An End-to-End Workflow for Engineering of Biological Networks from High-Level Specifications

Jacob Beal,*[†] Ron Weiss,[‡] Douglas Densmore,[¶,§] Aaron Adler,[†] Evan Appleton,[§] Jonathan Babb,[‡] Swapnil Bhatia,[¶] Noah Davidsohn,[‡] Traci Haddock,[¶] Joseph Loyall,[†] Richard Schantz,[†] Viktor Vasilev,[§] and Fusun Yaman[†]

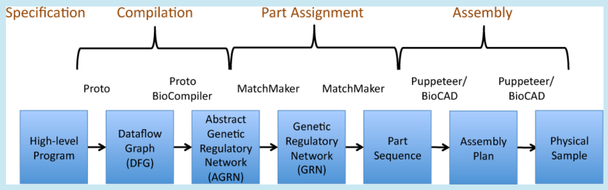
[†]Raytheon BBN Technologies, 10 Moulton St., Cambridge, Massachusetts, United States

[‡]Department of Biological Engineering, MIT, Cambridge, Massachusetts, United States

[¶]Department of Electrical and Computer Engineering and [§]Bioinformatics Department, Boston University, Boston, Massachusetts, United States

S *Supporting Information*

ABSTRACT: We present a workflow for the design and production of biological networks from high-level program specifications. The workflow is based on a sequence of intermediate models that incrementally translate high-level specifications into DNA samples that implement them. We identify algorithms for translating between adjacent models and implement them as a set of software tools, organized into a four-stage toolchain: *Specification*, *Compilation*, *Part Assignment*, and *Assembly*. The specification stage begins with a Boolean logic computation specified in the Proto programming language. The compilation stage uses a library of network motifs and cellular platforms, also specified in Proto, to transform the program into an optimized Abstract Genetic Regulatory Network (AGRN) that implements the programmed behavior. The part assignment stage assigns DNA parts to the AGRN, drawing the parts from a database for the target cellular platform, to create a DNA sequence implementing the AGRN. Finally, the assembly stage computes an optimized assembly plan to create the DNA sequence from available part samples, yielding a protocol for producing a sample of engineered plasmids with robotics assistance. Our workflow is the first to automate the production of biological networks from a high-level program specification. Furthermore, the workflow's modular design allows the same program to be realized on different cellular platforms simply by swapping workflow configurations. We validated our workflow by specifying a small-molecule sensor-reporter program and verifying the resulting plasmids in both HEK 293 mammalian cells and in *E. coli* bacterial cells.



KEYWORDS: *design tools, design automation, high-level design, BioCAD, compilation, part assignment, DNA assembly, specification*

Synthetic biology has tremendous potential for enabling the design and implementation of sophisticated biochemical mechanisms for sensing, computing, control, production, and actuation. This approach is anticipated to lead to many breakthrough applications in biotherapeutics, bioremediation, biomaterials, and *in vivo* sensing, actuation, and self-assembly—many of which are already being explored.[1−4] As these applications become more complex, however, there is an increasingly pressing need for tools to assist in design and in the production of organisms that implement those designs.

In this paper, we propose a workflow called TASBE (Toolchain to Accelerate Synthetic Biology Engineering), the first complete method for the design and production of physical biological networks from high-level specifications. This workflow is developed around a sequence of models that we have identified, moving from high-level specifications to DNA samples in four stages: *Specification*, *Compilation*, *Part Assignment*, and *Assembly*. Given a behavioral program and a target cellular platform, both specified in a high-level programming language called Proto,[5] TASBE compiles the specification into an optimized design represented as an *abstract genetic regulatory network* (AGRN), then assigns DNA parts to realize the design, and finally produces an optimized sequence of protocols for assembling the design from available DNA samples. The protocols are then executed by a liquid-handling robot or manually, to obtain a physical sample of cells satisfying the specification.

The motivation for the TASBE workflow comes from observing patterns and problems that occur in current work on synthetic biology applications. Many of the envisioned applications of synthetic biology are based on a synthetic genetic construct organized into three modules: sensing of input signals, computation to make decisions based on the inputs, and actuation to turn decisions into actions. For example, Rinaudo et al. constructed an RNA interference-based

Special Issue: Bio-Design Automation

Received: April 15, 2012
Published: July 10, 2012

Boolean function evaluator in mammalian cells for applications in sensing endogenous molecular inputs;[6] Zhen et al. constructed a cellular state classifier for detecting a specific type of cancerous cell using a mixture of threshold tests and Boolean logic, with the miRNA expression profile as input;[3] Anderson et al. built an AND gate to detect a tumor cell;[2] and Tamsir et al. constructed networks implementing all possible two-input Boolean functions with a NOR gate circuit as a basic building block by arranging colonies spatially, which may have applications to pattern formation.[7] Applications also often need computations with memory or time-varying functions to carry out tasks such as counting, detection of events in sequences, and pattern formation. Notable sense−compute−actuate systems using memory or time-varying functions include a transcriptional toggle switch,[8] a five-state recombinase-based sequential memory element,[9] a recombinase-based rewritable memory element,[10] a single-cell transcriptional oscillator,[11] and a multicellularly synchronized genetic clock.[12] There are, of course, many applications that do not currently use the sense−compute−actuate pattern, such as synthesis of drugs,[1] biofuels,[13] and biomaterials.[14,15] The widespread sense−compute−actuate pattern represents an important opportunity, however: the separation of sensing and actuation from computation means that a good approach for producing functioning biological computations from high-level design has the potential to impact a large number of current and potential future applications. Moreover, because many of these prior systems share patterns in their design and construction, it suggests that this class of synthetic biological systems may be particularly susceptible to automation by identification and extraction of these patterns.

At the same time, the growing capabilities of synthetic biology pose an important problem, as the field looks forward to more sophisticated applications that will require larger networks with more complex control logic. Current practices in synthetic biology for specifying, designing, building, debugging, characterizing, and iterating over this process cannot scale as required for such applications. The constructs built thus far have been small, with relatively simple intended behavior and only a small set of available computational parts—hence possible designs—to choose from. The number of available parts is rapidly expanding, however, and as the number of parts and the size and complexity of applications grow, there is an exponential increase in the number of possible designs, and the number of possibilities for problems, such as unknown interactions between parts or mismatched expression levels. These are not distant challenges: even a single actuation that is a Boolean function of three sensors may require 20 to 30 basic functional sequences of DNA and a three to five stage cascade of transcriptional regulation. These difficulties can be addressed by factoring the larger task into tractable subproblems and constructing a modular tool-chain in which each tool addresses a subproblem. This way, existing specification, design, and optimization techniques can be used or adapted, and applications can be shifted easily from one cellular platform to another. While new biological discoveries will alleviate some of the fundamental difficulties in building robust devices, future bioengineers will still have to confront the problems of specifying a desired function succinctly, searching a massive space of potential designs for one that satisfies the specification, and of realizing that design into physical DNA. A well-specified, verifiable, and reproducible workflow will allow the bioengineer to focus on and cope with the key biological complexities of their design. This is consistent with and necessary to the goal of transitioning synthetic biology from a niche experimental art requiring broad and deep training, to a widely accessible engineering discipline.

Our implementation of the TASBE workflow addresses these challenges with a set of software tools whose algorithms are based on the recurring design patterns of sense−compute−actuate synthetic biology systems. Having conceived and implemented the TASBE workflow, we validated it by applying it to the specification of a simple small-molecule sensor/actuator program. Starting with a high level specification of its behavior and using the TASBE flow, we obtained a physical sample of cells satisfying the behavioral specification. TASBE's modular design also allows the same program to be realized on different cellular platforms by swapping workflow configurations, which we demonstrate by executing the flow for two different target cellular platforms: HEK293 mammalian cells and *E. coli* bacterial cells. To our knowledge, the TASBE workflow is the first complete method for translating an abstract program specification to a functioning biological network.

Related Work. A large number of prior projects have addressed particular fragments of this challenge, but no prior effort has actually made the end-to-end connection from specification to *in vivo* functionality.

For example, there are a few high-level programming languages that begin with abstract specifications and map to genetic regulatory network designs at various levels of abstraction: Proto[5,16] uses a dataflow model to produce network designs from a library of motifs, while GEC[17] uses logic programming to search for genetic networks based on chemical reaction models, and the tool by Marchisio et al.[18] uses Karnaugh maps to transform a truth table into a genetic network design for a logic circuit.

Other tools begin at the level of DNA sequence features, allowing the designer to build a design by composing features such as promoters, open reading frames, and terminators. For example, GenoCAD[19] allows a user to specify part sequences constrained by a grammar aimed at ensuring functionality, while Eugene[20] allows textual specification of both parts and combinatorial assemblies of parts. TinkerCell[21] provides similar capabilities in a graphical form, allowing visualization and graphical editing of parts and reaction networks.

Yet others focus on the chemical properties that are needed to ensure correct operation of circuits. A particularly well-known example is the RBS Calculator,[22] a bidirectional translator between RBS sequence and strength. Other tools[23,24] assume such tuning capabilities and focus on searching for the combinations of reaction parameters that can implement particular classes of networks, or use simulation to allow a human to explore the effect of reaction parameters, e.g., SynBioSS.[25] Tools like GLAMM[26] aid a designer in visualization and search for exploring novel transgenic pathways in large databases. Chemical parameters selection can also be considered as an optimization problem, as in the case of OptCircuit,[27] which produces a genetic circuit design given the desired temporal response and a parts library annotated with part parameters, and the tool by Huynh et al.,[28] which assigns parts to a given genetic network by solving a nonlinear optimization problem.

Finally, tools have been created that focus on the low-level problems of sequence design, assembly, and sample production. The j5[29] tool and the Device Editor[30] are tools for specifying combinatorial variations of designs and creating oligos to assemble them; Eugene[20] also provides some capabilities in this area. GeneDesign[31] and GeneDesigner[32] are DNA sequence design tools, whereas OptMAGE[33] is a oligo design tool for genome engineering.

As a collection, these tools address a large number of problems in the design and construction of synthetic biology systems. In general, however, each tool has been formulated to solve only its specific problem, and no effort has previously integrated these tools in order to address the end-to-end design problem; translating a high level behavioral description stated in a programming language into a physical sample implementing the behavior *in vivo*.

