nature protocols

PROTOCOL

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

Check for updates

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]✉

Cells interact with their environment, communicate among themselves, track time and make decisions through functions controlled by natural regulatory *genetic circuits* consisting of interacting biological components. Synthetic programmable circuits used in therapeutics and other applications can be automatically designed by computer-aided tools. The Cello software designs the DNA sequences for programmable circuits based on a high-level software description and a library of characterized DNA parts representing Boolean logic gates. This process allows for design specification reuse, modular DNA part library curation and formalized circuit transformations based on experimental data. This protocol describes Cello 2.0, a freely available cross-platform software written in Java. Cello 2.0 enables flexible descriptions of the logic gates' structure and their mathematical models representing dynamic behavior, 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. Designing circuits with Cello 2.0 produces an abstract Boolean network from a Verilog file, assigns biological parts to each node in the Boolean network, constructs a DNA sequence and generates highly structured and annotated sequence representations suitable for downstream processing and fabrication, respectively. The result is a sequence implementing the specified Boolean function in the organism and predictions of circuit performance. Depending on the size of the design space and users' expertise, jobs may take minutes or hours to complete.

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 (to reduce complexity), modularity (to subdivide the system into parts), standards (to create a data structure) and modeling (to repeatedly create the same patterns). A particular design approach is to use composable DNA elements to design functional, genetic circuits that can robustly control desired biological tasks. Namely, synthetic biology and rapid, automated design of genetic circuits promise to produce advances in biosensing[2], smart therapeutics[3], biofuels[4] and biomaterials[5]. 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[6,7]. This transformation process is quite powerful not only because it is efficient but also because it is provably functionally correct, removing many errors and suboptimizations that would have resulted in a manual process.

Artificial *genetic circuits* are engineered regulatory networks that accept a set of inputs, and produce the desired output response. One class of networks is based on transcriptional control that generate output signals in response to sensed inputs, e.g., chemicals and temperature pertubations[8], light stimuli[9] and intercellular signals[10]. The transcription of a gene is controlled by the interaction of its promoter region and RNA polymerase, repressors and activators, among other transcriptional factors[11]. In synthetic circuits, such regulatory elements can be rationally designed with the support of software tools, such as Cello version 1.0[12], and their functionality in live cells tested and validated using, e.g., fluorescent probes[12–14] and flow cytometry[12,14].

In the Cello tool, the digital logic design abstraction can be adapted to transcriptional genetic circuits[11] 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

[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. ✉e-mail: dougd@bu.edu

Box 1 | 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 0 × 01, 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 main Procedure.
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 (Fig. 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 15 and 16 in the main Procedure.

constructed around a specific genetic product, the binary on and off states are not uniform over all gates. The first implementation of the Cello software design framework (Cello 1.0) used this modified digital logic design paradigm to design genetic circuits in *E. coli*[12]. 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, DNA sequences (captured as modular 'parts'), 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[14].

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 (where transcriptional start sites in close proximity interfere with each other via RNA polymerase competition). The design rules[15] 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. 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 Materials and Procedure sections provide pragmatic instructions on using the software and its resources, including a case study on designing a circuit in *Saccharomyces cerevisiae* (Box 1). The case study leads the user through the circuit design process in the graphical user interface (GUI) 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 and 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 2.0.

Development of Cello 2.0

Several approaches to the automated design of multiple genetic regulatory networks, for exploring the design space[16] (the space created by the large combinations of modular DNA parts) and engineering

stable systems[17] (systems that act predictably over time), are documented in the literature. A few of these are SBROME[18], MatchMaker[19], Proto & BioCompiler[20], GenoCAD[21], Device Editor[22], iBioSim[23], RBS Calculator[24] and Genetdes[25]. In addition, novel programming languages, such as GSL[26] and GEC[27], have been developed to provide additional high-level genetic circuit specification environments focused on either desired experimental assembly processes or composite circuit behavior. Each of these approaches 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 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 descriptive output format is also important for design verification.

Cello version is distinct from the abovementioned circuit design tools in that there is a separation between the gate technology-independent phases in the design process and the technology-dependent phases. Independent phases manipulate pure Boolean algebraic representations, while dependent phases manipulate biology mapped to those functions. A technology-independent Boolean network is optimized first, and then biological gates are mapped to that network. Cello's UCF library format captures 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. Cello users write code in the Verilog language to compile arbitrary combinational logic into genetic circuits. Cello has been used to develop genetic circuits that allow majority voting[12], environmental biosensors[3] and proof-of-concept circuits to test the limits of genetic computation[5].

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 (Fig. 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. 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 UCF that 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 UCFs and input and output files that can be selected. To submit a batch of jobs programmatically for a parallel analysis of multiple Verilog files and Input files at once, more ambitious or advanced users may prepare their own UCFs and load them into the web application, utilize the RESTful HTTP application programming interface (API), run their instance of the Cello web application or use Cello 2.0 locally from the command line.

The Cello workflow is divided into four stages (Fig. 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, as explained in more detail below. 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. In the LS stage, Cello 2.0 uses an open-source LS framework called Yosys[28] to produce a Boolean gate network (netlist) based on a Verilog description of the desired circuit function. The process of compiling Verilog to a gate-level network has been studied at length[6,7].

