this feature may be added in future versions of Cello.



Additive model:
$y_i = f_i(x_i)$
$x_3 = y_2 + y_1$
Roadblocking model:
$y_i = f_i(x_i)$
$x_3 = \underbrace{\left(\alpha_1 \frac{K_1^{n_1} + \beta_1 x_1^{n_1}}{K_1^{n_1} + x_1^{n_1}}\right)}_{t_1(x_1)} y_2 + y_1$
x_2 x_1 y_2 y_1 x_3 y_3
S1_SrpR_model
MODEL
parameters
1.4 0.001 0.02 3.1 0.5 0.05
ymax ymin K n α β
functions
H response_function
t tandem_factor
f input_composition
A1_AmtR_model
3.1 0.04 0.05 1.7 0.3 1.0
P1_PhlF_model
6.9 0.004 0.04 3.8 0.2 0.06
Hill_function
FUNCTION
inputs
x #//model/functions/input_composition
ymax #//model/parameters/ymax
ymin #//model/parameters/ymin
K #//model/parameters/K
n #//model/parameters/n
equation
$y_{min} + \frac{(y_{max} - y_{min})K^n}{K^n + x^n}$
tandem_input_composition
x_1 #//structure/inputs/in1/model/functions/response_function
t_1 #//structure/inputs/in1/model/functions/tandem_factor
x_2 #//structure/inputs/in2/model/functions/response_function
$t_1 \cdot x_2 + x_1$
tandem_interference_factor
x #//structure/inputs/in1/model/functions/response_function
K #//model/parameters/K
n #//model/parameters/n
α #//model/parameters/alpha
β #//model/parameters/beta
$\alpha \frac{K^n + \beta x^n}{K^n + x^n}$
Hill function
y ymax ymin K x
parameters
inputs g outputs
Valid strings for variable and parameter maps
#/
model structure
functions parameters inputs
response_function ... ymax ... in1 ...
model structure
recursion
required by UCF (always present)
user-defined

Figure 3: Visualization of gate model and function definitions. In all available UCFs, the gates' input/output characteristics are described by Hill functions. The Eco1C2G2T2 library uses a special roadblocking model to compute the input to the Hill function. The function used to compute

the input is parameterized by each gate and referred to in the Hill function definition by a pointer: #//model/functions/input_composition.

DNA mapping with placement rules

Cello uses Eugene rules[8] in the PL stage to constrain the relative placement of parts and gates into a DNA sequence. Examples of such rules along with components that satisfy them are shown in Figure 4. For example, certain input promoter orderings may be disallowed, or a particular gate may need to be placed upstream of another for proper functionality. The Cello 2.0 UCF format has expanded the rule definitions to allow libraries to constrain the placement of gates across predefined integration sites in a plasmid or genome. The *genetic_locations* section of the UCF now includes insert locations that are labeled symbolically, e.g., L1, L2, L3. The insert locations are defined relative to either a complete specification of the host genome or an input plasmid defined in a GenBank file or a link to an NCBI sequence. The insert location symbols can be referenced in the Eugene rules for gate placement. For example, if location L2 occurs after L1, a gate can be constrained to always appear in L1 by the rule GateA BEFORE L2. Cello 2.0 is able to accept rules nested arbitrarily in Boolean functions, for example: OR{ AND{ "GateX_a BEFORE L2", "GateX_b AFTER L2"}, OR{"GateX_a AFTER L", "GateX_b BEFORE L" }}.