■ RESULTS AND DISCUSSION

Two key goals of synthetic biology are engineering of new behaviors in organisms and discovering the principles of such biological engineering.[34−36] The TASBE workflow contributes to the first goal by providing a generalizable solution for translating a desired behavior into a biologically implementable design and assembling DNA implementing that design. Formulating algorithms for this problem also contributes to the second goal, by capturing fundamental engineering principles and practices, their interrelationships, and their strengths and limitations.

The TASBE workflow decomposes the problem of mapping from desired organism to DNA samples into qualitative stages based loosely on the types of work that humans do when they design biological systems: conceptualizing, sketching an initial design, filling in details, and lab work. These stages are expected to apply across a broad range of synthetic biology applications. These qualitative stages are further resolved into a sequence of intermediate models, chosen to be tractable for automation. The models that we present in this paper are focused on sensor/actuator programs that use combinatorial Boolean logic, and with minor extension should be applicable to any sensing or control application, but likely not to complex metabolic engineering. Finally, there is a sequence of transformations that map from each model to the next. Together, these transformations incrementally translate from high-level design specifications to DNA samples. The transformations are expected to be improved frequently in order to expand the classes of behaviors and organisms that can be engineered, to improve algorithms, and to take advantage of new results in the scientific literature. Figure 1 shows the stages and models comprising the TASBE workflow, and the current set of transformations used to implement it.

At the highest level, the TASBE workflow decomposes the problem of designing an organism into four qualitative stages, based on the type of work being carried out:

1. **Specification:** The starting point for the workflow is a precise statement of the organism behavior that is desired and of the target cellular platform in which the behavior is to be implemented.
2. **Compilation:** With the aid of a dictionary that maps between specification elements and genetic motifs, the compilation stage transforms the specification into an abstract genetic regulatory network (AGRN, a "template" implementation that leaves as much flexibility as possible for sequence selection.
3. **Part Assignment:** In the part assignment stage, the AGRN is realized fully as one or more sequences of DNA parts (available assembly-compatible DNA fragments). The choice of DNA parts is guided by a database of available parts and characterization data to determine which parts have interactions that can correctly represent the data values of the AGRN.
4. **Assembly:** Finally, each DNA part sequence is assembled into a physical sample of DNA. The protocols for doing so are determined by the sample library and automation equipment available in the laboratory where assembly is carried out.

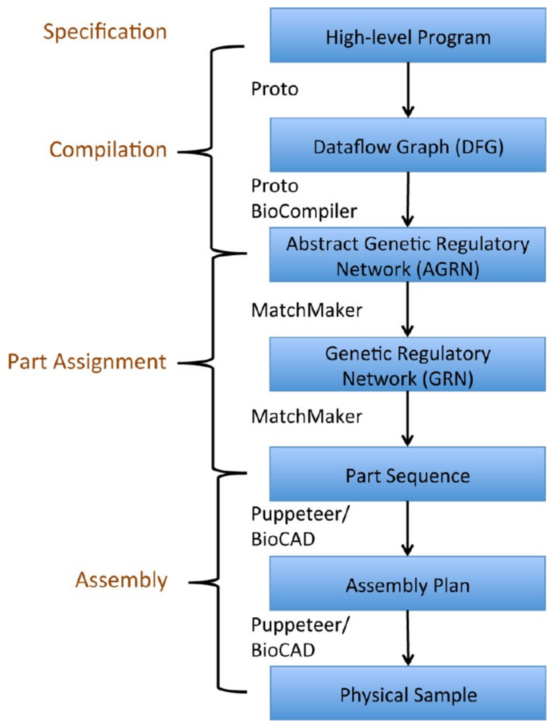


Figure 1. The TASBE workflow has four general stages: specification, compilation, part assignment, and assembly. These are further refined into seven models: (1) high-level program specifying the desired behavior; (2) dataflow graph representing a computation as a set of functional operators that produce and consume data values; (3) abstract genetic regulatory network (AGRN) consisting of a set of partially specified functional units; (4) genetic regulatory network (GRN), consisting of the same elements as an AGRN, except that all of the elements must be fully specified; (5) part sequence(s), where each sequence is a concatenation of available DNA parts with corresponding sample information; (6) assembly plan including a sequence of protocols for creating each part sequence from available DNA samples; (7) physical samples comprising DNA that actually instantiates a biological network. The tools transform each model in turn to its successor.

As shown in Figure 1, these stages are then resolved into a sequence of models that incrementally progress from high-level program specification to DNA samples. We have selected intermediate models as junctures in the design process when new types of information must be introduced in order to proceed. We begin with the program specification, which is obviously essential as it captures the biological engineer's intent. The dataflow graph model is then a digestion of that intent into a discrete set of computational primitives that can implement it. The AGRN represents the farthest progress that can be made without selecting particular DNA parts from an external parts library, while the GRN represents the stage at which decisions must be made about how to organize design

```
(def simple-sensor-actuator ()
  (let ((x (test-sensor)))
    (debug x)
    (debug-2 (not x))))
```

(a) simple-sensor-actuator.proto

```
(annotate not
  :grn-motif ((P high R- arg0 value T)))
```

(b) Sample operator motif

```
(primitive Dox () boolean
 :grn-motif ((P high rtTA|boolean T)
             (RXN (Dox|boolean) activates rtTA)
             (P low R+ rtTA value T)))

(primitive blue (boolean) boolean :side-effect
  :type-constraints ((= value arg0))
  :grn-motif ((P R+ arg0 EBFP2|arg0 value T)))

(macro test-sensor Dox)
(macro debug blue)
```

(c) Excerpt of mammalian-platform.proto

```
(primitive Ara () boolean
 :grn-motif ((P high AraC|boolean T)
             (RXN (Ara|boolean) represses AraC)
             (P high R- AraC value T)))

(primitive green (boolean) boolean :side-effect
  :type-constraints ((= value arg0))
  :grn-motif ((P R+ arg0 GFP|arg0 value T)))

(macro test-sensor Ara)
(macro debug green)
```

(d) Exerpt of ecoli-platform.proto

Figure 2. In the TASBE workflow, Proto code is used to specify programs (a), genetic regulatory network motifs for implementing computations (b), and cellular platforms (c,d).

elements into plasmids and about what protocols will be used to assemble the design. The part sequences(s) are the outcome of those decisions but cannot proceed to the final assembly plan without integrating with laboratory information systems to determine available resources for actually executing protocols. Thus, the TASBE workflow's intermediate models each represent a necessary lowering of the level of abstraction. Although this sequence might not be unique, it is likely that any other end-to-end workflow would need to include equivalent stages.

Finally, we have implemented transformations with a set of tools for mapping between these models. In their present form, these tools can be applied to any sensor/actuator program that uses combinatorial Boolean logic. Specification is done in the Proto programming language.[16] Compilation is implemented using the MIT Proto compiler[37] and the Proto BioCompiler,[5] which produce an AGRN optimized to reduce complexity. Part assignment is handled by a new tool, MatchMaker,[38] which is configured with a platform-specific database using the Clotho data model.[39] Finally, assembly may be carried out using either of two new tools: Puppeteer[40] or BioCAD, both of which create an assembly plan minimizing the length and complexity of protocols and then use a protocol library to translate it to elementary instructions for execution by a human or a liquid-handling robot.

Validation. We validated the TASBE workflow by applying it to a simple sensor/actuator test program with the following behavior:

- Measure the value of a Boolean test sensor.
- If the sensor value is true, turn on a fluorescent debugging reporter.
- If the sensor value is false, turn on a different fluorescent debugging reporter.

We validated the workflow by creating a formal high-level specification of this test program, then applying each tool in sequence, thereby producing DNA samples that correctly implement the specification on two different cellular platforms: mammalian HEK293 cells and *E. coli* bacteria. To our knowledge, this is the first demonstration of end-to-end translation of a program specification to a functioning biological network, as well as the first automated realization of the same design on two different cellular platforms.

Note that the simplicity of the test program we use for validation should not be taken to represent a limitation on the tools or the workflow approach. The contribution of this work is the end-to-end workflow, and most particularly the intermediate models that break the design problem into tractable subproblems. Having established such a workflow, its breadth of coverage can be improved by improving individual tools. For example, Boolean feedback logic requires only the addition of new motifs to the Proto BioCompiler, while analog signal processing would require upgrades to Proto BioCompiler and MatchMaker.

We now describe the results produced by the test program as it progresses through each stage in turn, from specification to ultimate experimental validation of the assembled DNA samples *in vivo*. For details on internals of each transformation, see the Methods section and Supporting Information.

Program and Platform Specification. For our validation test, we began by specifying the sensor/actuator program, a set of network motifs for implementing various program operators, and cellular platform files for HEK293 and *E. coli* cells. High-level program specification, operator motifs, and cellular platform specification are all written in the Proto spatial-computing language.[16] Representative samples of each are shown in Figure 2; the full files are included in the Supporting Information.

Figure 2(a) shows our Proto specification of the simple sensor/actuator test program: the def statement names the program simple-sensor-actuator and declares that it requires no other external inputs (the empty parentheses). The let statement in the next line creates a program variable, in this case, placing the value of the Boolean test-sensor operator into a variable named x. This value is then used to set one of two fluorescent debugging reporters: the first is set when x is true, the second when x is false. Note that the variable x is implicitly of Boolean type, due to the input and output types of the operators producing and consuming it; if these did not match, it would be reported as a compiler error during the next stage.

Figure 2(b) shows the motif declaration for our not operator, taken from Beal et al.[5] This motif specifies a single functional unit comprised of a constitutively high promoter (P high), which can be repressed by the transcription factor representing the operator's input (R− arg0) and which regulates the expression of the protein that represents the operator's output (value).