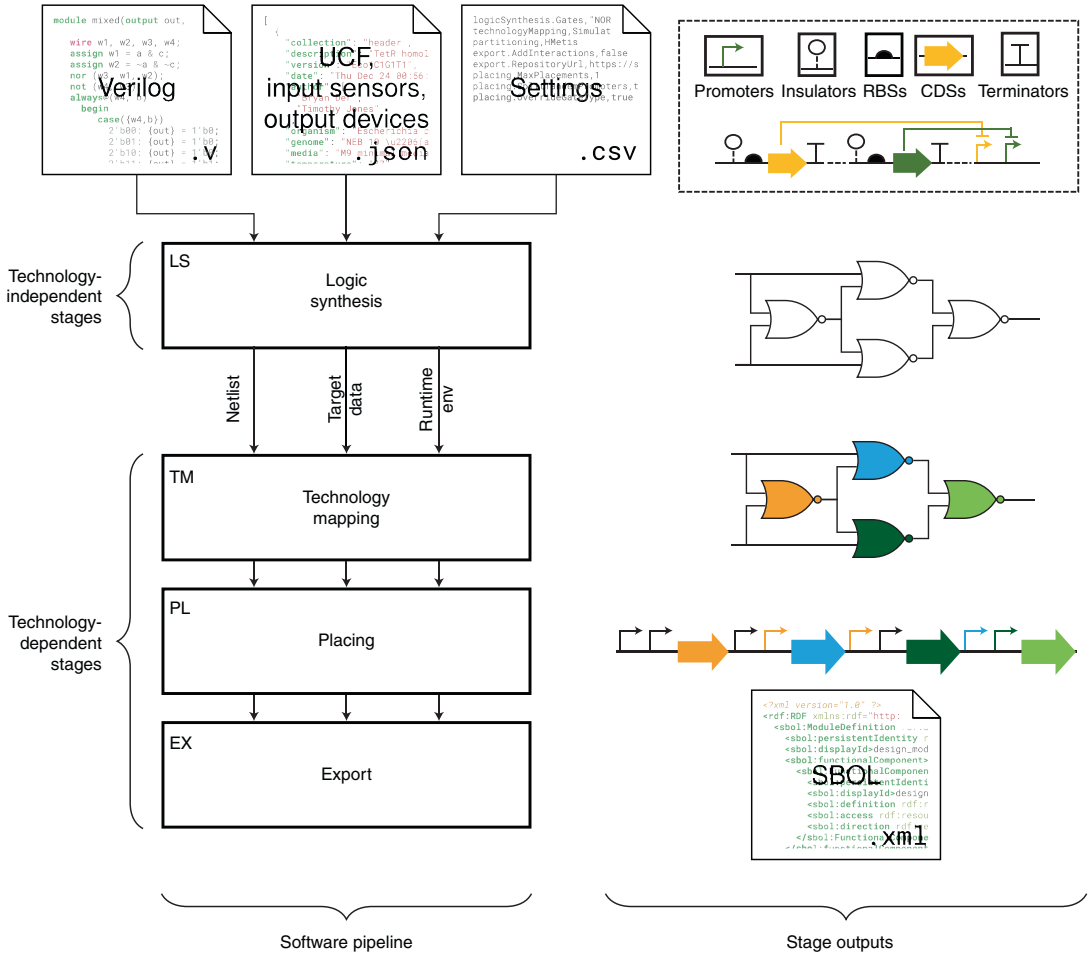
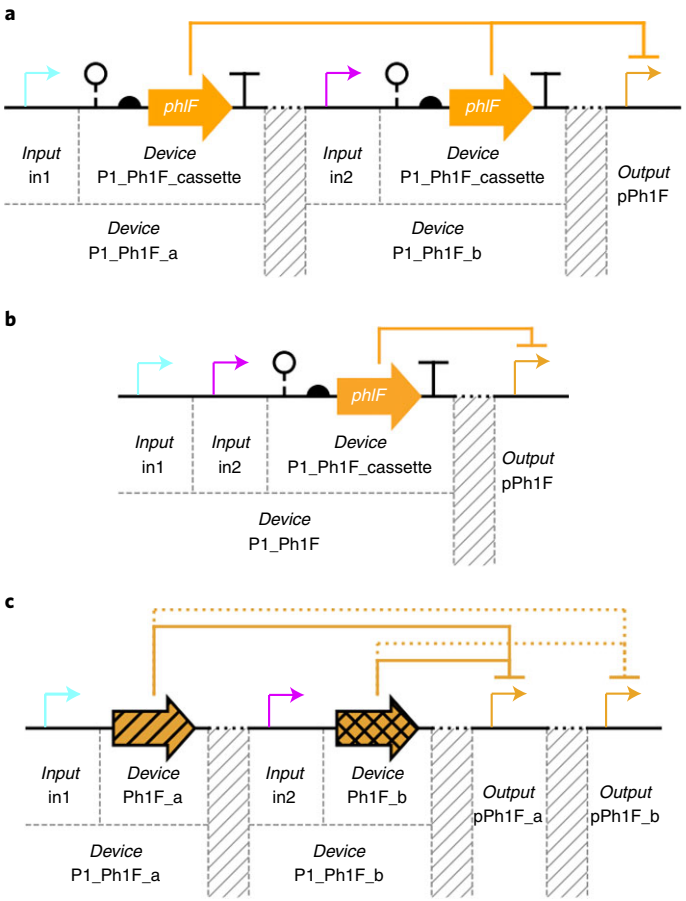


Fig. 1 | Basic software architecture of Cello 2.0. The workflow is divided into four stages, each with one or more algorithms to perform that stage's tasks. Cello ingests different input files: a Verilog file, the 'target data' (a composition of the UCF, the input sensor file and the output device file), and a stage configuration file. Each stage produces some characteristic output: the LS stage produces a Boolean gate-level network ('netlist') that implements the circuit behavior described by the Verilog code and the 'runtime environment' (env) of the synthesis, the TM stage assigns biological gates to each Boolean gate from the previous stage, the PL stage orders gates and parts into a linear DNA sequence and the EX stage produces interchange files (SBOL). In the dashed-line box, the legend shows the different modular components ('DNA parts') that compose Cello's transcriptional units ('DNA logic gates'), i.e., promoters, insulators, ribosome binding sites (RBSs), coding sequences (CDSs) and terminators.

Technology mapping. In the TM stage, biological gates that have been encoded in a UCF and that implement the Boolean operations in the result of the LS stage are assigned to every node in the netlist. The gate assignment process is equivalent to a graph coloring problem[29] 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[12]. After the TM stage, all the biological parts that implement the desired circuit function are determined.

Placing. The PL 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[15].

Export. Finally, in the EX stage, a circuit description file is created. The file is encoded using the Synthetic Biology Open Language (SBOL) data standard[30], which allows 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[31] parts repository where they can be assigned version numbers and grouped into collections.



d

```
{
  "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"
      ]
    } ]
}
```

Fig. 2 | 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 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 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 Supplementary Manual §2.9.2.

Gate architecture specification

The new UCF format has a structured JSON object type that 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 Fig. 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.