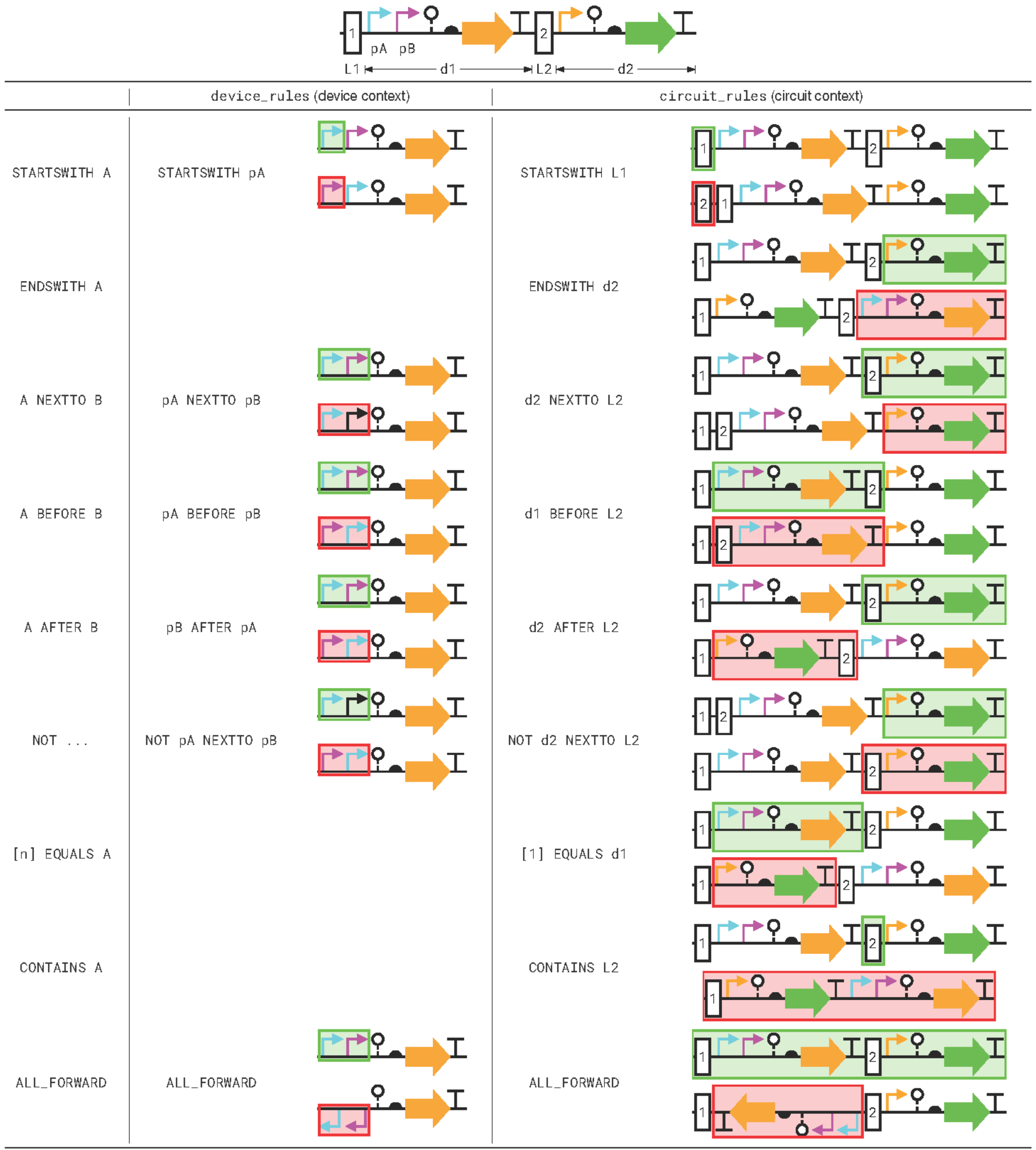


Figure 4: A selection of Eugene rules and circuits that satisfy or violate the rules. Cello accepts rules in the UCF to control the placement of input parts within gates (*device_rules*) and gates within a circuit (*circuit_rules*). Rules in the *circuit_rules* collection can include the location markers (L1 or L2 in the Figure) defined in the *genetic_locations* collection to control the placement of devices (gate fragments) across different plasmids or genome locations. The selections of circuits marked in green are those that pertain directly to and satisfy the adjacent rule. Selections marked in red violate the rule. pA and pB are promoters, d1 and d2 are devices (substructures of a gate). Location markers 1 (L1) or 2 (L2) are not parts but rather mark the beginning of an insert

location in the UCF's *genetic_locations* section. For example, everything to the right of the '1' marker, up until the '2' marker, would be placed at genetic location L1.

SBOL and SynBioHub

After a successful project run, Cello generates an encoding of the genetic circuit structure using the SBOL data standard[27] in the export (EX) stage. SBOL is a data standard that can describe a genetic construct in a hierarchical fashion, also encoding information like molecular interactions and numerical model parameterizations that cannot be represented natively in standard annotated sequence formats like GenBank or FASTA.

The information about the genetic design encoded in SBOL can be shared using the SynBioHub[28] repository. SynBioHub is an open-source repository that stores genetic design information encoded using SBOL. It supports the search, visualization, and exchange of this information both using a graphical user interface (GUI) and an application programming interface (API). SynBioHub facilitates the exchange and reuse of genetic parts and circuits between various research groups and software tools. To that end, the Cello web application GUI provides an interface to upload the SBOL-encoded genetic circuit to SynBioHub. Furthermore, all of Cello's UCF libraries have been converted to SBOL format and uploaded to a SynBioHub instance[*].

Verilog

Cello 2.0 uses the open-source logic synthesis tool Yosys[25], which comes with extensive support for the Verilog 2005 specification. This lets users specify the circuit logic using control structures familiar from high-level programming languages and will be useful when sequential logic or circuits partitioned over multiple cells become more common. It also allows for the exploration of many existing Verilog implementations to be tested quickly in Cello. See Table 1 for examples of various types of Verilog syntax supported in Cello.

Syntactic feature	**Example**	**Benefits**

structural	`not (w1, a);`
	`nor (w2, b, w1);`
	`nor (out, w1, w2);`
		simple, predefined layouts

continuous assign statements	`assign o = ~((a & b) | c ^ d);`	pure combinational logic expressions

conditionals	`if (reset == 1)`
	`  begin`
	`    q = 0;`
	`  end`
		sequential logic, complex designs

loops	`for (i = 0; i < n - 1; i++)`
	`  bus[i] = bus[i + 1];`
		flexible program control, complex designs

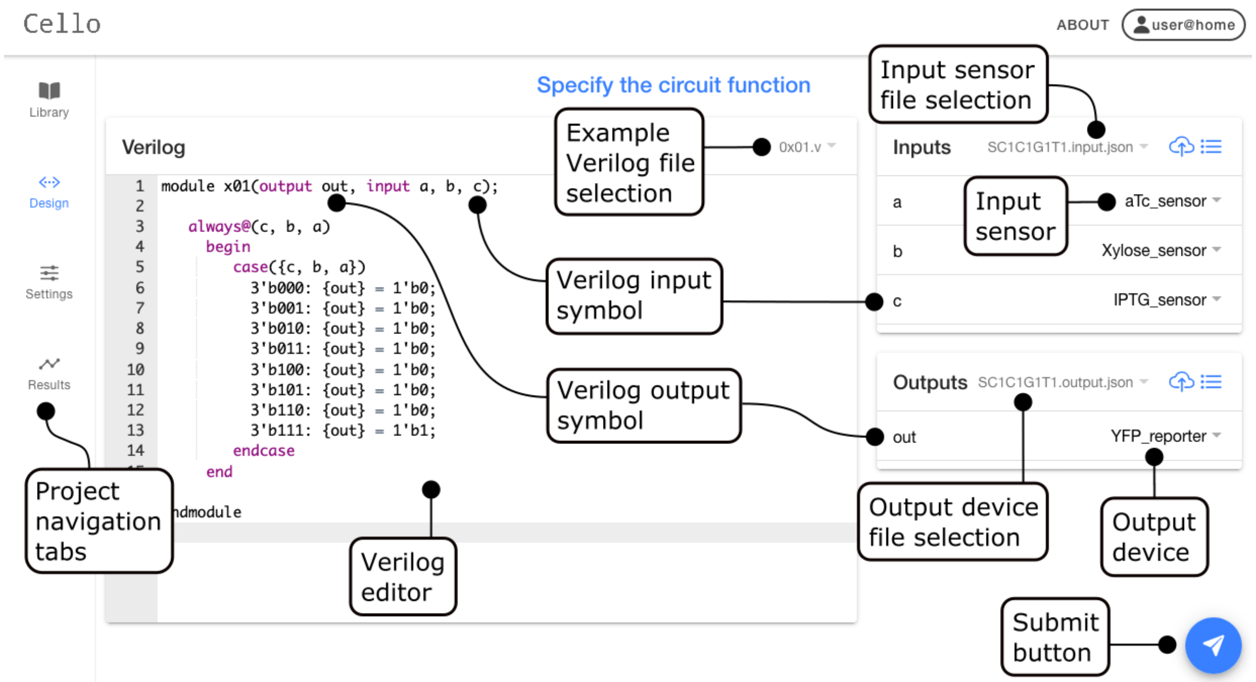
[*] https://synbiohub.programmingbiology.org

		relevant for future genetic circuits

Table 1: Example Verilog syntax supported in Cello 2.0. It supports almost all Verilog 2005 syntax, including continuous assign statements, conditionals, and loops.