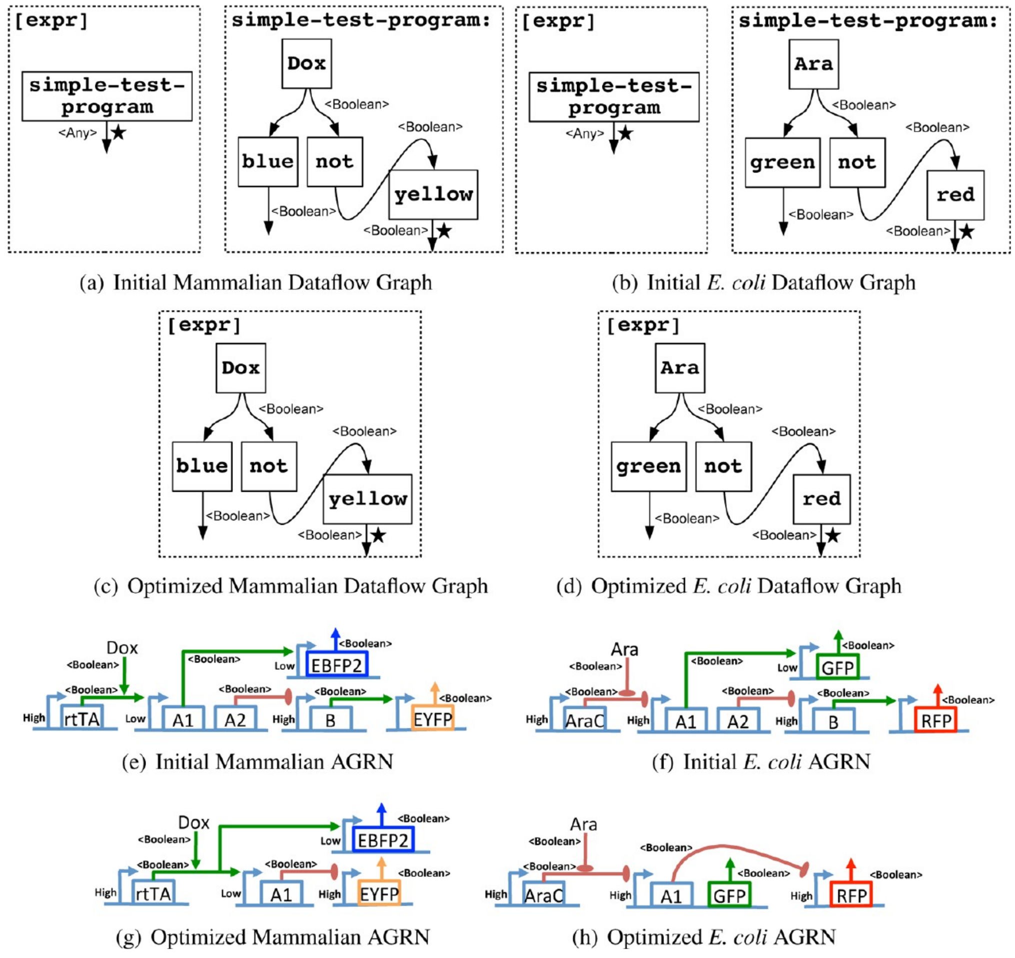


Figure 3. Stages of compilation and optimization for mammalian (a,c,e,g) and *E. coli* (b,d,f,h) cellular platforms. For dataflow graphs (a,b,c,d), operators are shown as boxes, variables as arrows connecting from output to inputs and annotated with their data type, and function definitions as dotted boxes with the name of the function in the corner and a star on the function's output variable. For AGRNs (e,f,g,h), functional units are low horizontal lines connecting promoters (bent arrows) and open reading frames (boxes) with terminators not shown on the diagram; activation and repression are green arrows and red stub connectors respectively, and all are annotated with the data types or values they represent.

Figure 2(c,d) shows sample excerpts from the cellular platform definition files for HEK293 and *E. coli* cells. These declare the available sensors and actuators available on each platform: for example, we had a doxycycline sensor (Dox) for HEK293 but not *E. coli* and an arabinose sensor (Ara) for *E. coli* but not HEK293. Likewise, operators available on both platforms may be implemented differently: for example, macros map the test-sensor and debug actuator families to different implementations reflecting the available operators and the preferences of the biological engineers for each platform.

Compilation. For the next stage of our validation test, we applied Proto and the Proto BioCompiler to the specifications set out in the previous section by evaluating the expression "simple-sensor-actuator", which invokes our sensor/actuator test program. We evaluated this expression once for each cellular platform, selecting between HEK293 and *E. coli* via the arguments provided to the compiler.

Figure 3(a,b) shows the initial dataflow graphs produced for mammalian and *E. coli* platforms, respectively. Note that the different definition files for the two cellular platforms result in a graph with Dox, blue, and yellow for the mammalian platform and with Ara, green, and red in the *E. coli* platform. Since the test program is already quite simple, optimization of the dataflow graph only replaces the function call with its contents (Figure 3(c,d)), a simple transformation but vital since we currently have no way to implement function calls in cells.

Subsequent transformation by the Proto BioCompiler produces first the initial AGRNs shown in Figure 3(e,f). For example, the not operator is mapped to the repression of a functional unit producing an unspecified protein "B", while the Dox and Ara operators map to more complex subnetworks comprising two functional units and a small-molecule reaction. The pattern-based optimizations then simplify these networks. For example, copy propagation and dead code elimination combine to remove protein "B" and the functional unit that produced it. The final complexity-optimized AGRNs are shown in Figure 3(g,h). In more complex networks, the optimizations can reduce network size by up to 83%.[5]

Part Assignment. The part assignment stage of the TASBE workflow fills in the underspecified elements of the AGRN to produce first a fully specified genetic regulatory network, in which each element is associated with an available DNA part, and then a linearization of those parts into one or more complete DNA sequences to be assembled. Our implementation of this stage is a new tool called MatchMaker (presented in detail in Yaman et al.[38]) that solves this problem using a database of available features, parts, and signal levels. MatchMaker comprises three algorithms: the feature assign-

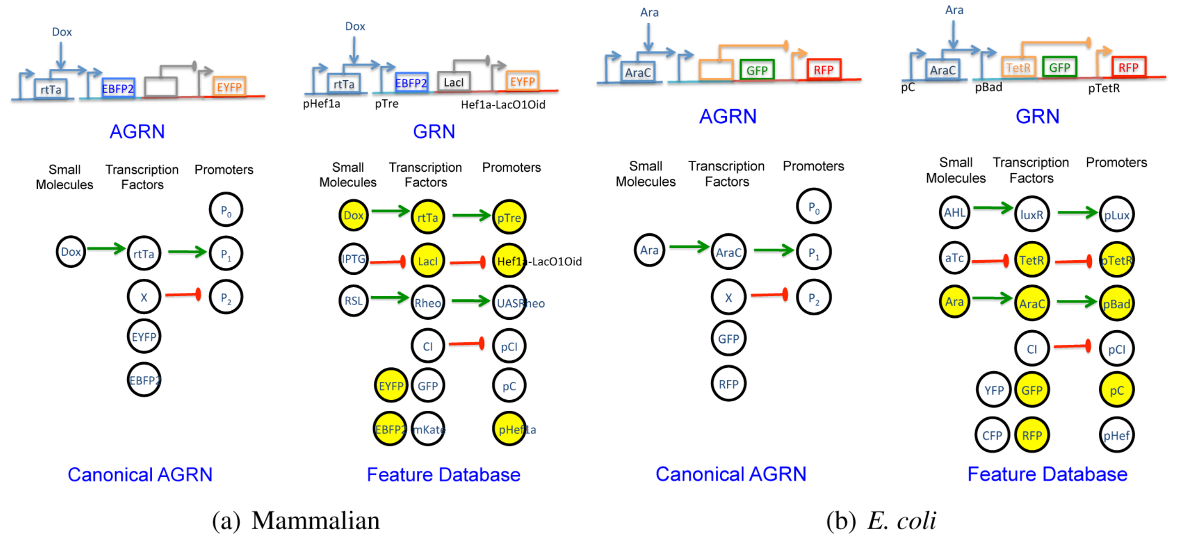


Figure 4. For each cellular platform, MatchMaker uses a feature database and signal level information (not shown) to select parts for instantiating the AGRN of the sensor/actuator test program. Parts selected by MatchMaker are marked in yellow.

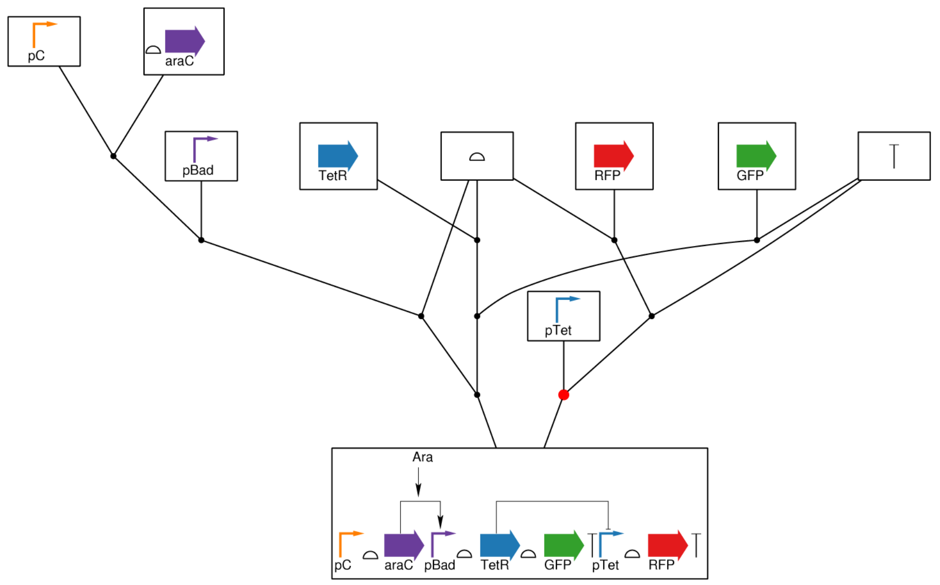


Figure 5. Plan generated by Puppeteer for BioBrick assembly of *E. coli* sensor/actuator test program (root at bottom) from available part samples (leaves at top). Edges represent restriction digests; dots represent ligations. The red dot ligation was executed robotically, while black dot ligations were executed by hand.

ment and signal matching algorithms transform an AGRN into a GRN; the part assignment algorithm then transforms the GRN into part sequences.

We executed MatchMaker on the two AGRNs produced by the compilation stage in our sensor/actuator test example, one targeted for mammalian cells and one for *E. coli*. Each AGRN was run against feature, signal, and part databases created for that cellular platform. Figure 4 illustrates the input AGRNs and the feature databases used for feature matching, as well as the implementing feature set selected by MatchMaker to realize the sensor-reporter program on each cellular platform. The signal database was populated with synthetic threshold data in the form of two pairs of induction-expression points for each promoter-inducer or promoter-repressor part, e.g., the TetR-pTet-RFP-aTc characterization construct was associated with simulated low and high threshold points (0.4,0.5) and (3.0,1.8)—such threshold values can be extracted from experimentally obtained data.[41,42] The final part sequences produced by MatchMaker are shown as the roots of the assembly trees in Figures 5 and 7.