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 a defined 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 (Fig. 3), but parameters are resolved by pointers to values defined on the gate in which the Hill function is being

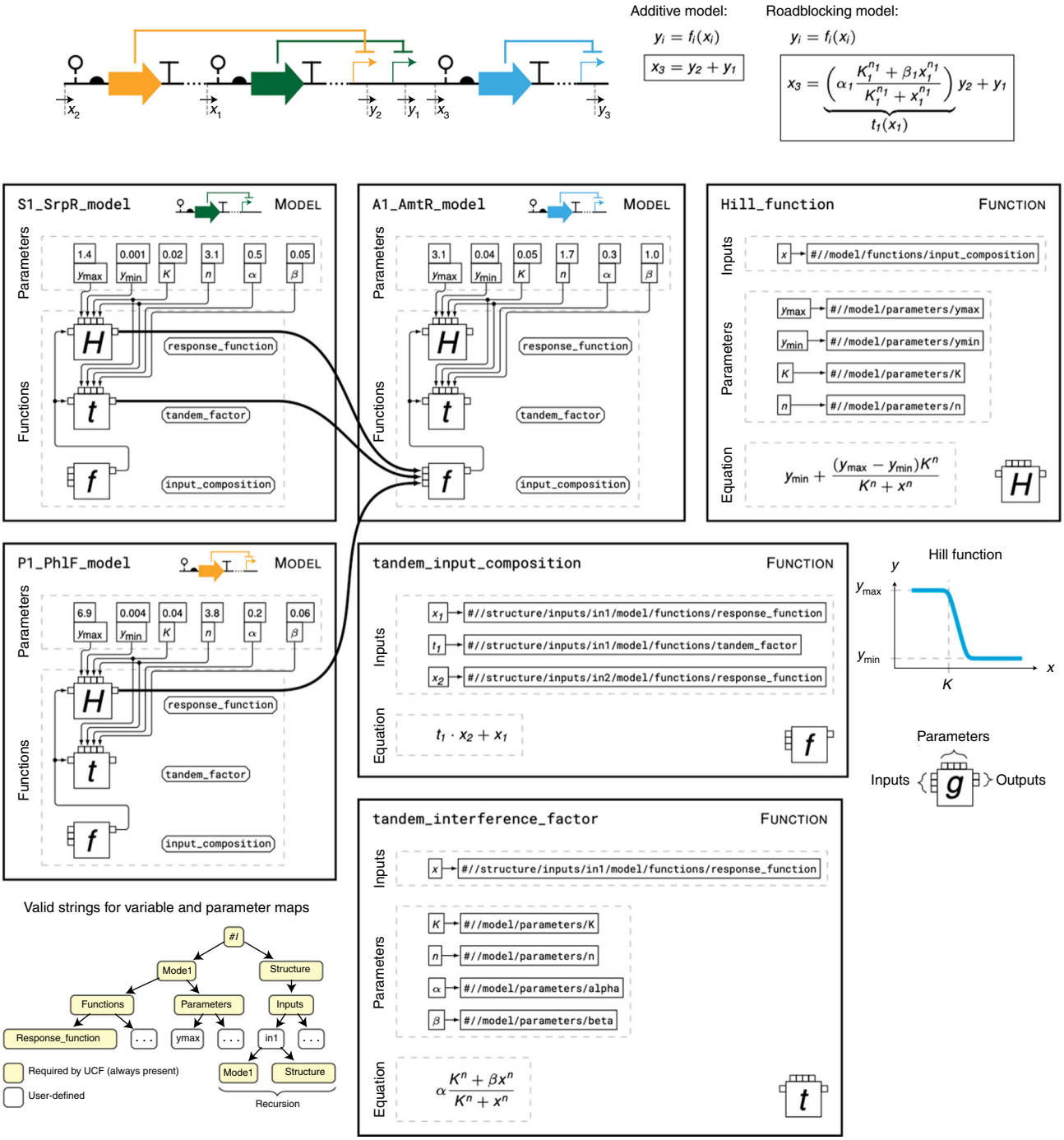


Fig. 3 | Visualization of gate model and function definitions. In all available UCFs, the gates' input/output characteristics are described by Hill functions. The Eco1C2G2T2 library uses a special roadblocking model to compute the input to the Hill function. The function used to compute the input is parameterized by each gate and referred to in the Hill function definition by a pointer: #//model/functions/input_composition.

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 that 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 *E. coli* UCF Eco2C1G3T1, 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 who design their own UCFs are free to specify their own signal carrier unit, as described in 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 datapoints of 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[32]. 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 Fig. 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 (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 Cello libraries, 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 nonadditive or synergistic toxicity produced by specific combinations of gates, though support for this feature may be added in future versions of Cello.

DNA mapping with placement rules

Cello uses Eugene rules[15] 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 Fig. 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. For all organisms, insertions are molecular cloning sites, at the DNA level, which can physically be implemented by molecular biology protocols, such as conventional cloning using restriction enzymes and isothermal recombination by Gibson assembly, among others (check the review papers refs. [2,3,11]). 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" }}.

SBOL and SynBioHub

After a successful project run, Cello 2.0 generates an encoding of the genetic circuit structure using the SBOL data standard[30] in the EX stage. SBOL is a data standard that can describe a genetic construct in a hierarchical fashion, also encoding information like molecular interactions and

