

Functional Synthesis of Genetic Regulatory Networks

Jacob Beal

Raytheon BBN Technologies
jakebeal@bbn.com

Aaron Adler

Raytheon BBN Technologies
aadler@bbn.com

Abstract

As synthetic biologists improve their ability to engineer complex computations in living organisms, there is increasing interest in using programming languages to assist in the design and composition of biological constructs. In this paper, we argue that there is a natural fit between functional programming and genetic regulatory networks, exploring this connection in depth through the example of BioProto, a piggyback DSL on the Proto general-purpose spatial language. In particular, we present the first formalization of BioProto syntax and semantics, and compare these to the formal syntax and semantics of the parent language Proto. Finally, we examine the pragmatics of implementing BioProto and challenges to proving correctness of BioProto programs.

Categories and Subject Descriptors B.6.3 [Logic Design]: Design Aids; D.3.1 [Programming Languages]: Formal Definitions and Theory; J.3 [Biology and Genetics]

General Terms Design, Languages

Keywords synthetic biology, functional programming, semantics, domain-specific languages, spatial computing, Proto

1. Introduction

Synthetic biologists are rapidly developing the ability to engineer and control the behavior of living organisms. As the scale and complexity of the systems that can potentially be designed and implemented is rapidly increasing, the need for effective design tools is rapidly increasing as well [30].

While most design tool work has been focused on lower-level problems of laboratory management and protocol automation (e.g., [18, 19, 31, 35]), there is an increasing interest in programming languages to assist in the design and composition of biological constructs. The range of languages developed to date includes functional [4, 29], declarative [8, 24], and graphical [11, 13] approaches—though to date no significant imperative language has been developed for the design of biological constructs

The lack of an imperative language may be due in part to the design challenge facing all synthetic biology programming languages (Figure 1): a high-level specification must produce a concrete biological realization—typically one or more DNA sequences implementing a genetic regulatory network—which then interacts with

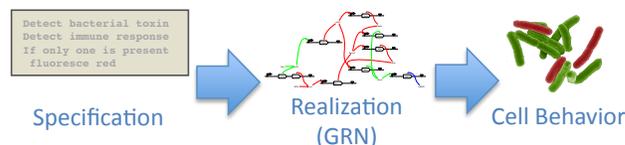


Figure 1. High-level specification of a biological system must produce both a concrete realization (typically as DNA implementing a genetic regulatory network) and the desired behavior in context in cells.

the chemical context and machinery of a living cell to produce a desired behavior.

Most synthetic biology programming languages have avoided the synthesis problem of mapping from biological realization to desired behavior, focusing on specifying the realization, leaving the problem of linking realization and behavior to the designer. To date only one programming language, BioProto [4], has been demonstrated to correctly map high-level behavior specifications into equivalent behaviors realized with actual cells in the laboratory¹. BioProto, a purely functional domain-specific-language first proposed in [2] and realized fully in [4], has recently been demonstrated as the highest-level component of an end-to-end toolchain for automation-assisted production of engineered cells from a high-level behavior specification [6].

Prior publications related to BioProto have been written from the primarily biological perspective, treating the language informally and ignoring the engineering details of its implementation. This paper, in contrast, aims to provide the first formal exploration of BioProto as a domain-specific language, intended to be of use both as a formalization of the language and as a case study of DSL implementation for a domain far outside of the typical range of programming language targets. We begin with an explanatory review of the class of biological constructs considered by BioProto, and provide a formalization of that class. We then develop a syntax and semantics for BioProto, discuss its realization as a plug-in for MIT Proto [26], and discuss the challenges that must be overcome for its correctness to be meaningfully provable.

2. Transcriptional Logic Networks

The target for BioProto compilation is the design of genetic regulatory networks, to be implemented in DNA and executed by insertion of that DNA into living cells. Both natural and engineered biological systems exhibit an extremely wide variety of genetic regulatory networks, encompassing many different mechanisms of by which gene expression is regulated.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FPICSL '13, September 22, 2013, Boston, MA, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2380-2/13/09...\$15.00.

<http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2505351.2505356>

¹The closest similar work of which we are aware is SBROME [21], a graphical programming language that has experimentally verified results, but only minimal support for behavioral specification.

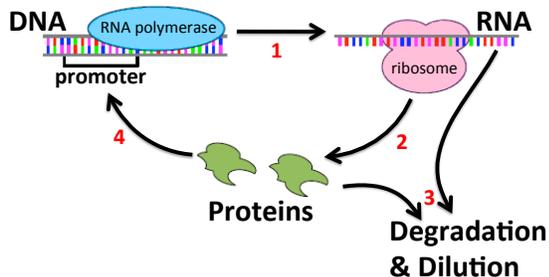


Figure 2. Transcriptional regulation by proteins: 1) transcription copies DNA to RNA, then 2) translation decodes the RNA to produce proteins. 3) Protein concentration is lost to degradation of the protein molecules and to their dilution by cell growth. Proteins carry out or regulate many cellular functions including 4) interacting with promoters to increase or decrease the rate at which particular DNA sequences are transcribed.

In its current incarnation, BioProto focuses on a restricted class of genetic regulatory networks, *transcription logic networks*, which are both widely used and depend on relatively well understood biological mechanisms. Although this covers only a small portion of systems and mechanisms under investigation by synthetic biologists², this limitation is due primarily to the current implementation, and this approach should also be extensible to other biological systems and mechanisms, as we will discuss below.