GUI

Cello 2.0 is outfitted with a new GUI served in the browser, available at http://cellocad.org. The user creates an account in the web application, then either retrieves a previously created project (a genetic circuit design) or begins a new project. A project view is divided into four tabs for the circuit specification and design results: the Library tab where the user chooses or uploads a UCF, the Design tab (Figure 5) where the user provides a Verilog circuit description and assigns input and output devices, a Settings tab where the user may tweak minor aspects of the workflow such as the number of sequence variants, and a Results tab (Figure 6) to view the final circuit design and its predicted behavior.



```
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

ndmodule
```

0x01.v

Inputs SC1C1G1T1.input.json

a	aTc_sensor
b	Xylose_sensor
c	IPTG_sensor

Outputs SC1C1G1T1.output.json

out	YFP_reporter

Figure 5: The Verilog editor and input & output selection panes in the GUI. In a project specification, the user navigates between this view and the library selection, advanced settings, and results views using tabs on the left-hand side. The user can choose from a few example Verilog files. Input and output symbols, whether provided in the Verilog module declaration or in the body of the module, are mapped to input sensor and output device selection boxes on the right. The user uploads or selects an available set of inputs and outputs, then optionally assigns a sensor or reporter to each input and output symbol extracted from the Verilog.

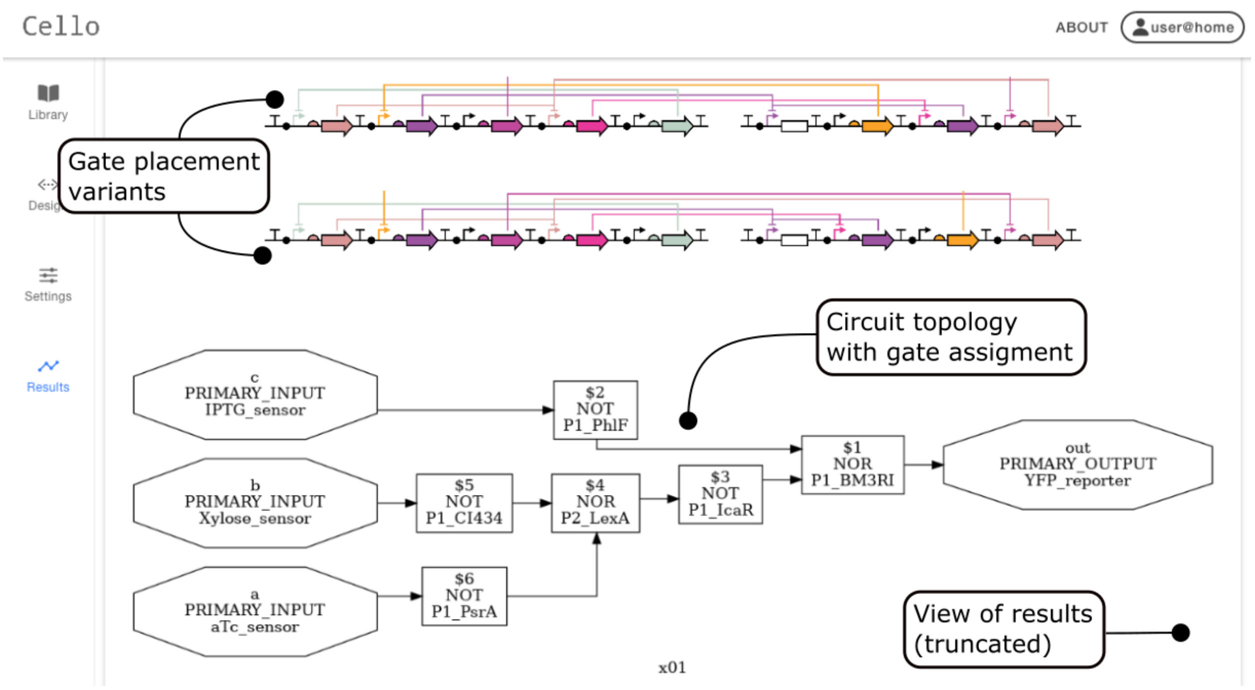

Cello
ABOUT
user@home
Library
Design
Settings
Results
Gate placement variants
Circuit topology with gate assigment
View of results (truncated)
c PRIMARY_INPUT IPTG_sensor
b PRIMARY_INPUT Xylose_sensor
a PRIMARY_INPUT aTc_sensor
$2 NOT P1_PhlF
$5 NOT P1_CI434
$4 NOR P2_LexA
$6 NOT P1_PsrA
$3 NOT P1_IcaR
$1 NOR P1_BM3RI
out PRIMARY_OUTPUT YFP_reporter
x01