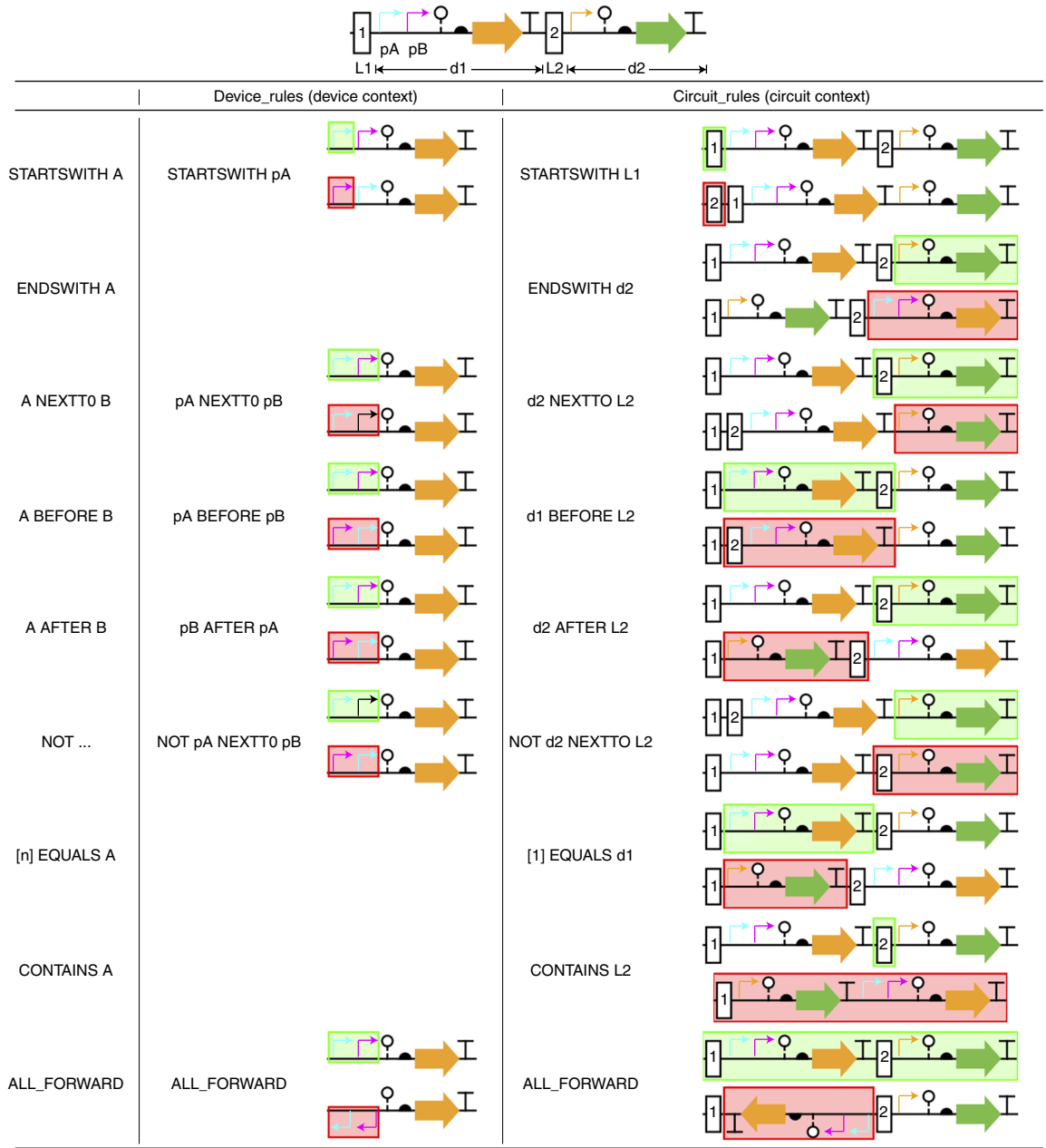


Fig. 4 | A selection of Eugene rules and circuits that satisfy or violate the rules. Cello 2.0 accepts rules in the UCF to control the placement of input parts within gates (*device_rules*) and gates within a circuit (*circuit_rules*). Rules in the *circuit_rules* collection can include the location markers (L1 or L2 in this 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.

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[31] 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 using both a GUI and an 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

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

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 relevant for future genetic circuits

libraries have been converted to SBOL format and uploaded to a SynBioHub instance (https://synbiohub.programmingbiology.org).

Verilog

Cello 2.0 uses the open-source LS tool Yosys[28], 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 exploration of many existing Verilog implementations to be tested quickly in Cello 2.0. See Table 1 for examples of various types of Verilog syntax supported in Cello 2.0.

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 (Fig. 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 (Fig. 6) to view the final circuit design and its predicted behavior.

Support for new organisms

Cello 2.0 supports new organisms with an updated UCF format and five ready-made UCFs in different hosts. After beginning a design in the Cello web application 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 Fig. 7. A complete description of the UCF format that enabled these libraries is included in Supplementary Manual §2.

Materials

Equipment

- Personal computer with an internet connection and a web browser such as Mozilla Firefox, Google Chrome or Microsoft Edge with JavaScript enabled
- A UCF describing the gates and the target organism. Complete files for gates in *S. cerevisiae*, *E. coli* and *Bacteroides thetaiotaomicron* are included with the software and are available on GitHub (https://github.com/CIDARLAB/Cello-UCF/tree/develop/files/v2/ucf) and http://cellocad.org. Information on creating a custom UCF is given in Supplementary Manual §2 and §4
- An input sensor file and output device file. Complete files for inputs and outputs in *S. cerevisiae*, *E. coli* and *B. thetaiotaomicron* are shipped with the software and are available on GitHub (https://github.com/CIDARLAB/Cello-UCF/tree/develop/files/v2/input, https://github.com/CIDARLAB/Cello-UCF/tree/develop/files/v2/output) and http://cellocad.org
- (Optional) Docker (https://www.docker.com/) if you wish to run your own instance of Cello 2.0. The webapp container is available on Docker Hub as (https://hub.docker.com/r/cidarlab/cello-webapp)