In the remainder of this section, we first review the core biological mechanism of transcriptional regulation that BioProto uses to realize designs. We then explain how digital logic computations can be implemented with transcriptional regulation, then finally give a formal model of transcriptional logic networks that will be the compilation target of BioProto.

2.1 Central Mechanism: Transcriptional Regulation

The central biological mechanism in this model is the regulation of DNA transcription by proteins, as illustrated in Figure 2. First, the process of *transcription* copies a stretch of DNA that includes a *coding sequence* for a protein into complementary RNA. The process of *translation* then decodes the sequence of nucleotides in the RNA to produce proteins. Proteins, in turn, are the molecular machinery used by the cell to carry out most of its functions. This includes regulating protein expression: proteins known as *transcription factors* bind to *operator* sites associated with the *promoter* region “upstream” of a coding sequence, thereby either increasing (“activating”) or decreasing (“repressing”) the rate of transcription for the coding sequence “downstream” of the promoter. In general, the higher the concentration of the transcription factor is, the stronger the regulatory effect.

The dynamics of this process depend on the balance between production, degradation, and dilution. The concentration of RNA and proteins in a cell increases over time at a rate determined by the rates of transcription and translation. At the same time, the concentration is generally decreased by two processes: the degradation of RNA and proteins back into their primitive building blocks and their dilution as the cell grows and divides.

Transcription and translation are carried out by natively occurring molecules in a cell (RNA polymerase and ribosomes, respectively), so an engineered network can in many cases be “powered”

simply by inserting a DNA sequence or sequences encoding it into a cell.

Because DNA sequences can be assembled together arbitrarily, it is possible to place any given set of coding sequences “downstream” from any promoter/operator region. In principle, this means that any transcription factor can be used to regulate any coding sequence (though there are exceptions to this rule). A collection of interacting promoters and coding sequences thus form a regulatory network. A *transcriptional logic network* is such a network where the high and low concentration are being interpreted as Boolean true and false values.

2.2 Computing with Transcriptional Logic Networks

Building on this basic model, synthetic biologists have identified and experimentally verified the designs for a wide variety of digital logic gates (see, for example, the seminal work in [34], [33], [16], and [17]).

For example, a NOT gate can be implemented with a transcription factor that represses a promoter. The transcription factor is the input to the NOT gate, and the coding sequences regulated by the promoter is its output. When the concentration of transcription factor is high, transcription at the promoter is repressed and the concentrations of its products will be low. When the concentration of transcription factor is low, transcription at the promoter is active and the concentrations of its products will be high. For purposes of this paper and BioProto, we will consider the logical signal to be encoded in terms of the chemical concentration, though alternate proposals have been made, such as the rates of transcription (PoPS) or translation (RiPS) [12].

Unlike voltages, chemical concentrations produced by different sources superpose. Thus, a NOT gate can be transformed into a multi-input NOR gate by having each input produce the same repressing transcription factor. This also means that each chemical can generally only be used once in a given genetic regulatory network: multiple uses will interfere with one another unless they are isolated in different compartments, which is not practical in most circumstances with current technology.

The output of one logic gate is “wired” to the input of second logic gate by placing the coding sequence for the second under control of the promoter from the first. Computation then evolves at all gates simultaneously in parallel, just as in a combinational electronic circuit. The time scales are quite different, however: transitions between high and low expression levels typically occur on a scale of hours to days. The speed is typically limited by the high-to-low transition, which depends on the dilution and degradation rates, which are organism and construct specific. For example, a fast degrading protein in *E. coli* may effectively switch states in a single hour, while a stable protein in a slow-dividing mammalian cell may take a week or more. While this means pure computation in cells will never rival silicon, the value of cell computation comes from using it for sensing and control of chemical and structural properties on the nanoscale.

Sensors and actuators in this model are also implemented with proteins. Sensor proteins are typically transcription factors that undergo a chemical reaction changing their regulatory activity when exposed to the stimuli that they react to, such as light, pressure, or small signalling molecules. Actuators are generally not transcription factors but proteins that implement some other form of physical effect, such as fluorescing, controlling the cell’s metabolic activity, or generating a voltage across the cell membrane.

2.3 Formal Model

We can formalize the class of typical transcriptional logic networks with the syntactic model given in Figure 3. This model is a textualization of a subset of standard diagrammatic models used

²Omitted classes include analog computation, dynamic modification of DNA sequences, cell-to-cell communication, biological nanostructures, and metabolic engineering.

TranscriptionalLogicNetwork	=	{(" , FunctionalUnit Reaction, ")
FunctionalUnit	=	{Operator}, Promoter, {Operator}, {CodingSequence}, ["terminator"]
Promoter	=	"High_Basal_Activity_Promoter" "Low_Basal_Activity_Promoter"
Operator	=	ChemicalSpecies, LogicalRegulation
CodingSequence	=	"produces", ChemicalSpecies
Reaction	=	ChemicalSpecies, LogicalRegulation, ChemicalSpecies
LogicalRegulation	=	"activates" "represses"
ChemicalSpecies	=	"[" , SpeciesName, ", ", LogicalType, "]"
SpeciesName	=	sequence of any visible ASCII characters except .[,]
LogicalType	=	"boolean"

Figure 3. Syntax in EBNF for specification of a transcriptional logic networks as a set of DNA functional units, each comprising a regulatory region followed by a set of coding sequences and a transcriptional terminator.

by biologists to describe and design genetic regulatory networks. In essence, this model specifies directed graphs with two classes of nodes, `FunctionalUnit` and `ChemicalSpecies`, and three classes of edges: `activates`, `represses`, and `produces`. The `activates` and `represses` relations go from a `ChemicalSpecies` to either type of node, while `produces` relations go only from a `FunctionalUnit` to a `ChemicalSpecies`.