Assembly. The final stage of the TASBE workflow takes one or more part sequences and constructs physical samples comprising that DNA. For this stage, we used two different implementations: Puppeteer[40] and BioCAD.

We applied Puppeteer to assemble the part sequence produced for *E. coli* as an output of MatchMaker. Figure 5 shows the optimized assembly tree planned by Puppeteer, leading to a single root, which is the sensor/actuator test program part sequence output by MatchMaker for *E. coli*.

We first verified by hand that the protocol instructions generated by Puppeteer were a correct implementation of the assembly tree. To validate the robotic instructions, we executed a part of the assembly plan using a Tecan Freedom Evo 150 liquid handling robot. Using Puppeteer, we translated the instructions for assembling the *pTet-rbs-RFP-terminator* partial sequence from the *pTet* and *rbs-RFP-terminator* parts (indicated by the red dot in Figure 5) into robot-specific instructions. This resulted in three sets of instructions corresponding to the two digestion steps and one ligation step (see Supporting Information for details). For each step, we performed two robotic trials and one manual control.

All of the digestion and ligation steps were successful in each of the two trials and were verified by manual gel electrophoresis and by transformation into cells. Figure 6 shows the results of gel electrophoresis of the restriction mapped ligation samples (see Methods section and Supporting Information for details).

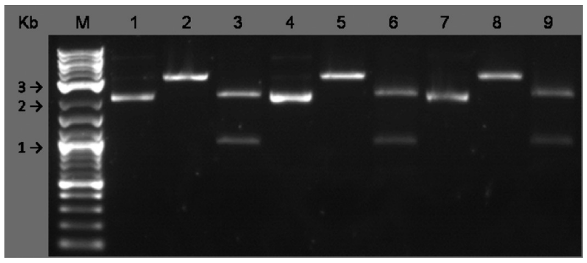


Figure 6. Agarose gel showing successful robotic ligation of *pTet* and the *rbs-RFP-Term* parts by Puppeteer. Lane M shows a 10-kb molecular weight marker. Lanes 1, 4, and 7 show uncut plasmid. Lanes 2, 5, and 8 show single cut plasmid where the plasmid was digested with SpeI and the correct band size is seen at 3016 bp. Lanes 3, 6, and 9 show double cut plasmid where the plasmid was digested with XbaI and SpeI and the correct band sizes are seen at 937 bp and 2079 bp. Lanes 1−3 and 4−6 represent two separate clones screened from the robot ligation, while lanes 7−9 show a clone screened from the manual ligation done simultaneously as a positive control for the automated ligation.

We applied BioCAD to assemble the part sequence produced for HEK293 cells as an output of MatchMaker, after first adding a constitutive reporter to be used as a transfection indicator. Figure 7 shows the assembly tree planned by BioCAD, leading to a single root, which is the sensor/actuator test program part sequence output by MatchMaker for HEK293 cells, plus the additional red fluorescent reporter mKate.

The first level of the assembly tree was executed by hand, creating Gateway entry vectors verified by manual screening and then digested with I-SceI and purified. The second level of the assembly tree was translated into a robotic script and automated using Gibson assembly using a Tecan Freedom Evo 150 liquid handling robot. Figure 8 shows the restriction digest (digested manually, with gel run on the robot; see Methods section and Supporting Information for details). Six samples assembled by the robot produced the same restriction map as six samples assembled manually, with all nonautomated steps being performed in parallel. In addition, the results match previous manual results performed under similar conditions (note that there are some differences between the restriction maps and expected bands as predicted from our sequence data; we are currently sequencing the carrier vector to identify the discrepancies). One robot sample was then chosen for transfection and *in vivo* verification and characterization of the network functionality.

Validation of TASBE Products *in Vivo*. The final test of an engineered biological construct is, of course, its behavior when incorporated into the intended cellular platform. This is the final product of the TASBE workflow—the "model" that is simply the actual engineered cells themselves. Validation of constructs *in vivo* at this stage is thus a validation of the entire end-to-end process used to produce those constructs: if the desired behaviors are observed, then the workflow has been successful.

For end-to-end validation of our sensor/actuator test program, we took the plasmid samples produced by the assembly stage and introduced them to their intended target cells—transfecting the mammalian network into HEK293 cells and transforming the *E. coli* network into *E. coli* bacteria—then cultured the cells in triplicate both with and without induction by the test-sensor chemical for each platform. After the platform-appropriate period of culturing, we measured cellular behavior via microscopy and FACS. For full details, see Methods section and Supporting Information.

Validation of Mammalian Construct. Figure 9 shows the behavior of the sensor/actuator test program in HEK293 mammalian cells. On this platform the Boolean test-sensor is

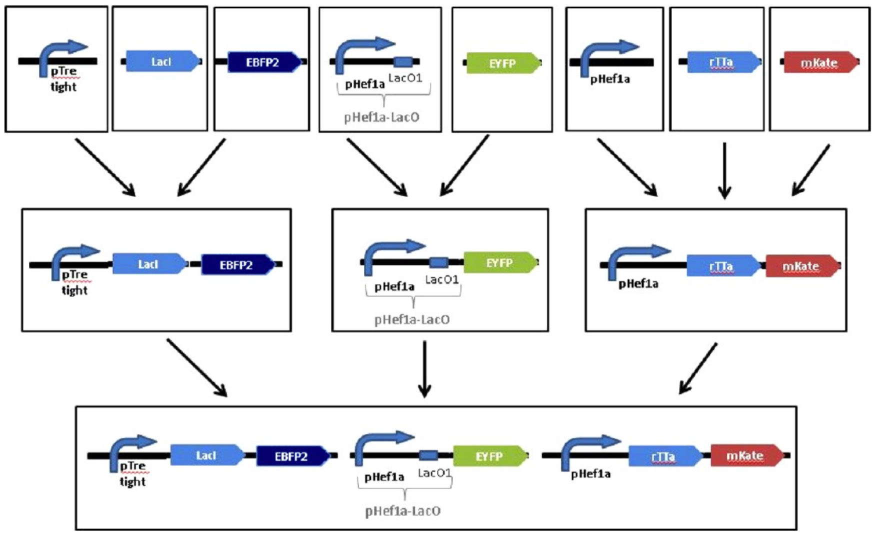
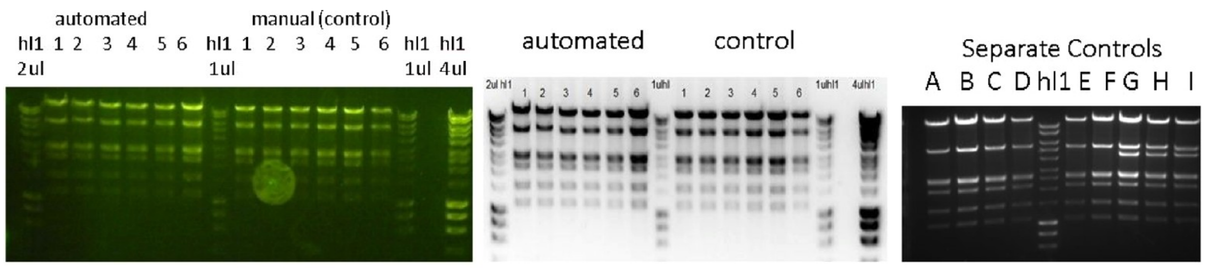
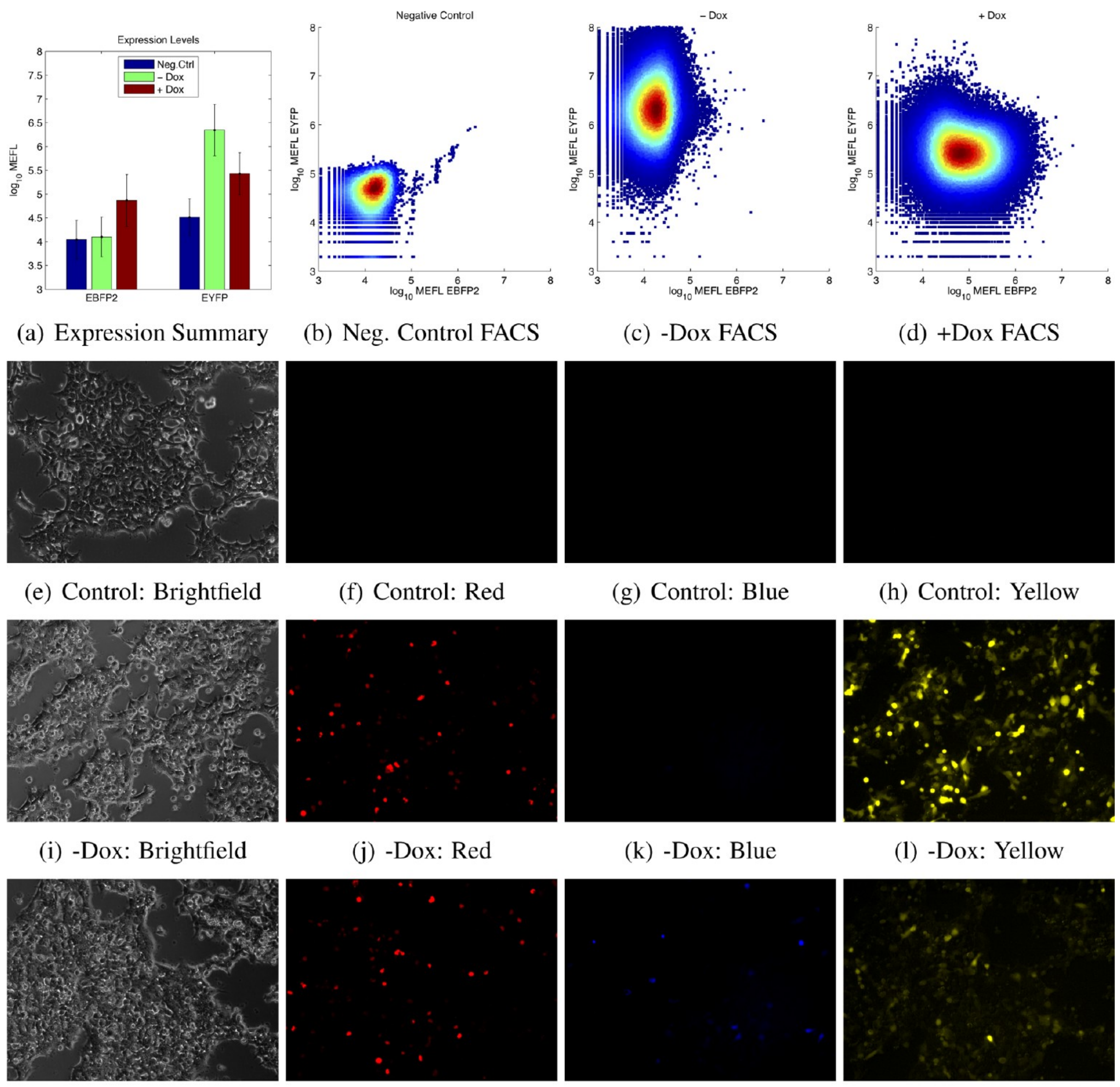


Figure 7. Plan generated by BioCAD for Gibson/Gateway assembly of mammalian sensor/actuator test program (root at bottom) from available part samples (leaves at top).