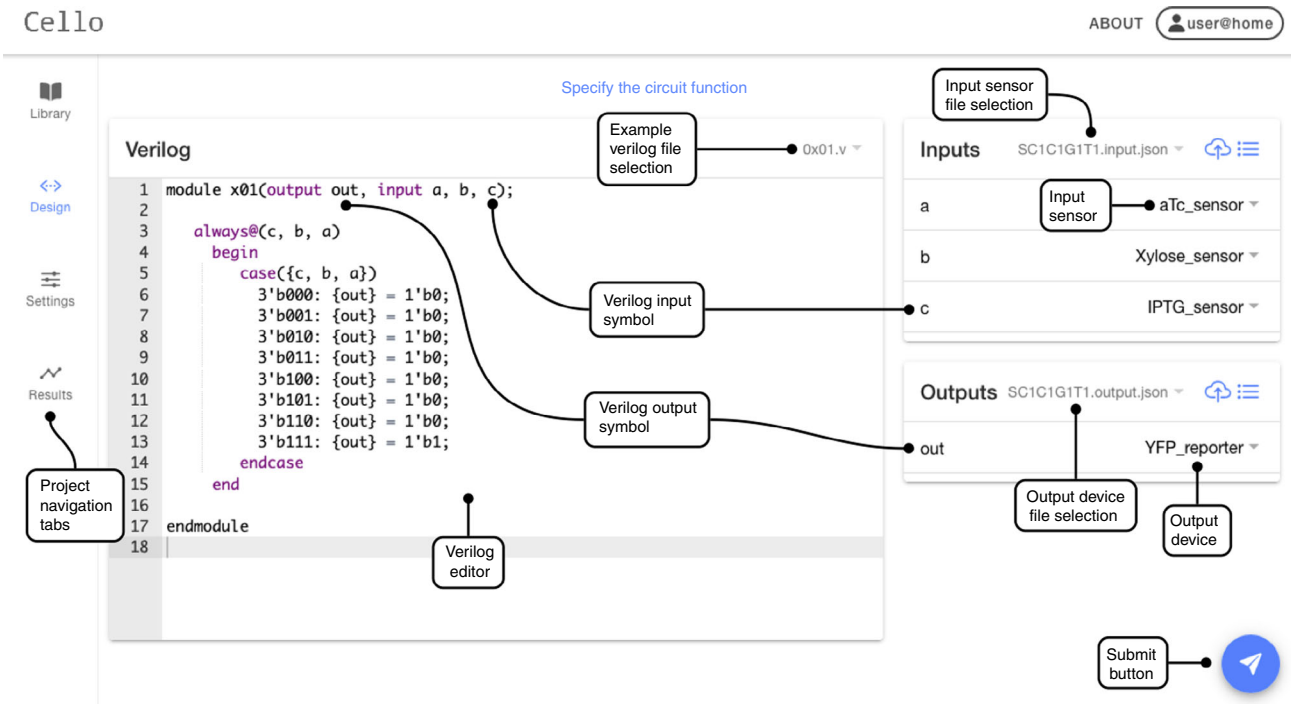


Fig. 5 | The Verilog editor and input and output selection panes in the GUI. In a project specification, the user navigates between this view and the library selection, advanced settings and results views using tabs on the left-hand side. The user can choose from a few example Verilog files. Input and 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.

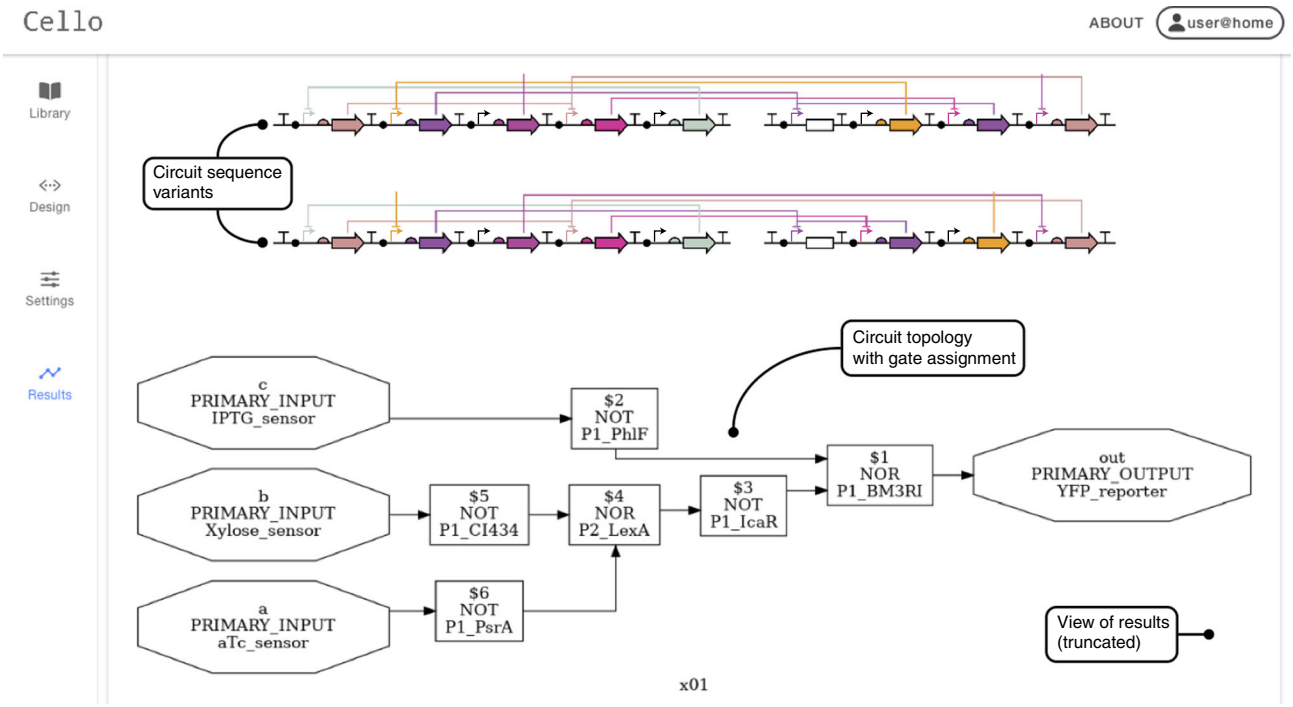


Fig. 6 | The results view in the GUI. The circuit topology is shown in the graph at the bottom. Each node in the network has three pieces of information: the node name, the Boolean gate type, e.g., NOT, NOR, etc., and the biological gate assignment. Many gate placements (orderings of gates into a DNA sequence) may be generated; two are visible in this screenshot. Many other results (not pictured) 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 as well as a prediction of optical density measurements.

Table 2 | Cello 2.0 is bundled with five different libraries in three organisms

Library name	Host	Regulators	Notes and 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[12]	Plasmid: p15a (circuit and 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[32]	Plasmid: p15a	9 NOT/NOR	Nonadditive tandem roadblocking	Yes	RPU
Eco2C1G3T1	*E. coli* MG1655 K-12	TetR homologs: PhlF, QacR, BM3R1, BetI, AmtR, AmeRs	Split transcriptional units[35]	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)[36]	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[37]	Genome: ura3, leu2	9 NOT/NOR	Additive	Yes	RPU

Each library contains a different set of gates. The library names follow a naming convention, see Supplementary Manual §2.1. Note that '7 NOT/NOR' in the 'Gate definition' column indicates that seven gates are defined in the UCF, each of which 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'). 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.

- (Optional) Java Runtime Environment (https://java.com/en/download/) version 8, Yosys[28] (https://github.com/YosysHQ/yosys), Python (https://www.python.org/) version 3, and dnaplotlib[33] (https://github.com/VoigtLab/dnaplotlib), if you wish to run an instance of Cello 2.0 in your host operating system outside of a Docker container

Procedure

Preliminary setup ● Timing 15–65 s

1 Navigate to http://cellocad.org in your web browser.
2 Create a new account or log in with an existing account.
3 Download results from a previous project or create a new project.

Selecting UCF files ● Timing 15–65 s

4 Choose an available gate library or upload an already prepared UCF.

Alternatively, preparing and uploading custom UCF files ● Timing 40–80 h

▲ **CRITICAL** Preparing Cello input files requires a series of steps, depending on whether characterization data of the gates are available or one needs to run experiments from scratch. See instructions in Supplementary Manual §6.1.