Each `ChemicalSpecies` is a pair associating a name and a type (though for logical genetic regulatory networks the type can only be a `Boolean`).

The `FunctionalUnit` nodes, on the other hand, have a stereotyped substructure going from “upstream” to “downstream” in the direction of transcription on a DNA sequence. First comes a `Promoter` with either a high or low base rate of transcription, which may be flanked on either side by `Operator` sites that regulate it. Downstream of this comes a sequence of `CodingSequence` sites for the proteins produced by this functional unit³, and finally a terminator sequence (needed in some organisms but not others) that marks the end of the region that is transcribed to RNA.

Note that there is no guarantee that all members of the class specified by this model are pragmatically realizable, merely that all practically realizable transcriptional logic networks belong to this class. Likewise, note that this model does not specify how or whether multiple functional units are to be linearized onto DNA strands, only the linearization of the components within a functional unit.

One issue with this formalization is that the relation between multiple operators is not clear: if a promoter is activated by some species but repressed by others, which should dominate? The reason this is unclear in the representation is because there is no scientific consensus in this question. For our current purposes, we will consider the behavior of a promoter that is both activated and repressed to be well-defined only if it has only two operators, such that the basal activity of the promoter can be used to tie-break between them (i.e., with high basal activity the activator overrides the repressor, while with low basal activity it is the other way around).

2.4 Transcriptional Logic Network Example

Let us illustrate the formal model with an example of a simple network, which fluoresces by expressing Green Fluorescent Protein (GFP) whenever there is a low concentration of anhydrotetracycline (aTc), an analogue of the antibiotic tetracycline:

```
(High_Basal_Activity_Promoter
 produces TetR|boolean terminator)
(aTc|boolean represses TetR)
(High_Basal_Activity_Promoter TetR represses
 produces LacI|boolean terminator)
```

³ Any additional necessary machinery around the coding sequence, such as ribosome binding sites or 2A sequences for polycistronic expression, is implicit.

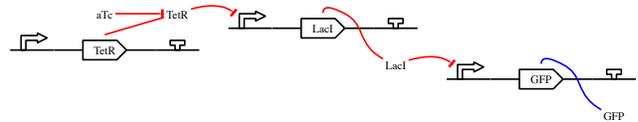


Figure 4. Diagram of a simple transcriptional logic network, which fluoresces green by expressing GFP whenever the small molecule aTc is not present. Red bars indicate repression, and the blue line indicates production of a protein with side effects—in this case, fluorescence.

```
(High_Basal_Activity_Promoter LacI represses
 produces GFP|boolean terminator)
```

The first statement specifies unregulated expression of a transcription factor called TetR that reacts to the presence of aTc. When aTc binds to TetR, it represses the activity of the transcription factor, effectively nullifying it, as expressed by the second statement. TetR, when not nullified, is itself a repressor, so aTc repressing a repressor is a double negation, corresponds to a positive sensor for aTc. The third statement links a promoter responding to this TetR to production of another transcription factor named LacI, so that LacI is produced when aTc is present at a high concentration and repressing TetR, but is not produced when aTc is not. Note that for cases like this, where the operator needs to follow the promoter on the DNA, the pseudo-English is unfortunately “backwards” as “P... TetR represses” rather than “TetR represses P...” LacI, in turn, represses the production of GFP. Thus, when aTc is present at a high concentration, LacI is produced and represses GFP. When the concentration of aTc is low, TetR represses LacI and GFP is free to be expressed. Finally, note that every chemical species is marked at least once with the type `boolean`, indicating that we interpreting these concentrations as Boolean values and expect them to be markedly high or low, and not generally at intermediate values.

For reasons of space, we cannot include a larger example in this text; for several such examples, including a large circuit implementing a two-bit adder, the reader is referred to [4].

3. BioProto Syntax

The functional dataflow model of computation is a good fit for specifying the design of a transcriptional logic network. If the flows in a dataflow graph are taken as analogous to the flows of regulation, then a specification of a dataflow computation treats both the realization and behavior levels of Figure 1: the computation is the behavior and the dataflow is the genetic regulatory network realization. Dataflow is also inherently parallel, which matches well with the simultaneous evolution of all elements in a genetic regulatory network. Moreover, functional programming languages allow a simple specification of dataflow graphs, since functional composition translates transparently to dataflow relations.

Our basic approach, as described in [4] is to associate fragments of transcriptional logic network with in a higher level primitive operators. A dataflow graph specification in a higher-level language can then be transformed into a transcriptional logic network by mapping each edge in the dataflow graph to a unique transcription factor and each operator in the dataflow graph to its associated fragment of transcriptional logic network.

Proto [3] is a functional dataflow language that aligns particularly well with the design of transcriptional logic networks, since the dataflow graphs it specifies are defined to operate constantly, evolving their values over continuous time, just as a transcriptional logic network must. Proto also has the advantage of being intended for succinct specification of programs for aggregates composed of large numbers of locally-interacting individual computational devices—it is, in fact, one of only a few general purpose spatial languages [7]. Many populations of cells, such as colonies, biofilms, or tissues in multicellular organisms, can be viewed as such locally-interacting aggregates, meaning that Proto has the potential to eventually allow simple specification of designs for complex biological aggregates.