(a) Automated Gel Image (b) BioRAD Gel Image (c) Additional Controls (BioRAD Gel Image)

Figure 8. Agarose gel showing successful robotic Gibson assembly by BioCAD: (a) six samples from automated Gibson reaction and six samples from a manual control are all in agreement after digestion with enzymes AgeI and NheI (NEB) and compared to Hyperladder I (BioLine). (Note the artifact in manual lane 2 is a plastic post in the prototype clear gel box). (b) Same image acquired with a BioRAD Imager using a blue filter for SybrSafe. (c) DNA fragment sizes match previous unautomated controls in a separate experiment (A–I are manual Gibson assembles imaged on BioRAD imager).



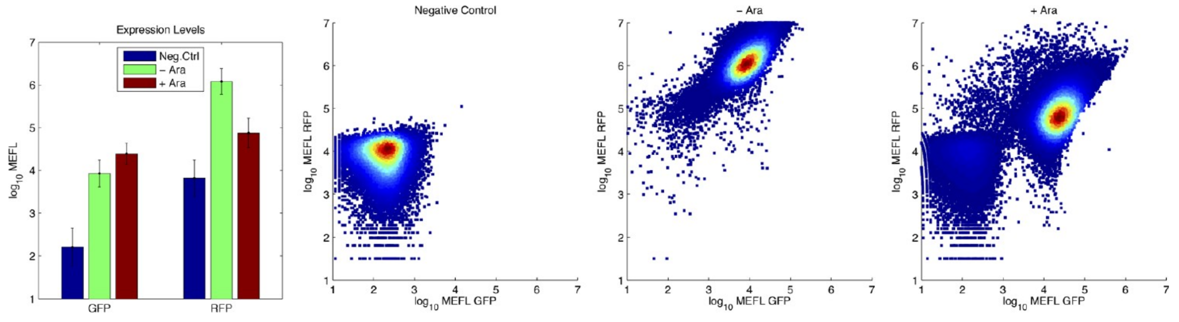
(a) Expression Summary (b) Neg. Control FACS (c) -Dox FACS (d) +Dox FACS

(e) Control: Brightfield (f) Control: Red (g) Control: Blue (h) Control: Yellow

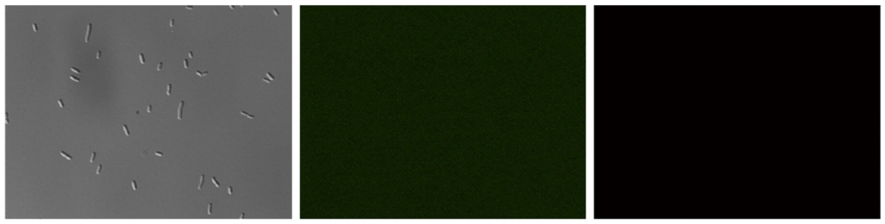
(i) -Dox: Brightfield (j) -Dox: Red (k) -Dox: Blue (l) -Dox: Yellow

(m) +Dox: Brightfield (n) +Dox: Red (o) +Dox: Blue (p) +Dox: Yellow

Figure 9. Validation of mammalian platform sensor/actuator test program plasmid transfected into HEK293 cells: the chart in (a) shows mean and per-cell variation of FACS data for a negative control (b), the true condition (c), and the false condition (d). Relative expression levels conform to the original high-level program specification, and microscopy (e–p) shows agreement with FACS results.

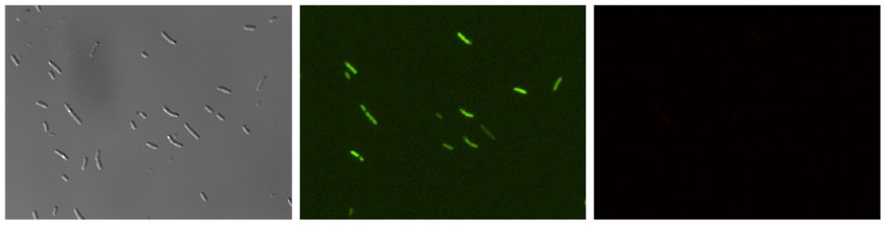


(a) Expression Summary (b) Neg. Control FACS (c) -Ara FACS (d) +Ara FACS



(e) Neg. Control DIC (f) Neg. Control Green (g) Neg. Control Red

(h) -Ara DIC (i) -Ara Green (j) -Ara Red



(k) +Ara DIC (l) +Ara Green (m) +Ara Red

Figure 10. Validation of *E. coli* platform sensor/actuator test program plasmid transformed into *E. coli* cells: the chart in (a) shows mean and per-cell variation of FACS data for a negative control (b), the true condition (c), and the false condition (d). Relative expression levels conform to the original high-level program specification, and microscopy (e−m) shows agreement with FACS results.

mapped to doxycycline concentration; we test the false state with a concentration of 0 nM Dox and the true state with a concentration of 2 μM Dox. The debug test actuator, which is mapped to the EBFP2 fluorescent protein, should be low in the false condition and high in the true condition, whereas the debug-2 actuator, mapped to EYFP, should be high in the false condition and low in the true condition.

Figure 9(a) shows the expression mean and per-cell standard deviation for both actuators in all three conditions. Figure 9(b−d) shows the FACS scatter plots of EBFP2 vs EYFP for negative control cells and for test program cells in the false condition and true condition, respectively. The observed behavior follows the specification: mean EBFP2 expression for the true condition is 5.9 times higher than for false and the low expression is nearly identical to the negative control, while mean EYFP expression for the false condition is 8.3 times higher than for true, though expression is not entirely repressed in the true condition (likely due to the slow dilution of stable EYFP produced before LacI concentration is high enough to repress effectively). Microscopy results are in agreement with FACS: in cells with significant transfection (as indicated by mKate expression on the red channel), induction by Dox markedly raises EBFP2 expression and lowers EYFP expression. Thus, in mammalian cells the test program's behavior *in vivo* complies with its original high-level specification in Proto (though there is much room for quantitative improvement of signal levels and network robustness).

Validation of E. coli Construct. Figure 10 shows the behavior of the sensor/actuator test program in *E. coli* cells. On this platform the Boolean test-sensor is mapped to arabinose concentration; we test the false state with a concentration of 0 mM Ara and the true state with a concentration of 50 mM Ara. The logical specification is the same as in mammalian cells, with different fluorescent reporters. The debug test actuator, which is mapped to the GFP fluorescent protein, should be low in the false condition and high in the true condition, while the debug-2 actuator, mapped to RFP, should be high in the false condition and low in the true condition.

Figure 10(a) shows the expression mean and per-cell standard deviation for both actuators in all three conditions. Figure 10(b–d) shows the FACS scatter plots of GFP vs RFP for negative control cells and for test program cells in the false condition and true condition, respectively. In the true condition a minority subpopulation of cells appear to have discarded or disabled the plasmid; when these are excluded the observed behavior closely follows the specification: mean GFP expression for the true condition is 2.6 times higher than for false (though there appears to be a high degree of leaky expression), and mean RFP expression for the false condition is 13.5times higher than for true. Microscopy shows the same results: induction by arabinose markedly raises GFP expression and lowers RFP expression. Thus, in *E. coli* cells as well the test program's behavior *in vivo* complies with its original high-level specification in Proto (though again there is much room for quantitative improvement of signal levels and network robustness).

■ DISCUSSION AND CONCLUSIONS

We have developed the TASBE workflow for the design and production of physical biological networks from high-level specifications. Our workflow is developed around a sequence of models that bridge from high-level specifications to DNA samples. The choice of appropriate intermediate models decomposes the overall problem into tractable subproblems, allowing the construction of software that realizes the workflow, creating a tool-chain for synthetic biology engineering.

We have validated the TASBE workflow by applying it to design and to construct a correctly functioning sensor/actuator network. As we have shown above, each stage of the tool-chain solves a well-defined subproblem, incrementally moving the design to the final assembled DNA samples, which then behave *in vivo* in accordance with their specification. The TASBE workflow thus provides the first complete method for translating an abstract program specification into a functioning biological network.

Because the TASBE workflow decomposes the overall problem into well-defined subproblems, the tool-chain is modular and therefore extensible. Any algorithm in the tool-chain can thus be modified without affecting other elements of the tool-chain. We have demonstrated this in two different ways: (1) realization of the same program on two different cellular chassis, HEK293 mammalian cells and *E. coli* bacterial cells, with sensors, actuators, and optimizations appropriate to each platform, and (2) swapping the assembly stage to instances customized for different protocols and different laboratory systems. Similarly, the programming language used for specification might easily be swapped for other languages customized to better suit particular problem domains or communities of users.

The modularity of the TASBE workflow also allows for extension with tools that provide new types of capabilities. Tools for verification,[43] debugging, and simulation[21,44] could be connected to the workflow relatively easily. Likewise, the Clotho applications framework allows AGRN and GRN designs to be stored into a Clotho database, where they are accessible to a variety of other applications. The designs can also be exported to SBOL[45] for use with applications outside of Clotho or the TASBE workflow.