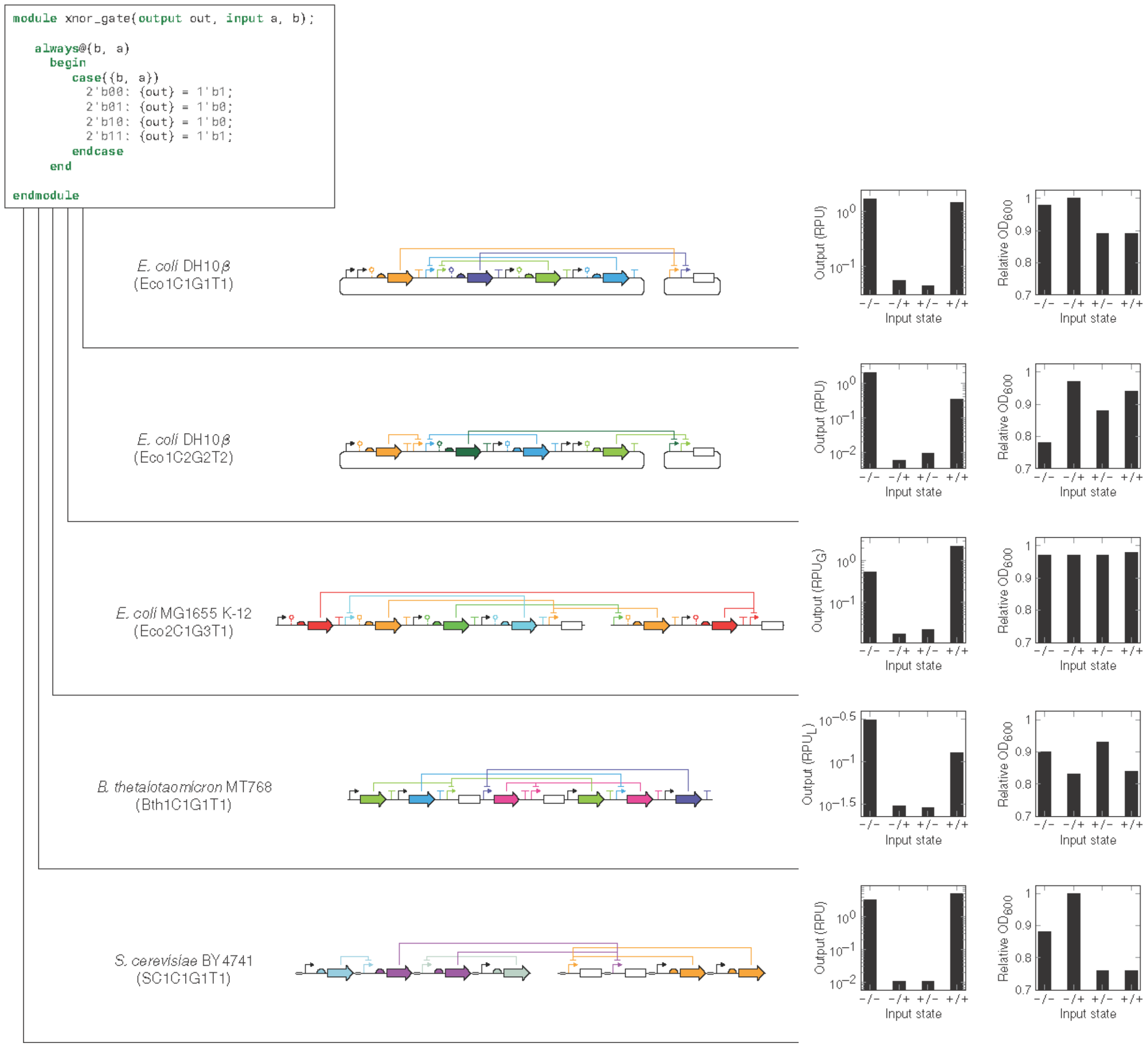
Figure 6: The results view in the GUI. The circuit topology is shown in the graph at the bottom. Each node in the network has three pieces of information: the node name, the Boolean gate type, e.g., NOT, NOR, etc., and the biological gate assignment. Many gate placements (orderings of gates into a DNA sequence) may be generated, two are visible in this screenshot. Many other results (unpictured) are also generated: response plots for each gate, predicted cytometry distributions for each gate, a truth table, and a table with predicted transcriptional activity for each gate, and a prediction of optical density measurements (see examples for all in Figure 8).

Support for new organisms

Cello 2.0 supports new organisms with an updated user constraints file (UCF) format and five ready-made UCFs in different hosts. After beginning a design in the Cello web application graphical user interface (GUI), the user can navigate to the Library tab to select a provided UCF or upload a custom file. All available UCFs are summarized in Table 2, and the results of designing an XNOR gate in all libraries are shown in Figure 7. A complete description of the UCF format that enabled these libraries is included in the Supplementary Manual §2.

Library name	Host	Regulators	Notes & references	Genetic location	Gate definition	Gate model	Cytometry	Signal carrier unit
Eco1C1G1T1	*E. coli* DH10β	TetR homologs: PhlF, SrpR, BM3R1, HlyIIR, BetI, AmtR, AmeR, QacR, IcaRA, LitR, PsrA, LmrA	Original Cello 1.0 library[6]	Plasmid: p15a (circuit & output on separate plasmids)	12 NOT/NOR	Additive	Yes	RPU
Eco1C2G2T2	*E. coli* DH10β	TetR homologs: PhlF, SrpR, BM3R1, HlyIIR, BetI, AmtR, AmeR, QacR, IcaRA	Improved gates and tandem promoter model[29]	Plasmid: p15a	9 NOT/NOR	Non-additive tandem roadblocking	Yes	RPU
Eco2C1G3T1	*E. coli* MG1655 K-12	TetR homologs: PhlF, QacR, BM3R1, BetI, AmtR, AmeRs	Split transcriptional units[30]	Genome, offsets: 3911814, 4196314, 4506858	6 NOT/NOR	Additive	Yes	RPU_G
Bth1C1G1T1	*B. thetaiotaomicron* MT768	CRISPRi: 1,2,...,7	Split transcriptional units, single guide RNAs (CRISPR-dCas9)[31]	Genome: attBT2-1, attBT2-2	7 NOT/NOR	Additive	No	RPU_L
SC1C1G1T1	*S. cerevisiae* BY4741	PhlF, QacR, BM3R1, PsrA, IcarA, CI, CI434, HKCI, LexA	Split transcriptional unit, up to 11 regulatory proteins[32]	Genome: ura3, leu2	9 NOT/NOR	Additive	Yes	RPU

Table 2: Cello 2.0 is bundled with five different libraries in three organisms. Each library contains a different set of gates. The library names follow a naming convention, see the Supplementary Manual §2.1. Note that "7 NOT/NOR" in the Gate definition column indicates that seven gates are defined in the UCF, each one can perform either the NOT or NOR operation. Gates have different architectures, some with tandem promoters and others with two copies of the regulator gene with one promoter for each copy ("split transcriptional units" in the Table). Each gate library also uses a certain signal carrier unit, either RPU or a variant. RPU_G is calculated with the measurement integrated on a genome landing pad, RPU_L is calculated with a luciferase-based measurement standard.



```
module xnor_gate(output out, input a, b);

   always@(b, a)
     begin
        case({b, a})
          2'b00: {out} = 1'b1;
          2'b01: {out} = 1'b0;
          2'b10: {out} = 1'b0;
          2'b11: {out} = 1'b1;
        endcase
     end

endmodule
```

Figure 7: An XNOR gate designed in all five libraries available in Cello 2.0, shown with predicted (not experimental) OD_{600} and RPU output levels.