5 To start, find the closest set of files to the one being designed following the criteria of: promoter arrangement (i.e., tandem or split), gene location (i.e., plasmid or genome-integrated) and Chassis-type (e.g., bacteria or yeast).
6 To download template files, navigate to https://github.com/CIDARLAB/Cello-UCF.
7 In each input file, use a text editor to change the information that already exists to the new one following the steps described in Supplementary Manual §2, §3 and §4. When creating new input files, if suitable, keep the types of motifs with EUGENE rules and circuits intact to avoid errors due to writing mistakes.
8 Prepare toxicity measurements (e.g., OD_{600} measurements by plate readers) and fluorescence activity (e.g., by cytometry data)[12,14], based on the transcription dynamics of promoters in different arrangements[34]. The number of datapoints 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 datapoints allowed.

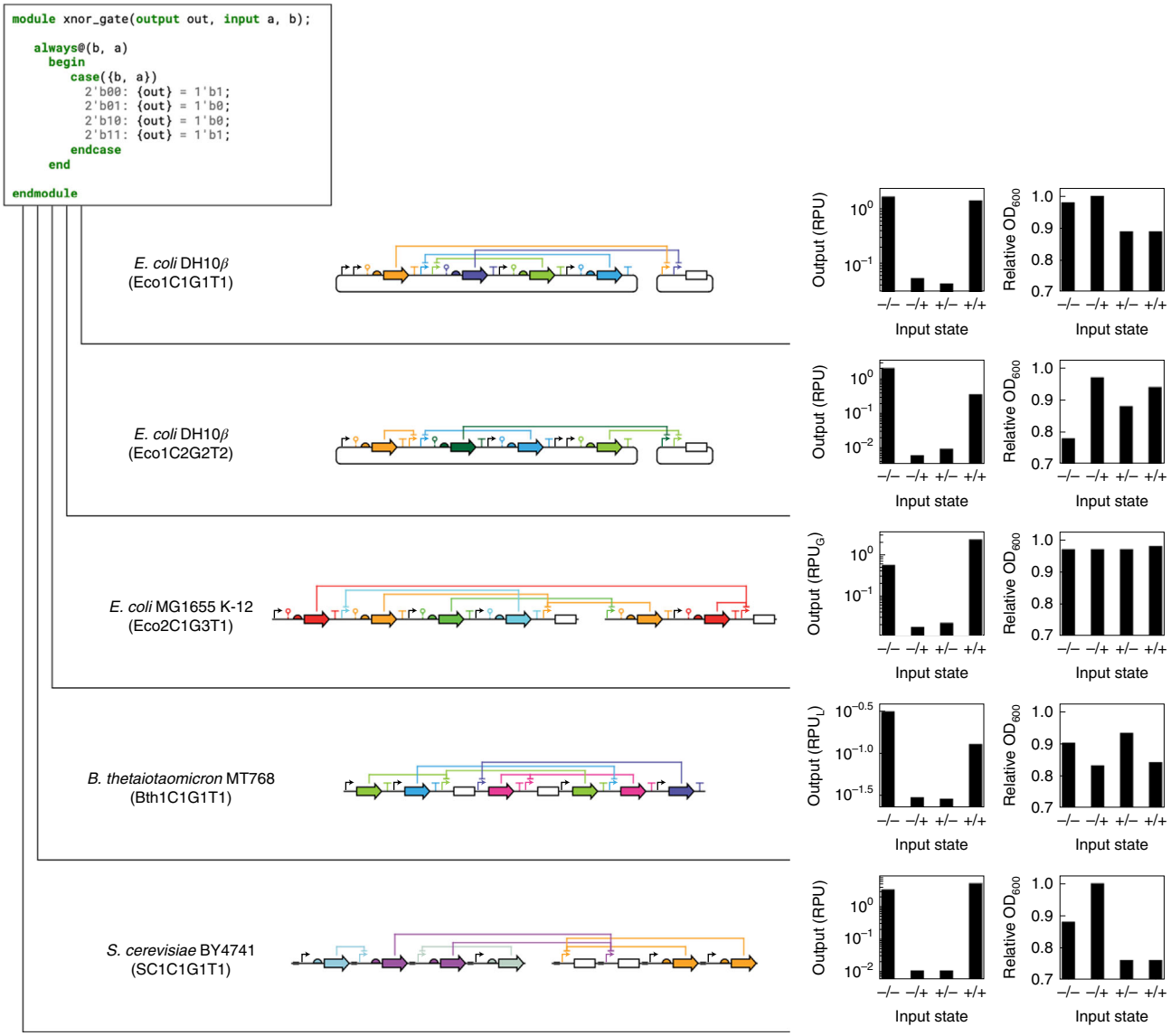


Fig. 7 | An XNOR gate designed in all five libraries available in Cello 2.0. The gate is shown with predicted (not experimental) OD_{600} and RPU output levels.

9 Upload custom UCF, input and output files onto Cello 2.0. Click on a filled-blue-cloud icon button (located on the top-right corner of the browser) to upload the (1) UCF library and the newly designed (2) input-sensor and (3) out-device files, following the instructions in Fig. 5.
▲ **CRITICAL STEP** Note that preparing a custom UCF may require 60–120 min from available measurement data, while running experiments usually requires an additional 40–80 h of effort for running and preparing characterization data for the number of gates intended.

Creating a Verilog description for your circuit ● Timing 15–65s

10 Users can opt to select an existing file (Fig. 5) or write their own files. Open the Verilog tab, and compose or paste the Verilog description of your circuit. 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.

11 To write a custom Verilog file, start from template files provided in §3 of Supplementary Manual and see Supplementary Manual §5 and the section 'Verilog' for more in-depth instructions. Additionally, there is a small number of sample Verilog files available for selection in the GUI. Note that users unfamiliar with Boolean logic and basic programming concepts may require additional time (1–30 min) to understand Verilog syntax.

Selecting input and output files ● Timing 2–20 min

12 In the Inputs section, under the file selection menu (Fig. 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 Supplementary Manual §3 for information on preparing a custom file. Preparing a custom input sensor file, including the necessary characterization experiments, may require additional 40–80 h of effort.

13 In the Outputs section, under the file selection menu (Fig. 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 Supplementary Manual §4 for information on preparing a custom file. Preparing a custom input sensor file, including the necessary characterization experiments, may require additional 40–80 h of effort.

14 (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.

Submitting job and exporting results ● Timing 6–20 min

15 Submit the job by clicking on the blue plane icon in the bottom corner, enter a job name and click run, then wait for the job to complete. Timing may vary depending on gate library and Verilog input.