The efficacy of the match between Proto and genetic regulatory networks has been explored extensively in [2] and demonstrated in [4], including the automated design and simulation of large and complicated genetic regulatory networks. It has also been demonstrated with the realization of small BioProto programs executed by actual living cells [6]. Note, however, that the present implementation of BioProto supports cell-to-cell communication only implicitly, such that it is not yet possible to take full advantage of Proto’s aggregate programming capabilities.

By the classification of domain-specific languages in [25], BioProto is a piggyback language on Proto. This means that its design and implementation is simplified by building off of the existing language, adding some features and restricting others. In particular, BioProto modifies Proto as follows:

- Primitives can be annotated with genetic regulatory network specifications, and
- Only a restricted subset of Proto is used, excluding all non-Boolean types, all operations that require non-Boolean types, and all operations over space or time.

3.1 Genetic Regulatory Network Specifications

Figure 5 shows the syntax for specifying genetic regulatory networks in BioProto (neglecting the use of whitespace to aid in tokenization). This syntax is intentionally nearly identical to the formal model of transcriptional logic networks presented in 3. The differences between the two are:

- A number of small permutations of surface syntax are made with the aim of making code more compact (e.g. T instead of `terminator`) or more explicit (e.g., starting reactions with `RXN`).
- BioProto inherits from Proto a tighter restriction on the characters that can make up names of chemical species.
- The type of a chemical species can be set indirectly, as being equal to the type of another species. Although all are Boolean, the equality relation also allows inference about constant values by network optimizers.

Note that although at present this only supports specification of transcriptional logic networks, it is designed for expansion to a wider range of genetic regulatory networks by adding more types, more classes of promoter and operator, etc.

One other key difference is not apparent at the syntactic level: the identifier for a chemical species may be either a literal that names a particular class of molecule, or a placeholder variable to

be filled in later. Which case this is depends on evaluation context, however, and will be discussed further when we consider program semantics in Section 4.

3.2 Functional Specification of Behavior

The restricted subset of Proto used by BioProto is shown in Figure 6. This subset has the same syntax as presented in [32], but without any of Proto’s distinctive space-time constructs. It might be argued that such a language is no longer meaningfully Proto, but for the roadmap laid out in [2] for reintegrating those constructs later, after more progress has been made on some of the challenges that we will discuss in Section 6.

The without Proto’s space-time constructs, the restricted syntax of BioProto is a simple LISP-like language based on S-expressions. A program consists of a sequence of declarations, followed by precisely one expression, which determines what the program will compute.

Primitive operators are declared with the `primitive` construct, associating an operator name and typed function signature with a map from annotation type to S-expression. Annotations recognized and used by BioProto are:

- `:grn-motif`, which associates a primitive with a genetic regulatory network fragment, specified using the genetic regulatory network syntax in Figure 5
- `:side-effect`, a standard Proto annotation marking a primitive as having side-effects that should not be optimized away.
- `:type-constraint`, another standard Proto annotation that declares identities between the types of variables, for use in compilation and optimization.

The `annotate` construct simply adds annotations to an existing primitive, possibly replacing some of those already declared.

Expressions follow the LISP conventions of S-expressions: a tree of function applications indicated by parentheses, leaves consisting of variables or constants, and variables declared in `let` constructs. Finally, `def` constructs can declare functions or global variables, abstracting and wrapping expressions respectively.

Taken together with the genetic regulatory network syntax in Figure 5, these form the totality of syntax for the BioProto domain-specific language.

3.3 BioProto Example

The transcriptional logic network example in Section 2.4 could be specified in BioProto as follows:

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

(primitive not (boolean) boolean
 :grn-motif ((P high R- arg0 value T)))