Materials

- Personal computer with an Internet connection and a web browser like Mozilla Firefox, Google Chrome, or Microsoft Edge with JavaScript enabled.
- A user constraints file (UCF) describing the gates and the target organism. *Complete files for gates in* S. cerevisiae, E. coli, *and* B. thetaiotaomicron *are shipped with the software and are available on Github*[†] *and cellocad.org.* Information on creating a custom UCF is given in the Supplementary Manual §2 and §4.
- An input sensor file and output device file. *Complete files for inputs & outputs in* S. cerevisiae, E. coli, *and* B. thetaiotaomicron *are shipped with the software and are available on Github*[‡,§] *and cellocad.org.*
- (Optional) Docker[**], if you wish to run your own instance of Cello 2.0. The webapp container is available on Docker Hub as cidarlab/cello-webapp[††].
- (Optional) Java Runtime Environment[‡‡] version 8, Yosys[25], Python[§§] version 3, and dnaplotlib[33], if you wish to run an instance of Cello 2.0 in your host operating system outside of a Docker container.

Procedure

General procedure

1 Navigate to http://cellocad.org in your web browser.
Timing 5s.

2 Create a new account or login with an existing account.
Timing 10–60s.

3 Download results from a previous project or create a new project.
Timing 10–60s.

4 Choose an available gate library or upload an already prepared UCF.
Timing 1–10 min (preparing a custom UCF, including gate characterization experiments, may require 40–80 hours of effort). See instructions for preparing Cello input files from scratch in the Supplementary Manual §6.1.

† https://github.com/CIDARLAB/Cello-UCF/tree/develop/files/v2/ucf

‡ https://github.com/CIDARLAB/Cello-UCF/tree/develop/files/v2/input

§ https://github.com/CIDARLAB/Cello-UCF/tree/develop/files/v2/output

** https://www.docker.com/

†† https://hub.docker.com/r/cidarlab/cello-webapp

‡‡ https://java.com/en/download/

§§ https://www.python.org/

5 Open the Verilog tab and compose or paste the Verilog description of your circuit. There is a small number of sample Verilog files available for selection in the GUI. Sample files are also provided in §3 of the Supplementary Manual. In general, the user does not need a strong familiarity with Verilog – the example files can easily be adapted to any Boolean function or truth table.
Timing 1–30 min (users unfamiliar with Boolean logic and basic programming concepts may require additional time to understand Verilog syntax).

6 In the Inputs section, under the file selection menu (see Figure 5), select an input sensor file to choose a set of available sensors. Optionally upload a new input sensor file by clicking on the cloud icon. A file must be selected to proceed with the design. See the Supplemental Manual for information on preparing a custom file.
Timing 1–5 min (preparing a custom input sensor file, including the necessary characterization experiments, may require 40–80 hours of effort).

7 In the Outputs section, under the file selection menu (see Figure 5), select an output device file to choose a set of available output devices. Optionally upload a new output device file by clicking on the cloud icon. A file must be selected to proceed with the design. See the Supplemental Manual for information on preparing a custom file.
Timing 1–5 min (preparing a custom input sensor file, including the necessary characterization experiments, may require 40–80 hours of effort).

8 (Optional) After a Verilog specification is present and an input sensor file has been selected, a specific input sensor can be associated with a particular Verilog input symbol defined in the Verilog module. Likewise, when an output device file has been selected, output devices can be assigned to output symbols in the Verilog.
Timing 0–10 min.

9 (Optional) Open the Settings tab and select stage algorithms or select stage settings for output format. These settings are unlikely to be used by most users.
Timing 0–5 min.

10 Submit the job by clicking on the blue plane icon in the bottom corner, enter a job name, click run, then wait for the job to complete.
Timing 1–30 min, depending on gate library and Verilog input.

11 View results in the results tab. Optionally download results and submit SBOL description of circuit design to a SynBioHub instance.
Timing 10–120 min.

Case study

We present a genetic circuit design case study with Cello 2.0 using the yeast (*S. cerevisiae*) gate library SC1C1G1T1. We design the circuit 0x01, otherwise called a three-input AND gate – the output is only high when all three inputs are high.

1 Follow steps 1-3 in the General procedure section above.
2 In the Library tab, click the library with identifier SC1C1G1T1. This will instruct Cello to use the *S. cerevisiae* gate library when designing the circuit. Note that instructions for preparing Cello input files from scratch are presented in Supplementary Manual §6.1.
3 Open the Design tab. In the Verilog editor, paste the following code:

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

  and(y, a, b, c);