Finally, the identification of subproblems exposes a number of new areas for investigation, both regarding the natural behavior of cells and the design and optimization of engineered biological systems. The algorithms deployed at each stage of our workflow are by no means optimal or complete but serve as a foundation for future research. The challenges exposed are not wholly new questions, but rather refinements of general open problems to specific actionable research questions. For example, "characterization of biological parts" is one of the foundational challenges of synthetic biology; see refs 41, 42, and 46–50 for some of the key prior approaches to the problem. One of the challenges for progress in this area, however, has been the lack of a clear definition of what a "part" is or what "characterization" experiments can provide necessary and sufficient data. In the TASBE workflow, the feature and signal matching problems provide a model of part that implies a minimal set of information necessary for the solution of these problems. Moreover, this information is within reach of current experimental techniques.[51] Similar well-defined challenges exist at every stage of the tool-chain: how motifs can be constructed for additional types of computation, sensing, and actuation; how motif equivalencies can best be used to explore the space of optimizations; what throughput yield can be achieved on various protocols with automation assistance; etc. Breaking the research space into such subproblems not only provides a focus for research but guarantees that extensions and improvements in any stage are compatible with research extending or improving other stages as well.

TASBE formalizes the automated design scenario from high-level specification to DNA assembly and addresses the needs of transcriptional networks composed from fully characterized parts.[52] It lays the foundation upon which future extensions can be built. As diverse synthetic biological applications mature,[53] a broader picture of the needs will become available. More complex and less modular design scenarios will require extensions and modifications to the TASBE workflow. The work presented here will serve as a foundation upon which more complex design flows can be built.

It is undeniable that the principles of design of genetic networks are not as well formulated or understood as the principles of software engineering or electronic circuit engineering. However, we hope that these principles will emerge from attempts such as TASBE, formalizing the design process into rigorously defined models and algorithms and that this approach will enable synthetic biologists and engineers to build large and complex genetic networks.

■ METHODS

This section provides details of the TASBE workflow stages: program and platform specification, compilation, part assignment, and assembly. Additional implementation details, as well as specifications of the protocols executed, can be found in the Supporting Information.

Program and Platform Specification. For a high-level program specification language, we selected the Proto[16] spatial computing language. Proto is a purely functional language that represents programs as a functional dataflow computation evolving over continuous space and time. This unusual model of computation makes Proto particularly well suited for the specification of genetic regulatory networks[5,54] and is also anticipated to be useful for specifying differentiated behavior in colonies, tissues, and biofilms.

In the TASBE workflow, Proto is used for three different classes of specification: the program to be implemented, the collection of biological network motifs for implementing elementary program operators, and the cellular platform on which the computation will be realized, including the set of available sensors and actuators.

Specification of Programs. Proto is used to specify the desired behavior that will eventually be realized as a genetic regulatory network. Because Proto uses a functional dataflow model of computation, any operator with a biological network implementation can be composed with any other such operator, and these composites abstracted and composed together further into an arbitrarily complex program. The composite program is guaranteed to still have a biological network implementation (though it may be very complex) so long as there is no recursion.

Specification of Network Motifs. The biological network implementation for each Proto operator is also specified in Proto, using the methods developed in Beal et al.[5] Basic program operators (called "primitives") are either defined or annotated with a:grn-motif annotation. This associates the operator with a motif specifying an abstract genetic regulatory network (AGRN) fragment, with variables that will be filled in when the motif is instantiated and used.

Specification of Cellular Platforms. We also extend the prior work[5] to allow representation of cellular platforms in Proto. This is implemented by modifying the Proto BioCompiler to allow specification of a cellular platform definition file, which is read in and evaluated before the program. The definition file for a cellular platform contains motif definitions of the sensors and actuators available on that platform, as well as macro aliases mapping some of these particular operators to platform-generic operators (e.g., for debugging).

Thus, simply by switching which platform definition file is being used, the design of a program can be customized for different types of organism. Platform files can also be used to represent relations between strains or lab-specific customizations: platform definitions can reference one another using the Proto include command, allowing arbitrary hierarchical families of cellular platforms to be defined, inheriting and overriding definitions from one another.

Compilation. The compilation stage comprises two transformations between models. First, the Proto program is interpreted to produce a dataflow graph model of the computation. Second, the dataflow graph is compiled to an AGRN. During each transformation, pattern-based optimizations are applied to reduce the complexity of the model.

Interpreting Proto into a Dataflow Graph. We use the standard MIT Proto compiler[37] to interpret the Proto program into a dataflow graph. A dataflow graph is a model of an abstract computation in terms of variables and operators. Each variable holds a data value of some type (e.g., Boolean), which can change over time as the computation evolves. Each operator takes a set of variables as input and produces a new variable calculated from the value of its inputs. Some operators, such as sensors, constants, and pattern generators, have no inputs and report either a function of the environmental conditions or a value generated by some internal function. There are also operators, such as actuators and communication, that have side-effects on environmental conditions. The dataflow graph representation has the advantage of being both abstract and universal: any program in any language can be represented as a dataflow graph with appropriate semantics.

Technically, Proto programs actually describe the flow of information over continuous regions of space-time (e.g., the behavior of an entire colony of cells over time), and part of the interpretation process is a global-to-local transformation from this aggregate view to the dataflow graph describing the computation that should happen in single cells at any given instant of time.[16,55] We will not discuss this global-to-local transformation here, as it is trivial for any program that does not include cell-to-cell communication.

For purposes of this paper, then, interpretation of Proto to produce a dataflow graph occurs in two stages. First, each block of code is evaluated into its corresponding dataflow graph structure, and these are connected together on the basis of the relation of the code blocks. Macros act as operators on code blocks; for example, the statement (macro debug blue) causes each debug to be replaced with blue. Next, we infer the types of the variables in the dataflow graph and use standard compiler techniques for pattern-based heuristic optimization to produce produce a more compact equivalent computation.

Compilation of Dataflow Graph to AGRN. The abstract genetic regulatory network (AGRN) model represents a biological design as a collection of DNA functional units, chemical species, and regulatory relations. Each functional unit is a sequence of DNA features—promoters, open reading frames, regulatory regions, and terminators—annotated with data type, regulation, and product information, as appropriate. Each chemical species is annotated with the data type that its concentration represents and the regulatory relations that it participates in, where a regulatory relation is a chemical species that activates or represses some promoter or other chemical species.

Conversion from dataflow graph to AGRN is carried out by the Proto BioCompiler,[5] which uses a method based on design motifs to first produce an AGRN and then to reduce its complexity. Motifs, such as those shown in Figure 2(b–d), are typed patterns that are used to translate each Proto operator in the dataflow graph into an equivalent biological network construct (i.e., fragment of an AGRN). Motifs can specify particular chemical species (e.g., rtTA and Dox in the definition of the mammalian doxycycline sensor) or leave them undefined (e.g., the unidentified repressor in not), to be filled in during the next stage, part mapping. Because they are network fragment specifications, motifs are composable and can easily be extended to represent more types of biological construct. We have enhanced the prior BioCompiler work[5] by allowing type information to be attached to motifs (e.g., the boolean labels in the primitive definitions). This enables better optimization and is required for part mapping. We also now include qualitative reactions (RXN statements) between chemical species (e.g., the activation of rtTA by Dox).

To transform a dataflow graph to an AGRN, the Proto BioCompiler first maps each dataflow operator to its associated motif and each dataflow variable to an unknown regulatory protein with an arbitrary unique identifier. These fragments are then stitched together, replacing the motif input and output variables with the proteins for the corresponding dataflow variables.

The design complexity of this initial AGRN is then reduced by applying a set of pattern-based optimizations, thereby increasing the probability of successful execution *in vivo* by reducing the complexity of the regulatory network as well as the size of the constructs that must be assembled. These pattern-based optimizations are based on standard software compiler optimizations such as dead code elimination, constant elimination, algebraic simplification, and copy propagation. Each transforms the AGRN to an equivalent but simpler network, resulting in a reduced-complexity AGRN. For example, copy propagation tests whether an activator is used only to transfer a data value from one functional unit to another; if so, the original input may be used directly instead. This likely leaves the activator regulating nothing, which allows it to be removed by dead code elimination. The optimizations we use improve on the prior BioCompiler work[5] by the inclusion of additional optimizations and optimizations capable of further reducing the network size based on type information in the motifs.

The BioCompiler tests and applies this set of optimizations, trying to apply each optimization at each location in the AGRN until none of the optimizations can further reduce the AGRN. The reduction in the size of the AGRN varies by program, but it has been observed to reduce network size by up to 83% and the optimized AGRN is often homologous to designs produced by human experts.[5]

The Proto BioCompiler thus translates the behavioral specification of the genetic program into a structural description as an AGRN, which factors the desired behavior into a collection of abstract biological parts that together should produce the desired functionality.

Part Assignment. The part assignment stage of the TASBE workflow fills in the underspecified elements of the AGRN to produce first a fully specified genetic regulatory network, in which each element is associated with an available DNA part, and then a linearization of those parts into one or more complete DNA sequences to be assembled. Our implementation of this stage is a new tool called MatchMaker (presented in detail in Yaman et al.[38]) that solves this problem using a database of available features, parts, and signal levels. MatchMaker comprises three algorithms: the feature assignment and signal matching algorithms transform an AGRN into a GRN; the part assignment algorithm then transforms the GRN into part sequences.

From AGRN to GRN: Feature Assignment and Signal Matching. The AGRN specifies the regulatory relationships required among parts to produce the desired behavior. We need to match these requirements with the behaviors of available genetic parts, whether those parts are designed *de novo* or curated from natural organisms. We call the specific DNA sequence associated with a particular behavior a *feature*. A feature may be related to other features, e.g., the TetR-repressible promoter feature is related to the TetR coding feature due to their natural chemical repression relationship. Given a database of such features and feature relationships and a target AGRN to be instantiated, the feature assignment algorithm maps the elements of the AGRN to a set of features that match the desired regulatory relationships to features and feature relationships. This is done by interpreting both AGRN and feature database as directed graphs and searching the feature database graph for subgraphs that precisely match the structure of the AGRN. Any such subgraph is a fully specified genetic regulatory network that has the qualitative regulatory relationships specified by the AGRN.

Whether a design will function correctly cannot be determined by only using the pairwise relationships between features in isolation. The quantitative nature of these relations (which must be characterized) may not match the context of other relations that they link with, causing the network to fail due to a mismatch of signal levels between relations.