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

(green (not (aTc)))
```

The network is now specified in terms of three basic capabilities, sensing `aTc`, inverting a logical value, and expressing green fluorescence. These are then connected together to form a composite behavior specification, `(green (not (aTc)))`, which might be read as “fluoresce green when not sensing `aTc`.” We will examine how such a specification is interpreted to produce a genetic regulatory network in the next section.

GeneticRegulatoryNetwork	=	{(" , FunctionalUnit Reaction, ")};
FunctionalUnit	=	{Operator}, Promoter, {Operator}, {CodingSequence}, [Terminator];
Promoter	=	"P", ["high" "low"];
Operator	=	("R+" "R-"), ChemicalSpecies;
CodingSequence	=	ChemicalSpecies;
Terminator	=	"T";
Reaction	=	"RXN", ChemicalSpecies, ("activates" "represses"), ChemicalSpecies;
ChemicalSpecies	=	ProtoIdentifier, [" ", (ProtoIdentifier ProtoType)]
		"(", ChemicalSpecies, ")";
ProtoType	=	"boolean";
ProtoIdentifier	=	<i>any sequence of alphanumeric and * + \ - . / < = > ? _ & : characters, except those sequences that parse as a number</i>

Figure 5. Syntax in EBNF for specification of genetic regulatory network fragments in BioProto, closely derived from the transcriptional regulatory network syntactic model in Figure 3

BioProtoProgram	=	{ Primitive Annotate Variable Function }, Expression;
Primitive	=	"(", "primitive", ProtoIdentifier, Signature, {Annotation}, ")";
Signature	=	"(", {Argument}, ")";
Argument	=	ProtoType ProtoIdentifier, [" ", ProtoType];
Annotate	=	"(", "annotate", ProtoIdentifier, {Annotation}, ")";
Annotation	=	:grn-motif, "(" , GeneticRegulatoryNetwork, ")"
		:side-effect
		:type-constraints, "(" , { "(" , "=", ProtoIdentifier, ProtoIdentifier, ")" }, ")"
Expression	=	"(", Expression, ")"
		ProtoIdentifier
		Constant;
		"(", "let" "(" { "(" , ProtoIdentifier, Expression, ")" } ")" { Expression } ")";
Constant	=	"true" "false";
Variable	=	"(", "def", ProtoIdentifier, Expression, ")";
Function	=	"(", "def", ProtoIdentifier, "(" , {Argument}, ")" , {Expression}, ")";

Figure 6. Syntax in EBNF for remainder of BioProto beyond specification of genetic regulatory network fragments; this language is a restriction of general Proto syntax.

One noteworthy aspect of this specification is that the `aTc` operator includes multiple interacting elements in its `:grn-motif` annotation, illustrating that these annotations are complete network fragments, not just single functional units. Also of note is the fact that `green` produces a value as well as expressing green fluorescent protein (GFP). This is because Proto requires all functions to produce some value; `green` is thus set up as a “pass-through” actuator, which returns a copy of the value on which it was invoked. This is reflected in the `:type-constraint` annotation, which, along with the `| arg0` equality typing on GFP, sets up type identity relations that can be used for optimization.

Working in the opposite direction, the `:side-effect` annotation ensures that BioProto knows that this network fragment is important for functionality and not just computation, and thus its protein literal GFP must not be optimized away.

This example also illustrates why optimization is needed and a potential down-side to higher-level specifications: the specification of `green` requires an activation regulation, `R+ arg0`, that is not in the network in 2.4, in order to copy the concentration of an input protein to the concentration of GFP, since it cannot *a priori* know the context in which the primitive will be used. Thus, BioProto requires optimization in order to be able to remedies inefficiencies such as this one that are introduced by using a higher level of representational abstraction.

4. Interpretation Semantics

In specifying the semantics of BioProto, we will concentrate on the unique aspects of the language. To that end, we make the following assumptions:

- The BioProto program has already parsed correctly, such that we can safely assume syntax compliance.
- All primitives and annotations declarations are have been parsed into a map \mathcal{P} of tuples (p, \bar{v}, e) that associate each primitive p identifier with its signature \bar{v} and `:grn-motif` expression e . Each element in the signature is a name, defaulting to `arg k` for the k th argument and `value` for the return if they are specified with only types rather than names.
- Function calls and variables have already been inlined.

To map from BioProto to transcriptional logic networks under these assumptions, our semantics thus only needs to treat interpretation of `Expression` and `:grn-motif` expressions.

The semantics we develop uses a small-step transition model based on [22]. Rules are shown with the transition \rightarrow that they implement on the bottom and preconditions on the top. Successful evaluation is indicated by production of a single expression matching the `TranscriptionalLogicNetwork` syntax from Figure 3. These rules also have a progress property (not proven here for reasons of space), such that for any expression for which such a production is possible, every sequence of rule matches advances monotonically toward the same final network expression.

We will designate variables in italics (e.g., v), sequences with an overbar (e.g., \bar{v}). Interpretation state is represented using a parsing placeholder \square and triple $\langle \mathcal{M} : \mathcal{E} : T \rangle$. In this state, $\mathcal{M} \in \{\mathcal{C}, \mathcal{G}, \mathcal{P}\}$ is a mode marker used to swap between three modes: interpretation of chemicals, genetic regulatory networks, and Proto expressions. The \mathcal{E} state is an evaluation environment consisting of a mixed sequence of variable bindings $x \Rightarrow y$ and typed

chemicals $[c, t]$, where t is either `boolean` or not yet typed, which we designate as \emptyset . We annotate extraction of a value from the environment as $v \otimes \mathcal{E}$, which matches environments containing v as well as pulling it to the front of the sequence. Finally, state \mathcal{N} is the current transcriptional logic network.