endmodule
```

4 In the Inputs section, under the file selection menu (see Figure 5), select *SC1C1G1T1.input.json*. This will instruct Cello to use the input sensors that have been prepared to work in conjunction with the SC1C1G1T1 UCF.
5 The input symbols in the Verilog file (a, b, and c) should be displayed in the Inputs section. Select *aTc_sensor* in the dropdown menu next to the input a. Select *Xylose_sensor* for b and *IPTG_sensor* for c.
6 In the Outputs section, select the SC1C1G1T1.output.json file.
7 Select *YFP_reporter* for the output y.
8 Follow steps 7 & 8 in the General procedure section above.

The results of the case study are given in the Anticipated results section.

Timing

Steps 1 and 2, navigate to cellocad.org and login: 5–65 s
Step 3, create a new project: 10–60 s
Step 4, choose a gate library or upload your own: 1–10 min
Step 5, enter Verilog: 1–30 min
Steps 6 and 7, select inputs & outputs files or upload your own: 2–10 min
Step 8, assign input sensors and output devices: 0–10 min
Step 9, select stage-specific settings: 0–5 min
Step 10, submit the job: 1–30 min
Step 11, view results or submit to SynBioHub: 10–120 min

Troubleshooting

This section describes a couple of error messages that may occur in the GUI.

Step	**Problem (Error message)**	**Possible reason**	**Solution**

Verilog (Design tab of GUI)	*Error processing inputted Verilog.*	There is a syntax or structural error in the user-provided Verilog code. The GUI provides sample Verilog files, and we encourage designers to modify these as an introduction to the language syntax.	Structural errors are not found until runtime. Ensure that each symbol is defined before the point at which it is evaluated in the code.
Verilog (Design tab of GUI)	*Failed to synthesize verilog into netlist.*	The Verilog code is too structurally complex to create a valid output within the given service time limit.	Reduce the complexity of the circuit by removing declarations and assignments to determine which Verilog element is causing the issue.
UCF selection (Library tab) or Input & Output selections (Design tab)	*Invalid JSON in target data*	The UCF or input and output files provided by the user could not be parsed as valid JSON. Trailing or missing commas and unclosed blocks or strings can trigger this error.	For large files, use a JSON validator such as JSONLint (https://jsonlint.com/) to locate formatting errors.
Placing stage of execution	*Could not resolve inputted rule set.*	The rules provided in the *circuit_rules* or *device_rules* sections of a custom UCF contain incompatible information, e.g., pTet BEFORE pTac AND pTac BEFORE pTET.	Review your custom UCF file. Check the Eugene syntax[11] to ensure you understand the meaning of each keyword. Verify that no pair of rules can create a contradiction. Optionally download the failed project on your projects page and open the Eugene script (see Table 4) to view the subset of rules from the UCF that are applied to the circuit.
Input file selection (Design tab)	*Input Sensor File not found*	The user did not select an input sensor file in the Design tab of the GUI.	See Figure 5 and Step 6 of the Procedure section. Specifying an input sensor file is not optional, as the sensors are characterized with ON and OFF transcriptional activities that propagate to allow Cello to model the behavior of the circuit.

Output file selection (Design tab)	*Output Device File not found*	The user did not select an output device file in the Design tab of the GUI.	See Figure 5 and Step 7 of the Procedure section. Specifying an output device file is not optional.
UCF selection (Library tab)	*UCF File not Found*	The user did not select a UCF in the Library tab of the GUI.	Review Step 4 of the procedure section. The selection of a UCF is not optional. Simply click the row corresponding to the UCF to be used and observe that it is highlighted, indicating that it is selected. A UCF that has been uploaded will appear in the table for selection and must be chosen after upload.
Project Submission	*JSON Conversion Problem*	The UCF or input and output files provided by the user could not be parsed as valid JSON. Toxicity or cytometry data with integer values (e.g., '1', without quotation marks)	Review the preparation of Cello Input files (§6.1 in Supplementary Manual) and convert integer values to float data type (e.g., '1.00'. without quotation marks).
Project Submission	*Project name is empty. Please fill out project name.*	The user did not enter in a project name during project submission.	See Figure 5 and Step 7 of the Procedure section. Specifying a project title is not optional.
Project Submission	*Project description is empty. Please fill out project description.*	The user did not enter in a project description during project submission.	See Figure 5 and Step 7 of the Procedure section. Specifying a project description is not optional.

Table 3: Troubleshooting. Note that the *Step* column indicates the source of the error, but the error will appear after project execution.

Anticipated results

General

The expected result of a Cello 2.0 project is a genetic circuit design that implements the behavior in the Verilog file provided by the user. The software generates multiple images and text files that describe the circuit design and its expected behavior. Table 4 gives a complete listing of all the output files generated by Cello. Figure 8 contains a combination of the visual results presented in the GUI. It will generate a truth table for the Boolean circuit as well as expected values of transcriptional activity in the signal carrier unit (RPU or a variant for any standard library) for every gate and every row in the truth table. Complete distributions of expected transcriptional activity for each gate are also drawn. The topology of the circuit is represented in two visualizations: a graph where each node represents a gate and is annotated with the Boolean gate type and the implementing biological gate and a sequence diagram showing the ordering of the parts in the circuit and the interactions between them. A response curve for each gate is also drawn along with a table of expected toxicities for each state of the circuit. All the results are illustrated in the GUI (see example in Figure 8) and can be downloaded for offline use.