16 Visualize results in the results tab. Optionally download results and submit SBOL description of circuit design to a SynBioHub instance.

Timing

Steps 1 and 2, navigate to cellocad.org and log in, and Step 3, create a new project: 15–65 s
Step 4, choose a gate library: 1–10 min
Steps 5–9, prepare and upload your own gate libraries: 40–80 h
Steps 10 and 11, create and enter Verilog files: 15–65 s
Steps 12–14, assign and select inputs and outputs files or upload your own: 2–20 min
Steps 15 and 16, submit the job and view results or submit to SynBioHub: 6–20 min

Troubleshooting

Table 3 describes the most common types of error messages that may occur in the GUI.

Table 3 | Troubleshooting table (the 'Step' column indicates the source of the error, but the error itself will appear after project execution)

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
	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 and 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
PL 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.,	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

Table continued

Table 3 (continued)

Step	Problem (error message)	Possible reason	Solution
		pTet BEFORE pTac AND pTac BEFORE pTET	create a contradiction. Optionally download the failed project on your projects page and open the Eugene script (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 Fig. 5 and Step 14 of the Procedure. 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 Fig. 5 and Step 15 of the Procedure. 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. 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 name is empty. Please fill out project name	The user did not enter in a project name during project submission	See Fig. 5 and Step 18 of the Procedure. Specifying a project title is not optional
	Project description is empty. Please fill out project description	The user did not enter in a project description during project submission	See Fig. 5 and Step 18 of the Procedure. Specifying a project description is not optional

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: (1) a graph where each node represents a gate and is annotated with the Boolean gate type and the implementing biological gate and (2) 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 Fig. 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.

Case study

A collection of the anticipated results from the Case study are given in Fig. 8, and additional results are presented in 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

Table 4 | Output files generated by Cello 2.0

*<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 (Fig. 8b). 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[a] 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. The results of this table are plotted in Fig. 8d
*<project name>*_dpl.{pdf,png}	An image of the final circuit sequence generated by DNAplotlib (Fig. 8e)
dpl_{dna_designs,part_information, regulatory_information}.csv, plot_parameters.csv	The inputs to DNAplotlib's library_plot.py
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 (Fig. 8c)
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 (Fig. 8a)
*<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

[a]The meaning and accuracy of the transcriptional activity and toxicity predictions are discussed in more detail in Supplementary Manual (§7.4.2). See Supplementary Manual §5 for complete examples of some of these files.

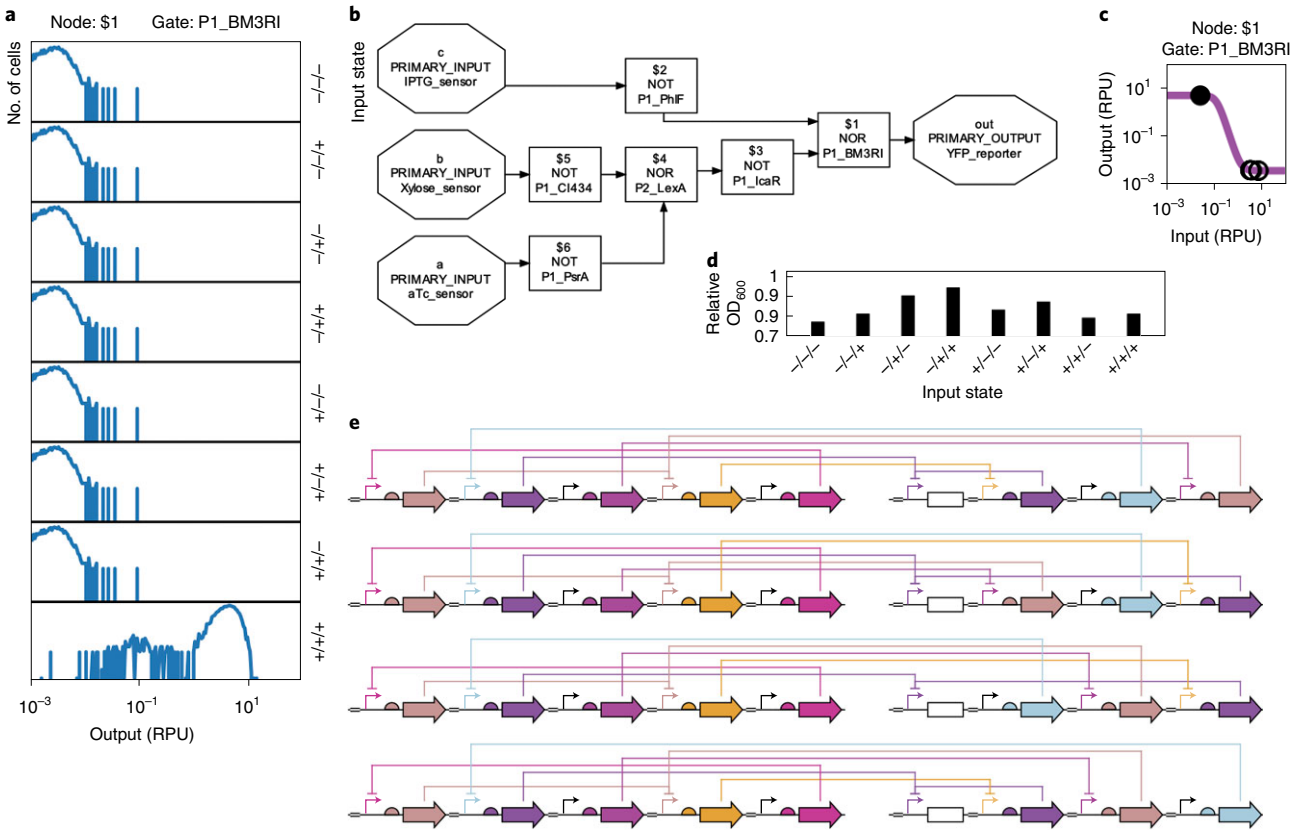


Fig. 8 | The expected results of a case study aiming at designing the circuit 0×01 (three-input AND gate) using the SC1C1G1T1 library. a, The computational predictions of the cytometry profiles given the gate assignments in **b**. The right-hand *y*-axis label indicates the input state corresponding to the subplot. For example, +/−/+ indicates that inputs a and c are ON and input b is OFF. **b**, The circuit network for the 0 × 01 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.