Interpretation of a BioProto program begins with the state $\langle \mathcal{P} : \emptyset : \emptyset \rangle e$: no environment bindings, no network contents, and the entire program expression e .

$\frac{\mathcal{L}(\mathcal{E}, e) \rightarrow^* \mathcal{E}' : [c, t]}{\langle \mathcal{C} : \mathcal{E} : \mathcal{N} \rangle e \rightarrow \langle \mathcal{C} : \mathcal{E}' : \mathcal{N} \rangle [c, t]}$	[Chemical]
$\frac{t \in \text{ProtoType} \quad t' \in \{t, \emptyset\}}{\langle \mathcal{C} : [c, t'], \mathcal{E} : \mathcal{N} \rangle t \rightarrow \langle \mathcal{C} : [c, t], \mathcal{E} : \mathcal{N} \rangle [c, t]}$	[TypeChemical]

Figure 7. Rules for interpretation of chemicals

In both sublanguages, chemical parsing and typing is handled by the rules shown in Figure 7. Rule [CHEMICAL] uses a lookup subfunction:

$$\mathcal{L}(\mathcal{E}, c,) = \begin{cases} \mathcal{L}(\mathcal{E}, c') & \text{if } c \Rightarrow c' \in \mathcal{E} \\ [c, t] \otimes \mathcal{E} : [c, t] & \text{if } [c, t] \in \mathcal{E} \\ [c, \emptyset], \mathcal{E} : [c, \emptyset] & \text{otherwise} \end{cases} \quad (1)$$

to follow variable bindings and extract a chemical from the evaluation environment if it exists there, and to add it if it does not. Rule [TypeChemical] can only be invoked immediately after a chemical is looked up, per the syntactic rules, and asserts the type on the chemical.

$\frac{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \bar{e} \rightarrow^* \langle \mathcal{G} : \mathcal{E}' : \mathcal{N} \rangle \bar{i}'}{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle (\bar{i} \parallel \bar{e} e') \rightarrow \langle \mathcal{G} : \mathcal{E}' : \mathcal{N} \rangle (\bar{i} \bar{i}' \parallel e')}$	[Traverse]
$\frac{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle (\bar{e} \parallel) \rightarrow^* \langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle (\bar{e} \parallel)}{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle (\bar{e} \parallel) \rightarrow \langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle (\bar{e} \parallel)}$	[Recurse]
$\frac{-}{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{P} \rightarrow \langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{Low_Basal_Activity_Promoter}}$	[P:Default]
$\frac{-}{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{P low} \rightarrow \langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{Low_Basal_Activity_Promoter}}$	[P:Low]
$\frac{-}{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{P high} \rightarrow \langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{High_Basal_Activity_Promoter}}$	[P:High]
$\frac{\langle \mathcal{C} : \mathcal{E} : \mathcal{N} \rangle \bar{e} \rightarrow^* \langle \mathcal{C} : \mathcal{E}' : \mathcal{N} \rangle i}{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{R+} \bar{e} \rightarrow \langle \mathcal{G} : \mathcal{E}' : \mathcal{N} \rangle i \text{ activates}}$	[Activate]
$\frac{\langle \mathcal{C} : \mathcal{E} : \mathcal{N} \rangle \bar{e} \rightarrow^* \langle \mathcal{C} : \mathcal{E}' : \mathcal{N} \rangle i}{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{R-} \bar{e} \rightarrow \langle \mathcal{G} : \mathcal{E}' : \mathcal{N} \rangle i \text{ represses}}$	[Repress]
$\frac{-}{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{T} \rightarrow \langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{terminator}}$	[Terminate]
$\frac{\bar{e}_0 \notin \{\text{R+}, \text{R-}, \text{P}, \text{T}, \text{RXN}\} \quad \langle \mathcal{C} : \mathcal{E} : \mathcal{N} \rangle \bar{e} \rightarrow^* \langle \mathcal{C} : \mathcal{E}' : \mathcal{N} \rangle i}{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \bar{e} \rightarrow \langle \mathcal{G} : \mathcal{E}' : \mathcal{N} \rangle \text{produces } i}$	[Coding]
$\frac{e' \in \{\text{activates}, \text{represses}\} \quad \langle \mathcal{C} : \mathcal{E} : \mathcal{N} \rangle \bar{e} \rightarrow^* \langle \mathcal{C} : \mathcal{E}' : \mathcal{N} \rangle \bar{i}'}{\langle \mathcal{C} : \mathcal{E}' : \mathcal{N} \rangle \bar{e}'' \rightarrow^* \langle \mathcal{C} : \mathcal{E}'' : \mathcal{N} \rangle \bar{i}'}$	[Reaction]
$\frac{-}{\langle \mathcal{G} : \mathcal{E} : \mathcal{N} \rangle \text{RXN} \bar{e} e' e'' \rightarrow \langle \mathcal{G} : \mathcal{E}'' : \mathcal{N} \rangle \bar{i} e' \bar{i}'}$	

Figure 8. BioProto interpretation rules for genetic regulatory network specifications.

The semantics of interpreting a dataflow graph fragment are given in Figure 8. Rules [Traverse] and [Recurse] walk the dataflow graph, moving the parsing placeholder token ahead as each element is interpreted. The remainder of the rules perform the minor transformations necessary to map the dataflow graph specification into a transcriptional logic network, looking up and/or creating chemicals in the environment as needed.

Finally, the rules for interpreting Proto expressions are given in Figure 9. These semantics depend on the fact that every primitive, and thus every expression, has precisely one chemical associated as

$\frac{\langle \mathcal{P} : \mathcal{E} : \mathcal{N} \rangle e \rightarrow^* \langle \mathcal{P} : \mathcal{E}' : \mathcal{N}' \rangle i}{\langle \mathcal{P} : \mathcal{E} : \mathcal{N} \rangle (\bar{i} \parallel e e') \rightarrow \langle \mathcal{P} : \mathcal{E}' : \mathcal{N}' \rangle (\bar{i} \bar{i}' \parallel e')}$	[P:Traverse]
$\frac{\langle \mathcal{P} : \mathcal{E} : \mathcal{N} \rangle (p \parallel \bar{e}) \rightarrow^* \langle \mathcal{P} : \mathcal{E} : \mathcal{N}' \rangle i \parallel}{\langle \mathcal{P} : \mathcal{E} : \mathcal{N} \rangle (\bar{p} \bar{e}) \rightarrow \langle \mathcal{P} : \mathcal{E} : \mathcal{N}' \rangle i \parallel}$	[P:Recurse]
$\frac{(p, \bar{v}, e) \in \mathcal{P} \quad \langle \mathcal{G} : \bar{v} \Rightarrow \bar{i}, \phi : \mathcal{N} \rangle e \rightarrow^* \langle \mathcal{G} : \mathcal{E}' : \mathcal{N} \rangle i}{\langle \mathcal{P} : \mathcal{E} : \mathcal{N} \rangle (p \bar{v} \parallel) \rightarrow \langle \mathcal{P} : \phi, \mathcal{E} : \mathcal{N}' \rangle i \phi \parallel}$	[Primitive]