The SBOL file is one of the most important outputs as it contains the fully annotated DNA sequence. SBOL files can be processed in libraries available in several programming languages and are generally intended to be stored in a SynBioHub repository. A SynBioHub instance can run database queries over its contents and serve as a distributor of circuit construct data in automated software pipelines.

<*project name*>_<*stage*>.{dot,pdf,png}	An image of the netlist after each stage. This will include a gate name, Boolean gate type (e.g., NOR), and biological gate for each node in the netlist. See Figure 8 **b**. Example: andgate_technologyMapping.png
<*project name*>_outputNetlist.json	The netlist encoded in JSON format.
<*project name*>.xml	The SBOL file representing the circuit.
<*project name*>_logic.csv	The truth table of the circuit.
<*project name*>_activity.csv	A table of the predicted transcriptional activity[***] in RPU of each node in the circuit.
<*project name*>_toxicity.csv	A table of the predicted optical density for a cell carrying the circuit. "Toxicity" is sometimes used to describe a gate's or circuit's load on the host cell's growth, as indicated by an optical density measurement. Plotted in Figure 8 **d**.
<*project name*>_dpl.{pdf,png}	An image of the final circuit sequence generated by DNAplotlib. See Figure 8 **e**.
dpl_{dna_designs,part_information, regulatory_information}.csv, plot_parameters.csv	The inputs to DNAplotlib's library_plot.py.

[***] The meaning and accuracy of the transcriptional activity and toxicity predictions are discussed in more detail in the Supplementary Manual (§7.4.2).

response_plot_<*node*>_<*gate*>.{png,py}	A response plot and the Python script used to generate it. There is a response plot for each node in the netlist. The response function is marked with the different states the gate is expected to take. See Figure 8 **c**.
cytometry_plot_<*node*>_<*gate*>.{png,py}	A plot of the expected cytometry profile of the circuit and the Python script used to generate it. There is a cytometry plot for each node in the netlist. See Figure 8 **a**.
<*project name*>_eugeneScript.eug	The Eugene program that is executed to find rules-compliant DNA sequences.
<*project name*>.ys	The Yosys script used to synthesize the Boolean layout from the Verilog file.

Table 4: Output files generated by Cello. See Supplementary Manual §5 for complete examples of some of these files.

Case study

A collection of the anticipated results from the Case study are given in Figure 8, and additional results are presented in the Supplementary Manual §6. Along with the circuit layout, a predicted cytometry distribution is shown for each possible input state of the resultant circuit. The distribution for input state "+/+/+" is centered around a higher RPU output than the distributions for the other states, consistent with the expected output (three-input AND). Toxicity, as given by cell growth measured via relative OD600, is also shown to be consistently above a threshold value of 0.7 in the case study results. Several different variants of gate ordering (DNA sequence) are given, as Cello 2.0 does not model the effect of such orderings.

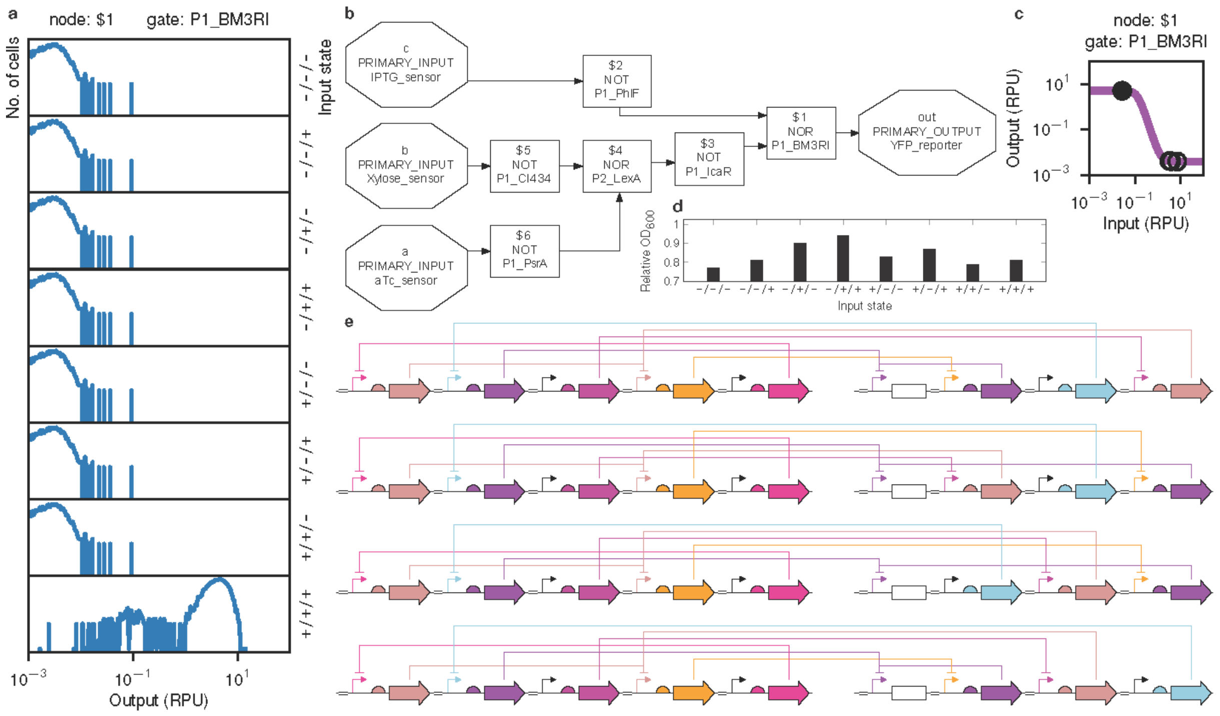


Figure 8: The expected results of a case study aiming at designing the circuit 0x01 (three-input AND gate) using the SC1C1G1T1 library. **a** The computational predictions of the cytometry profiles given the gate assignments in **b**. The right-hand y-axis label indicates the input state corresponding to the subplot. For example, +/-/+ indicates that inputs a and c are ON, input b is OFF. **b** The circuit network for the 0x01 circuit in *S. cerevisiae*. Each node is marked with a node name, e.g., $1, a Boolean gate type, e.g., NOR, and a biological gate assignment, e.g., P1_PhlF. **c** The Hill function response of the final gate (named "$1") before the output. Closed circle markers indicate ON states, and open circle markers indicate OFF states. **d** The relative OD_{600} of the circuit, defined for each state. **e** Four sequence variants of the circuit layout.

Data availability

The data that support the findings of this study (i.e., standard UCF libraries, input sensor files, and output device files) is openly available at <https://doi.org/10.5281/zenodo.4676314>.

Code availability

Code for Cello 2.0 is divided among various openly available repositories. The core circuit design module is available at <https://doi.org/10.5281/zenodo.4676314>, the code for the web application is available at <https://doi.org/10.5281/zenodo.4676310>, the source code of the GUI (which is compiled into the webapp) is available at <https://doi.org/10.5281/zenodo.4676300>. A supplementary file (in .zip format), containing all the source code in the version used in the study, associated test data, parameters, and documentation, is openly available at <https://publication-artifacts.s3.amazonaws.com/cellov2.zip>. The source code is openly distributed in accordance with Boston University's Data Protection Standards <https://www.bu.edu/policies/data-protection-standards/pdf/>, and under the MIT license at <https://opensource.org/licenses/MIT>.

References