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 OD_{600}, 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.

Data availability

The data that support the findings of this study (i.e., standard UCF libraries, input sensor files and output device files) are 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 and the source code of the GUI (which is compiled into the webapp) is available at https://doi.org/10.5281/zenodo.4676300. A 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. Khalil, A. S. & Collins, J. J. Synthetic biology: applications come of age. *Nat. Rev. Genet.* **11**, 367–379 (2010).
3. Endy, D. Foundations for engineering biology. *Nature* **438**, 449–453 (2005).
4. Bueso, Y. F. & Tangney, M. Synthetic biology in the driving seat of the bioeconomy. *Trends Biotechnol.* **35**, 373–378 (2017).
5. Purnick, P. E. & Weiss, R. The second wave of synthetic biology: from modules to systems. *Nat. Rev. Mol. Cell Biol.* **10**, 410–422 (2009).
6. Mano, M. M. R. & Ciletti, M. D. *Digital Design: With an Introduction to the Verilog HDL, VHDL, and SystemVerilog* (Pearson, 2018).
7. Rabaey, J. M., Chandrakasan, A. P. & Nikolić, B. *Digital Integrated Circuits: A Design Perspective* (Prentice Hall, 2008).
8. Oliveira, S. M. D. et al. Temperature-dependent model of multi-step transcription initiation in *Escherichia coli* based on live single-cell measurements. *PLoS Comput. Biol.* https://doi.org/10.1371/journal.pcbi.1005174 (2016).
9. Tabor, J. J. et al. A synthetic genetic edge detection program. *Cell* **137**, 1272–1281 (2009).
10. Stephens, K. & Bentley, W. E. Synthetic biology for manipulating quorum sensing in microbial consortia. *Trends Microbiol.* **28**, 633–643 (2020).
11. Brophy, J. A. N. & Voigt, C. A. Principles of genetic circuit design. *Nat. Methods* **11**, 508–520 (2014).
12. Nielsen, A. A. K. et al. Genetic circuit design automation. *Science* **352**, aac7341 (2016).
13. Hasty, J., McMillen, D. & Collins, J. J. Engineered gene circuits. *Nature* **420**, 224–230 (2002).
14. Andrews, L. B., Nielsen, A. A. K. & Voigt, C. A. Cellular checkpoint control using programmable sequential logic. *Science* **361**, eaap8987 (2018).
15. Bilitchenko, L. et al. Eugene—a domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PLoS ONE* **6**, e18882 (2011).
16. 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).
17. Hossain, A. et al. Automated design of thousands of nonrepetitive parts for engineering stable genetic systems. *Nat. Biotechnol.* **38**, 1466–1475 (2020).
18. 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).
19. 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).
20. Beal, J., Lu, T. & Weiss, R. Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks. *PLoS ONE* **6**, e22490 (2011).
21. Czar, M. J., Cai, Y. & Peccoud, J. Writing DNA with GenoCADTM. *Nucleic Acids Res.* **37**, W40–W47 (2009).
22. 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).
23. Roehner, N. & Myers, C. J. Directed acyclic graph-based technology mapping of genetic circuit models. *ACS Synth. Biol.* **3**, 543–555 (2014).

24. Salis, H. M. The ribosome binding site calculator. *Methods Enzymol.* **498**, 19–42 (2011).
25. Rodrigo, G., Carrera, J. & Jaramillo, A. Genetdes: automatic design of transcriptional networks. *Bioinformatics* **23**, 1857–1858 (2007).
26. Wilson, E. H., Macklin, C., & Platt, D. Engineering genomes with genotype specification language. in *Synthetic Biology* 373–398 (Humana Press, 2018).
27. Pedersen, M. & Phillips, A. Towards programming languages for genetic engineering of living cells. *J. R. Soc. Interface* **6**, S437–S450 (2009).
28. Wolf, C. Yosys Open Synthesis Suite (2016).
29. Vaidyanathan, P. et al. A framework for genetic logic synthesis. *Proc. IEEE* **103**, 2196–2207 (2015).
30. Roehner, N. et al. Sharing structure and function in biological design with SBOL 2.0. *ACS Synth. Biol.* **5**, 498–506 (2016).
31. McLaughlin, J. A. et al. SynBioHub: a standards-enabled design repository for synthetic biology. *ACS Synth. Biol.* **7**, 682–688 (2018).
32. 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).
33. Der, B. S. et al. DNAplotlib: programmable visualization of genetic designs and associated data. *ACS Synth. Biol.* **6**, 1115–1119 (2017).
34. Häkkinen, A., Oliveira, S. M. D., Neeli-Venkata, R. & Ribeiro, A. S. Transcription closed and open complex formation coordinate expression of genes with a shared promoter region. *J. R. Soc. Interface* **16**, 20190507 (2019).
35. 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).
36. Taketani, M. et al. Genetic circuit design automation for the gut resident species Bacteroides thetaiotaomicron. *Nat. Biotechnol.* **38**, 962–969 (2020).
37. Chen, Y. et al. Genetic circuit design automation for yeast. *Nat. Microbiol.* **5**, 1349–1360 (2020).

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., T.S.J. and S.M.D.O.), 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. and S.M.D.O.). The authors also acknowledge the support of William Jackson from D.D.'s CIDAR Lab in providing programming assistance and reviewing the manuscript.

Author 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. S.M.D.O assisted with input files and UCF designs and preparation. T.S.J. wrote the Cello 2.0 software. All authors read and revised the manuscript.

Competing interests

D.D. and C.A.V. are co-founders of Asimov. Asimov is a company that uses software to engineer biology.

Additional information

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s41596-021-00675-2.

Correspondence and requests for materials should be addressed to Douglas Densmore.

Peer review information *Nature Protocols* thanks Mark Isalan and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 7 November 2020; Accepted: 29 November 2021;
Published online: 23 February 2022

Related links

Key references using this protocol

Nielsen, A. et al. *Science* **352**, aac7341 (2016): https://doi.org/10.1126/science.aac7341
Chen, Y. et al. *Nat. Microbiol.* **5**, 1349–1360 (2020): https://doi.org/10.1038/s41564-020-0757-2
Vaidyanathan, P. et al. *Proc. IEEE* **11**, 2196–2207 (2015): https://doi.org/10.1109/JPROC.2015.2443832

Key data used in this protocol

Taketani, M. et al. *Nat. Biotechnol.* **38**, 962–969 (2020): https://doi.org/10.1038/s41587-020-0468-5