$\frac{i = (\text{High_Basal_Activity_Promoter produces } [\phi, \text{boolean}])}{\langle \mathcal{P} : \mathcal{E} : \mathcal{N} \rangle \text{true} \rightarrow \langle \mathcal{P} : [\phi, \text{boolean}], \mathcal{E} : \mathcal{N}' \rangle i}$	[TRUE]
$\frac{i = (\text{Low_Basal_Activity_Promoter produces } [\phi, \text{boolean}])}{\langle \mathcal{P} : \mathcal{E} : \mathcal{N} \rangle \text{false} \rightarrow \langle \mathcal{P} : [\phi, \text{boolean}], \mathcal{E} : \mathcal{N}' \rangle i}$	[FALSE]
$\frac{\text{for each } x, y, n \in e, i, \mathcal{N}' \quad \langle \mathcal{P} : \mathcal{E} : x \rangle \rightarrow^* \langle \mathcal{P} : \mathcal{E}' : n \rangle y}{\langle \mathcal{P} : \mathcal{E} : \mathcal{N} \rangle \text{let } (\bar{v} e) e' \rightarrow \langle \mathcal{P} : \bar{v} \Rightarrow \bar{i}, \mathcal{E} : \mathcal{N}' \rangle \bar{i} e'}$	[Let]
$\frac{-}{\langle \mathcal{P} : e \Rightarrow \bar{i} \otimes \mathcal{E} : \mathcal{N} \rangle e \rightarrow \langle \mathcal{P} : e \Rightarrow \bar{i} \otimes \mathcal{E} : \mathcal{N} \rangle i}$	[Variable]

Figure 9. BioProto interpretation rules for Proto subset.

its return value. We use ϕ to designate the creation of a new unique return chemical, guaranteed to not collide with any other chemical in the system. As the [Proto:Traverse] and [Proto:Recurse] rules traverse the expression tree, they replace expressions with the chemical return values of their subexpressions.

The [Primitive] rule takes these chemical values, binds it to the arguments of the primitive being invoked, creates a fresh return chemical, and evaluates the associated genetic regulatory network against these chemical bindings. The resulting network is then added to the network accumulated so far. The [TRUE] and [FALSE] rules do the same for a fixed motif implementing their primitive values. Finally, the [LET] rule evaluates subexpressions and binds their chemicals to variables in the environment, where they can be looked up by rule [VARIABLE].

Together, these three sets of rules implement the full current semantics of BioProto.

5. Implementation Details

We have implemented BioProto as a plug-in for the MIT Proto [26] distribution of Proto, which is free and open software. MIT Proto has a flexible architecture designed for extensibility, including the ability to emit code for different platforms.

Figure 10 shows the structure of the MIT Proto compiler, along with the BioProto plug-in: Proto code is first parsed into S-expressions that are then interpreted to produce a dataflow-graph. This dataflow graph is transformed from aggregate-level to local operations (which has no effect on the restricted subset of Proto currently used in BioProto).

Finally, the local dataflow graph is sent to an emitter to be transformed into platform-specific executable code. This defaults to an assembler for the Proto virtual machine [1], but can be switched to plug-in provided platforms by a command-line argument. For BioProto, we thus provide a plug-in emitter that produces a genetic regulatory network as its output. This genetic regulatory network can then either be tested in simulation (as shown for complex networks in [4]) or progress through further realization stages: first a “technology mapping” that determines the DNA sequence or sequences used to realize the GRN, then synthesis and assembly to construct physical samples of those sequences, and finally insertion into living cells for execution. This realization was recently successfully demonstrated with a simple stimulus-response system for both E. coli and mammalian cells in [6].

BioProto requires only one extension over the base language of Proto: the `:grn-motif` annotation. Proto already includes an anno-

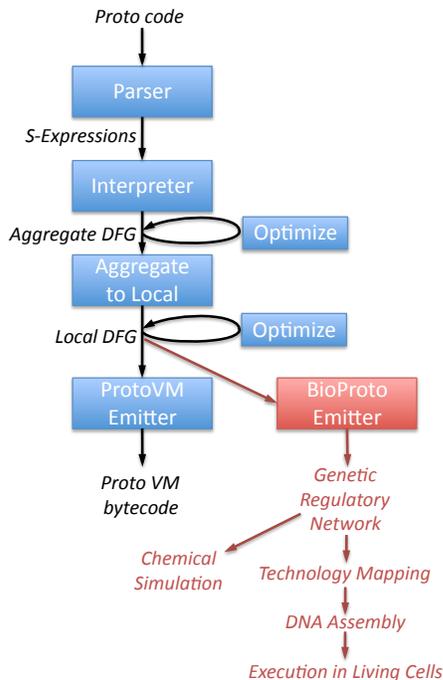


Figure 10. Our BioProto implementation builds on the MIT Proto compiler: earlier stages of the compiler interpret Proto code into a local dataflow graph representation. The emitter stage, which transforms a dataflow graph into an executable implementation, allows selection of alternate target platforms, and BioProto is implemented primarily as an emitter for transforming the dataflow graph IR into a genetic regulatory network, which can then be tested in simulation or further processed for execution in living cells.

tation facility, which allows arbitrary tokens beginning with `:` and optionally followed by an `S-expression`; unrecognized annotations are simply ignored. BioProto’s genetic regulatory network sublanguage is constructed to comply with this format, so all that was necessary was to implement handling of the annotation within the BioProto emitter plugin. Similarly, forced inlining of functions is already an option in MIT Proto, so no change was needed for that.

The restrictions on Proto for BioProto are implemented solely in the emitter: any Proto primitive that the BioProto emitter encounters that does not have a `:grn-motif` annotation produces a compilation error, as does any literal that is not a Boolean. Thus, we simply declare and/or annotate only those primitives that are currently supported; future expansions of BioProto to larger subsets of Proto can thus be handled simply by declaring/annotating the missing primitives or by adding the missing types to the language.

The BioProto emitter also implements a collection of optimization methods, as documented in [4]. These are largely biology-adapted versions of standard heuristic program optimizations (e.g., copy propagation, dead code elimination), but also including some biology-specific optimizations, such as using the fact that chemical concentrations can be superposed to compress NOR patterns.