1. Cheng, A. A. & Lu, T. K. Synthetic Biology: An Emerging Engineering Discipline. *Annu. Rev. Biomed. Eng.* **14**, 155–178 (2012).
2. Hasty, J., McMillen, D. & Collins, J. J. Engineered gene circuits. *Nature* **420**, 224–230 (2002).
3. Mano, M. M. & Ciletti, M. D. *Digital Design: With an Introduction to the Verilog HDL, VHDL, and System Verilog*. (Pearson, 2018)
4. Rabaey, J. M., Chandrakasan, A. P. & Nikolić, B. *Digital Integrated Circuits: A Design Perspective*. (Prentice Hall, 2008).
5. Brophy, J. A. N. & Voigt, C. A. Principles of genetic circuit design. *Nat. Methods* **11**, 508–520 (2014).
6. Nielsen, A. A. K. *et al.* Genetic circuit design automation. *Science* **352**, aac7341 (2016).
7. Andrews, L. B., Nielsen, A. A. K. & Voigt, C. A. Cellular checkpoint control using programmable sequential logic. *Science* **361**, eaap8987 (2018).
8. Bilitchenko, L. *et al.* Eugene – A Domain Specific Language for Specifying and Constraining Synthetic Biological Parts, Devices, and Systems. *PLOS ONE* **6**, e18882 (2011).
9. Khalil, A. S. & Collins, J. J. Synthetic biology: applications come of age. *Nat. Rev. Genet.* **11**, 367–379 (2010).
10. Endy, D. Foundations for engineering biology. *Nature* **438**, 449-453 (2005).
11. Bueso, Y. F., & Tangney, M. Synthetic biology in the driving seat of the bioeconomy. *Trends Biotechnol.* **35**, 373-378 (2017).
12. Purnick, P. E., & Weiss, R. The second wave of synthetic biology: from modules to systems. *Nat. Rev. Mol. Cell Biol.* **10**, 410-422 (2009).
13. Woodruff, L. B. et al. Registry in a tube: multiplexed pools of retrievable parts for genetic design space exploration. *Nucleic Acids Res.* **45**, 1553-1565 (2017).

14. Hossain, A. et al. Automated design of thousands of nonrepetitive parts for engineering stable genetic systems. *Nat. Biotechnol.* **38**, 1466-1475 (2020).

15. Huynh, L., Tsoukalas, A., Köppe, M. & Tagkopoulos, I. SBROME: A Scalable Optimization and Module Matching Framework for Automated Biosystems Design. *ACS Synth. Biol.* **2**, 263–273 (2013).

16. Yaman, F., Bhatia, S., Adler, A., Densmore, D. & Beal, J. Automated Selection of Synthetic Biology Parts for Genetic Regulatory Networks. *ACS Synth. Biol.* **1**, 332–344 (2012).

17. Beal, J., Lu, T. & Weiss, R. Automatic Compilation from High-Level Biologically-Oriented Programming Language to Genetic Regulatory Networks. *PLOS ONE* **6**, e22490 (2011).

18. Czar, M. J., Cai, Y. & Peccoud, J. Writing DNA with GenoCAD^{TM}. *Nucleic Acids Res.* **37**, W40–W47 (2009).

19. Chen, J., Densmore, D., Ham, T. S., Keasling, J. D., & Hillson, N. J. DeviceEditor visual biological CAD canvas. *J. Biol. Eng.* **6**, 1-12 (2012).

20. Roehner, N., & Myers, C. J. Directed acyclic graph-based technology mapping of genetic circuit models. *ACS Synth. Biol.* **3**, 543-555 (2014).

21. Salis, H. M. The ribosome binding site calculator. *Methods Enzymol.* **498**, 19-42 (2011).

22. Rodrigo, G., Carrera, J. & Jaramillo, A. Genetdes: automatic design of transcriptional networks. *Bioinformatics* **23**, 1857–1858 (2007).

23. Wilson, E. H., Macklin, C., & Platt, D. Engineering Genomes with Genotype Specification Language. *In Synthetic Biology*. Humana Press, New York, NY 373-398 (2018).

24. Pedersen, M., & Phillips, A. Towards programming languages for genetic engineering of living cells. *J. R. Soc. Interface* **6**, S437-S450 (2009).

25. Wolf, C. *Yosys Open Synthesis Suite* (2016).

26. Vaidyanathan, P. *et al.* A Framework for Genetic Logic Synthesis. *Proc. IEEE* **103**, 2196–2207 (2015).

27. Roehner, N. *et al.* Sharing Structure and Function in Biological Design with SBOL 2.0. *ACS Synth. Biol.* **5**, 498–506 (2016).

28. McLaughlin, J. A. *et al.* SynBioHub: A Standards-Enabled Design Repository for Synthetic Biology. *ACS Synth. Biol.* **7**, 682–688 (2018).

29. Shin, J., Zhang, S., Der, B. S., Nielsen, A. A. & Voigt, C. A. Programming Escherichia coli to function as a digital display. *Mol. Syst. Biol.* **16**, e9401 (2020).

30. Park, Y., Espah Borujeni, A., Gorochowski, T. E., Shin, J. & Voigt, C. A. Precision design of stable genetic circuits carried in highly-insulated E. coli genomic landing pads. *Mol. Syst. Biol.* **16**, e9584 (2020).

31. Taketani, M. *et al.* Genetic circuit design automation for the gut resident species Bacteroides thetaiotaomicron. *Nat. Biotechnol.* **38**, 962–969 (2020).

32. Chen, Y. *et al.* Genetic circuit design automation for yeast. *Nat. Microbiol.* **5**, 1349–1360 (2020).

33. Der, B. S. *et al.* DNAplotlib: Programmable Visualization of Genetic Designs and Associated Data. *ACS Synth. Biol.* **6**, 1115–1119 (2017).

Acknowledgements

This work was supported by funding from DARPA award FA8750-17-C-0229 “Synergistic Discovery and Design (SD2)” (D.D., C.A.V., C.J.M., and T.S.J.), DARPA award HR011-12-C-0067 “Living Foundries: 1000 Molecules” (C.A.V), US Department of Energy award DE-SC0018368 (C.A.V.), NSF Synthetic Biology Engineering Research Center SA5284-11210 (C.A.V.), and NSF Grant No. 1522074 “The Living Computing Project” (D.D., C.J.M., T.S.J.). The authors also acknowledge the support of William Jackson from D.D.'s CIDAR Lab in reviewing the manuscript.

Author information

Contributions

C.A.V. and D.D. conceived and supervised the project. C.J.M. provided supervision and technical assistance with SBOL and SynBioHub integrations. W.J. provided programming assistance with Cello 2.0 software. S.M.D.O assisted with designing the case study and input-files. T.S.J. wrote the Cello 2.0 software. All authors read and revised the manuscript.

Ethics declarations

Competing interests

Douglas Densmore and Christopher A. Voigt are co-founders of Asimov, Inc. Asimov is a company that uses software to engineer biology.