The signal matching algorithm addresses this problem, testing each GRN to ensure a quantitative match between the input and output expression ranges of each pair of regulatory arcs with an input/output relation. The signal matching algorithm relies on input/output characterization of per-cell expression level mean and variance, obtained using a characterization protocol such as the one presented in Beal et al.[51] To signal match Boolean data values, thresholds for interpreting an expression level as “true” or “false” are identified for the inputs and outputs of each regulatory relation. The algorithm then uses these thresholds and their variances to ensure that the output signal of each regulatory element of a GRN is at a level matched to the input threshold of the parts of the GRN driven by it. Furthermore, the algorithm uses the per-cell expression variation to select the GRNs with the maximal expected noise margin, that is, the amount of variation that can be tolerated without misbehavior.

Together, feature matching and signal matching thus map an AGRN to one or more fully specified GRNs, optimized to maximize the amount of variation that can be tolerated and still correctly implement the AGRN.

From GRN to Part Sequence. The functional units of a GRN, which have no particular ordering, now need to be converted into one or more DNA sequences and an associated set of available DNA samples, which when assembled produce the DNA sequence of the GRN. Given a database of available DNA samples, each sample associated with its constituent genetic parts, and a fully specified GRN, the part selection algorithm chooses a sequence of samples from the database that form an *exact cover* of the set of parts assigned to the GRN, that is, the ordering relations of the GRN are preserved, every part is a member of precisely one sample, and no sample contains a part other than the ones in the GRN. MatchMaker’s part selection algorithm finds a single sequence of covering samples using a greedy heuristic that attempts to minimize the number of samples required.

Assembly. Assembly, the final stage of the TASBE workflow, takes one or more part sequences and constructs physical samples comprising that DNA. For this stage, we used two different implementations: Puppeteer[40] is a generalized protocol automation system, whereas the prototype BioCAD tool has been specialized for rapid assembly of large networks, often important for mammalian systems where the size of the DNA features is typically much larger than *E. coli*. For our end-to-end validation, we thus use the more general Puppeteer system to assemble DNA samples for *E. coli* and use BioCAD to assemble DNA samples for HEK293 mammalian cells. Note, however, that there is no tie between implementation and platform; given appropriate configuration data and libraries of samples, Puppeteer and BioCAD are both capable of assembling the same constructs.

Assembly with Puppeteer. In Figure 6, which shows the results of gel electrophoresis of the restriction mapped ligation samples, lanes 1−3 and 4−6 contained samples from the robotic ligation, and lanes 7−9 contained samples from identical manual ligations. In each set, the first lane is the uncut plasmid, and the other two lanes contain a single cut and double cut plasmid, respectively. The identical bands indicate that the robotic ligation step and, therefore, the robotic digestion step were both successful. Compared to the manual control, we did, however, see a larger than expected variation in the reaction volume in all three steps, and in the subsequent colony count after the ligation step. For example, the ligation reaction volume in the manual control was 20 μL, whereas the volume in the two robotic trials was measured to be 14.8 and 16.8 μL. The plates from the manually ligated transformation had more than 300 colonies, whereas the transformations from the two trials of robotic ligations yielded 5 and 37 colonies, respectively. We picked the same number of colonies from both robotic trials and the manual trial, and the restriction map (done manually) verified that they were all correct. We conducted further tests and confirmed that variation in robotics was the cause of the variation in the reaction volume. Since our particular execution of the BioBricks protocol involved the pipetting of liquid volumes as small as 1 μL, it is possible that the robotic pipetting volume variation may have led to the exclusion of a significant amount of a key ingredient of the ligation reaction, e.g., the sample containing the backbone, or the ligase buffer. This would explain the lower colony count. As per robot vendor documents, the robotic pipetting is expected to have a coefficient of variation within 10% when the pipetting parameters for each liquid type being pipetted are tuned to match the physical properties of the liquid. In our validation experiment, we used the default set of parameters as a template for all liquids and only minimally tuned the parameters. We believe this is likely to explain the high variation we observed, which therefore should be correctable by tuning pipetting parameters.

The Puppeteer system[40] is a software system for the specification and automation of biological protocols. Puppeteer accepts as input one or more composite parts, a collection of associated samples, and a supported assembly method. These are input to an assembly planner for the given assembly method, which produces an optimized assembly plan, a sequence of DNA concatenation steps that minimize the total assembly cost by assembling intermediates that may be used repeatedly in the set of parts to be assembled. This plan is converted to a protocol plan by expanding each assembly step into a sequence of protocol steps for the given assembly method. Using a protocol library, each protocol step is then expanded into a sequence of low-level protocol instructions such as aspirate, dispense, shake, and incubate. These instructions can be executed either manually or robotically; where robotic execution is specified, Puppeteer translates these instructions to robot-specific commands and executes them on the robot.

For the work described in this paper, we configured Puppeteer to use the BioBricks assembly method,[56] selecting that method for its frequent use and for the large number of BioBrick parts available for free from the MIT Registry of Standard Biological Parts, as well as for the closure property that implies that any composition of BioBricks is also a BioBrick. However, Puppeteer can also be applied to other assembly methods such as Gibson assembly[57] and Modular Cloning assembly.[58]

Assembling two BioBrick parts into a composite part involves isolating their plasmids from cell stocks, digesting the parts, extracting the digested parts using gel electrophoresis, ligating the digested parts, and transforming them into cells. Each of these steps can be expressed as a protocol in the Puppeteer language and saved in its protocol library. Except for the extraction of a digested part from a gel and the isolation of plasmids from cells, all steps can be executed on a liquid handling robot with a basic grasping arm and a pipetting arm. Complete automation of gel extraction and plasmid isolation steps requires that the robot have access to a robot-compatible gel imaging and cutting station and a centrifuge. Our configuration of Puppeteer executed Puppeteer-generated restriction digestion and ligation protocol steps on the robot and all other protocol steps manually.

For BioBrick assembly, Puppeteer generates optimal plans using the dynamic programming algorithm from Densmore et al.,[59] annotating them with the necessary protocol steps from the BioBricks protocol library. This plan is represented as a multitree data structure, where the leaves are the initial parts, each edge is translated into a restriction digestion protocol with the appropriate front or back insert or vector cuts,[56] and every branch node is translated into a ligation protocol step. This annotated assembly graph is then translated by the Puppeteer compiler[40] into an intermediate-level program with specific pipetting instructions for executing the complete assembly plan (see Supporting Information for details). These instructions can be used to execute the assembly plan manually, or further translated for robotic execution by the same process of protocol expansion.

Assembly with BioCAD. BioCAD is a new prototype tool that has been specialized to a two-level multiway assembly method for rapid assembly of large networks. This is often important for mammalian systems, where the sizes of DNA features are typically much larger than in *E. coli*. The first level uses three-way Gateway Assembly[60] of promoter and gene entry vectors with positional destination vectors. The second level uses multiway Gibson Assembly[57] to compose the Gateway expression vectors into large composite parts. This composite protocol is based on work in Li,[61] in which similar mammalian network assembly of up to 60 Kb has been demonstrated. BioCAD includes an additional optimization that permutes Gibson positions to minimize the number of Gateway reactions required, useful in cases where positional effects on network functionality are negligible. BioCAD also supports the Golden Gate and BioBrick assembly protocols.

As with Puppeteer, BioCAD is configured using lab-specific information, in this case a library file that designates lab-specific well locations for input parts (Gateway entries) and vectors (for both Gateway and Gibson steps). It then uses this information and an interface to laboratory robotics to control laboratory robotics for automation assistance: Gibson steps and gel electrophoresis for validation are carried out robotically, whereas at present the remainder of the protocols are carried out manually.

Gibson assembly is performed by mixing plasmids contain transcriptional units with Gibson master mix and a vector (referred to as the carrier, since the resulting plasmid will be used for mammalian transfection). Challenges in implementing Gibson assembly correctly include pipetting small volumes, keeping the reaction cool (near 0 °C) during setup, and incubation at 50 °C while minimizing evaporative loss.

ASSOCIATED CONTENT

Supporting Information

Additional method details, Proto specification files, script and configuration files for DNA assembly and videos demonstrating the toolchain in action. This material is available free of charge via the Internet at http://pubs.acs.org.

AUTHOR INFORMATION

Corresponding Author

*E-mail: jakebeal@bbn.com.

Notes

The authors declare no competing financial interest.
Authors are listed in alphabetical order except for Principal Investigators Beal, Weiss, and Densmore, who are listed first.

ACKNOWLEDGMENTS

Work partially sponsored by DARPA under Contract No. HR0011-10-C-0168. The views and conclusions contained in this document are those of the authors and not DARPA or the U.S. Government.

REFERENCES

(1) Ro, D.-K., Paradise, E. M., Ouellet, M., Fisher, K. J., Newman, K. L., Ndungu, J. M., Ho, K. A., Eachus, R. A., Ham, T. S., Kirby, J., Chang, M. C. Y., Withers, S. T., Shiba, Y., Sarpong, R., and Keasling, J. D. (2006) Production of the antimalarial drug precursor artemisinic acid in engineered yeast. *Nature 440*, 940−943.

(2) Anderson, J. C., Clarke, E. J., Arkin, A. P., and Voigt, C. A. (2006) Environmentally controlled invasion of cancer cells by engineered bacteria. *J. Mol. Biol. 355*, 619−627.

(3) Xie, Z., Wroblewska, L., Prochazka, L., Weiss, R., and Benenson, Y. (2011) Multi-input RNAi-based logic circuit for identification of specific cancer cells. *Science 333*, 1307−1311.

(4) Dunlop, M. J., Keasling, J. D., and Mukhopadhyay, A. (2010) A model for improving microbial biofuel production using a synthetic feedback loop. *Syst. Synth. Biol. 4*, 95−104.

(5) Beal, J., Lu, T., and Weiss, R. (2011) Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks. *PLoS ONE 6*, e22490.

(6) Rinaudo, K., Bleris, L., Maddamsetti, R., Subramanian, S., Weiss, R., and Benenson, Y. (2007) A universal RNAi-based logic evaluator that operates in mammalian cells. *Nat. Biotechnol. 25*, 795−801.

(7) Tamsir, A., Tabor, J. J., and Voigt, C. A. (2011) Robust multicellular computing using genetically encoded NOR gates and chemical 'wires'. *Nature 469*, 212−215.

(8) Gardner, T. S., Cantor, C. R., and Collins, J. J. (2000) Construction of a genetic toggle switch in *Escherichia coli*. *Nature 403*, 339−342.

(9) Timothy, S. H., Lee, S. K., Keasling, J. D., and Arkin, A. P. (2008) Design and construction of a double inversion recombination switch for heritable sequential genetic memory. *PLoS One 3*, e2815.

(10) Bonnet, J., Subsoontorn, P., and Endy, D. (2012) Rewritable digital data storage in live cells via engineered control of recombination directionality. *Proc. Natl. Acad. Sci.U.S.A. 109*, 8884−8889.

(11) Elowitz, M., and Leibler, S. (2000) A synthetic oscillatory network of transcriptional regulators. *Nature 403*, 335−338.

(12) Danino, T., Mondragon-Palomino, O., Tsimring, L., and Hasty, J. (2010) A synchronized quorum of genetic clocks. *Nature 463*, 326−330.

(13) Keasling, J. D., and Chou, H. (2008) Metabolic engineering delivers next-generation biofuels. *Nat. Biotechnol. 26*, 298−299.

(14) Lewis, R. V. (2006) Spider Silk: Ancient Ideas for New Biomaterials. *Chem. Rev. 106*, 3762−3774.

(15) Widmaier, D. M.; Tullman-Ercek, D.; Mirsky, E. A.; Hill, R.; Govindarajan, S.; Minshull, J.; Voigt, C. A. Engineering the Salmonella type III secretion system to export spider silk monomers. *Mol. Syst. Biol.* 2009, *5*.

(16) Beal, J., and Bachrach, J. (2006) Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intell. Syst.*, 10−19.

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

(18) Marchisio, M. A., and Stelling, J. (2011) Automatic design of digital synthetic gene circuits. *PLoS Comput. Biol. 7*, e1001083.

(19) Czar, M. J., Cai, Y., and Peccoud, J. (2009) Writing DNA with GenoCAD. *Nucleic Acids Res. 37*, W40−W47.

(20) Bilitchenko, L., Liu, A., Cheung, S., Weeding, E., Xia, B., Leguia, M., Anderson, J. C., and Densmore, D. (2011) Eugene: A domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PLoS ONE 6*, e18882.

(21) Chandran, D., Bergmann, F., and Sauro, H. (2009) TinkerCell: modular CAD tool for synthetic biology. *J. Biol. Eng. 3*, 19.

(22) Salis, H. M., Mirsky, E. A., and Voigt, C. A. (2009) Automated design of synthetic ribosome binding sites to control protein expression. *Nat. Biotechnol. 27*, 946−950.

(23) Rodrigo, G., and Jaramillo, A. (2007) Computational design of digital and memory biological devices. *Syst. Synth. Biol. 1*, 183−195.

(24) Rodrigo, G., Carrera, J., and Jaramillo, A. (2011) Computational design of synthetic regulatory networks from a genetic library to characterize the designability of dynamical behaviors. *Nucleic Acids Res. 39*, e138.

(25) Hill, A. D., Tomshine, J. R., Weeding, E. M. B., Sotiropoulos, V., and Kaznessis, Y. N. (2008) SynBioSS: the synthetic biology modeling suite. *Bioinformatics 24*, 2551−2553.

(26) Bates, J. T.; Chivian, D.; Arkin, A. P. GLAMM: genome-linked application for metabolic maps. *Nucleic Acids Res.* 2011,.

(27) Dasika, M. S., and Maranas, C. D. (2008) OptCircuit: An optimization based method for computational design of genetic circuits. *BMC Syst. Biol. 2*, 24.

(28) Huynh, L., Kececioglu, J., Köppe, M., and Tagkopoulos, I. (2012) Automatic design of synthetic gene circuits through mixed integer non-linear programming. *PLoS ONE 7*, e35529.

(29) Hillson, N. J.; Rosengarten, R.; Keasling, J. D. j5 DNA assembly design automation software. *ACS Synth. Biol.* 2012, *1*.

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

(31) Richardson, S., Wheelan, S., Yarrington, R., and Boeke, J. (2006) GeneDesign: rapid, automated design of multikilobase synthetic genes. *Genome Res. 16*, 550−6.

(32) Villalobos, A., Ness, J. E., Gustafsson, C., Minshull, J., and Govindarajan, S. (2006) Gene Designer: A synthetic biology tool for constructing artificial DNA segments. *BMC Bioinf. 7*, 285.

(33) Wang, H. H., and Church, G. M. (2011) Multiplexed genome engineering and genotyping methods applications for synthetic biology and metabolic engineering. *Methods Enzymol. 498*, 409−426.

(34) Andrianantoandro, E., Basu, S., Karig, D. K., and Weiss, R. (2006) Synthetic biology: new engineering rules for an emerging discipline. *Mol. Syst. Biol. 2*, 2006.0028.

(35) Yeh, B. J., and Lim, W. A. (2007) Synthetic biology: lessons from the history of synthetic organic chemistry. *Nat. Chem. Biol. 3*, 521−525.

(36) Shetty, R. P. (2008) Applying engineering principles to the design and construction of transcriptional devices. Ph.D. thesis, MIT, Cambridge, MA.

(37) MIT Proto. software available at http://proto.bbn.com/. Retrieved April 12, 2012.

(38) Yaman, F., Bhatia, S., Adler, A., Densmore, D., and Beal, J. (2012) Automated selection of synthetic biology parts for genetic regulatory networks. *ACS Synth. Biol.*, DOI: 10.1021/sb300032y.

(39) Densmore, D., Van Devender, A., Johnson, M., and Sritanyaratana, N. A platform-based design environment for synthetic biological systems. TAPIA '09: The Fifth Richard Tapia Celebration of Diversity in Computing Conference. New York, NY, USA, 2009; pp 24−29.

(40) Vasilev, V., Liu, C., Haddock, T., Bhatia, S., Adler, A., Yaman, F., Beal, J., Babb, J., Weiss, R., and Densmore, D. (2011) A software stack for specification and robotic execution of protocols for synthetic biological engineering. International Workshop on Bio-Design Automation.

(41) Canton, B., Labno, A., and Endy, D. (2008) Refinement and standardization of synthetic biological parts and devices. *Nat. Biotechnol. 26*, 787−793.
(42) Weiss, R. (2011) Cellular computation and communications using engineered genetic regulatory networks. Ph.D. thesis, MIT, Cambridge, MA.
(43) Yordanov, B., Appleton, E., Ganguly, R., Gol, E., Carr, S., Bhatia, S., Haddock, T., Belta, C., and Densmore, D. Experimentally driven verification of synthetic biological circuits. Design and Test in Europe, Dresden, Germany, 2012.
(44) Myers, C., Barker, N., Jones, K., Kuwahara, H., Madsen, C., and Nguyen, N. (2009) iBioSim: a tool for the analysis and design of genetic circuits. *Bioinformatics 25*, 2848−2849.
(45) Galdzicki, M. et al. Synthetic Biology Open Language (SBOL) Version 1.1.0. RFC 87, 2012; doi: 1721.1/66172.
(46) Kelly, J. R., Rubin, A. J., Davis, J. H., Ajo-Franklin, C. M., Cumbers, J., Czar, M. J., de Mora, K., Glieberman, A. L., Monie, D. D., and Endy, D. (2009) Measuring the activity of BioBrick promoters using an in vivo reference standard. *J. Biol. Eng. 3*, 4.
(47) Group, B. F. A. B., Baker, D., Church, G., Collins, J., Endy, D., Jacobson, J., Keasling, J., Modrich, P., Smolke, C., and Weiss, R. (2006) Engineering life: building a fab for biology. *Sci. Am. 294*, 44−51.
(48) Ellis, T., Wang, X., and Collins, J. (2009) Diversity-based, model-guided construction of synthetic gene networks with predicted functions. *Nat. Biotechnol. 27*, 465−471.
(49) Rosenfeld, N., Young, J., Alon, U., Swain, P., and Elowitz, M. (2007) Accurate prediction of gene feedback circuit behavior from component properties. *Mol. Syst. Biol. 13*, 143.
(50) Hooshangi, S., Thiberge, S., and Weiss, R. (2005) Ultrasensitivity and noise propagation in a synthetic transcriptional cascade. *Proc. Natl. Acad. Sci. U.S.A. 102*, 3581.
(51) Beal, J., Weiss, R., Yaman, F., Davidsohn, N., and Adler, A. (2012) A Method for Fast, High-Precision Characterization of Synthetic Biology Devices; ; Technical Report: MIT-CSAIL-TR-2012-008 http://hdl.handle.net/1721.1/69973.
(52) Slusarczyk, A. L., Lin, A., and Weiss, R. (2012) Foundations for the design and implementation of synthetic genetic circuits. *Nat. Rev. Genet. 13*, 406−420.
(53) Khalil, A. S., and Collins, J. J. (2010) Synthetic Biology: Applications come of age. *Nat. Rev. Genet. 11*, 367−370.
(54) Beal, J.; Bachrach, J. Cells are plausible targets for high-level spatial languages. Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops. Washington, DC, USA, 2008; pp 284−291.
(55) Bachrach, J.; Beal, J.; Fujiwara, T. Continuous space-time semantics allow adaptive program execution. IEEE SASO 2007. New York, 2007; pp 315−319.
(56) KnightT. (2003) *Idempotent Vector Design for Standard Assembly of Biobricks*, pp 1−11, MIT Artificial Intelligence Laboratory, Cambridge, MA, http://web.mit.edu/synbio/release/docs/biobricks.pdf.
(57) Gibson, D. G., Young, L., Chuang, R. Y., Venter, J. C., C. A., H., and Smith, H. O. (2009) Enzymatic assembly of DNA molecules up to several hundred kilobases. *Nat. Methods 6*, 343−345.
(58) Weber, E., Engler, C., Gruetzner, R., Werner, S., and Marillonnet, S. (2011) A modular cloning system for standardized assembly of multigene constructs. *PLoS ONE 6*, e16765.
(59) Densmore, D., Hsiau, T. H. C., Kittleson, J. T., DeLoache, W., Batten, C., and Anderson, J. C. (2010) Algorithms for automated DNA assembly. *Nucleic Acids Res. 38*, 2607−2616.
(60) *GATEWAY^{TM} Cloning Technology Instruction Manual*, Life Techonologies
(61) Li, Y. A (2012) Novel method for mammalian large genetic circuit assembly and delivery, M.Sc. thesis, MIT, Cambridge, MA.

NOTE ADDED AFTER ASAP PUBLICATION

There were errors in the author affiliations in the version published ASAP on July 17, 2012. The corrected version was published on August 8, 2012.