Hosting BioProto within MIT Proto as a plug-in in this way is advantageous because it means that it can use all of the facilities of Proto without requiring any branch from the original distribution. This also allows a light-weight debugging of BioProto programs, which is important because chemical simulation of a genetic regulatory network is often computationally expensive. Early rounds of debugging, to ensure that the program correctly captures the in-

tent of the designer, can take advantage of emitter modularity and be carried out in the Proto VM rather than with a simulated genetic regulatory network. Then, once the designer is satisfied that the program is correct, they can move forward to more expensive simulations to verify that the genetic regulatory network produced by the BioProto plug-in maintains that correctness.

6. Challenges to Proving Correctness

Correct synthesis of a transcriptional logic network from a BioProto description, according to the semantics presented above, can be proved in a relatively straightforward but lengthy manner. In sketch: the specification of genetic regulatory network fragments has only shallow differences from the transcriptional logic network model that it targets, proof of the well-definedness of the dataflow semantics can be adapted from the proofs in [5], and it only remains to be demonstrated that the “stitching” between these two aspects links variables correctly.

The more important and difficult question, however, is whether any transcriptional logic network can be realized in cells such that it faithfully implements the behavior that has been specified. The root problem here is the fundamental concept of a transcriptional logic network itself. The appropriateness of this model for describing biological systems is much more tenuous than the appropriateness of the digital logic model for describing electronic systems.

First, transcription factors vary wildly in their dynamics and the properties of their regulatory relationships. Selecting particular transcription factors to realize a regulatory network requires matching regulatory relations to find compatible pairings. Although many attempts at quantification have been made (e.g., [10, 15, 20, 28, 33]), making even coarse predictions of the behavior of transcriptional logic networks has been problematic. Important reasons for this include the difficulty in obtaining correct coefficients for chemical reaction models and an incomplete understanding of the “cross-talk” interactions of the components of engineered genetic regulatory networks with each other and the cellular context in which they operate. Recent results based on black-box modeling [14] offer the potential to change this, but there are many challenges and open questions to be dealt with before quantitative prediction of network behavior can be considered reliable.

Even once the behavior of transcriptional regulation is measured and can be used to predict the behavior of a network, transcription factors are often far from ideal, with high noise and low amplification. In many cases, it may be impossible to establish a true digital abstraction with signal restoration ensuring that outputs have better signal than inputs, though some recent results offer the potential for much more “digital” gates [9]. Networks with poor amplification, however, can only have a limited depth of regulatory relations, beyond which the signal to noise ratio is too low to be effective.

Finally, the number of transcription factors available to engineer with is still quite small, meaning that transcriptional logic networks with more than a few elements cannot currently be realized at all. Recent developments in synthetic combinatorial protein families, however, such as TALE [27] and zinc finger [23] proteins are likely to greatly expand the number of available parts in the future.

These are not problems with transcriptional logic networks, but rather come from the general early stage of synthetic biology as an engineering discipline. Similar or worse problems attend other forms of genetic regulatory network design. The bottom line is this: the correctness of a genetic regulatory network designed by BioProto is largely moot unless it can actually be implemented. At present, the state of biological science is such that we generally cannot prove this in any way except by testing particular instances of networks. The challenges preventing broader guarantees, however, are areas of highly active research, however, and enabling progress is likely to be made in the near future.

7. Contributions

This paper has provided the first formal treatment of the BioProto functional dataflow language, analyzing it as a piggy-back domain-specific language on the Proto general-purpose spatial language. We have provided both syntax and semantics for BioProto, showing how these align with a formal model of transcriptional logic networks that can be implemented in engineered biological organisms. We also discuss key engineering decisions in the implementation of BioProto and the pragmatic challenges to proving correctness of designs synthesized with BioProto.

In addition to the utility of this work for synthetic biology, we hope that this paper also provides an accessible case study of domain specific language implementation for computations executed in an environment very different from modern electronic computers, as well as an example of a functional language well suited for its domain not only for reasons of programming succinctness, but also due to the close representational alignment between dataflow models and the physical realization of the program.

Acknowledgments

This work was partially funded by DARPA; views and conclusions contained in this document are those of the authors and not DARPA or the U.S. Government.

References

- [1] J. Bachrach and J. Beal. Building spatial computers. Technical Report MIT-CSAIL-TR-2007-017, MIT, March 2007.
- [2] J. Beal and J. Bachrach. Cells are plausible targets for high-level spatial languages. In *IEEE SASO 2008 Workshops*, 284–291.
- [3] J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, March/April 2006.
- [4] J. Beal, T. Lu, and R. Weiss. Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks. *PLoS ONE*, 6(8):e22490, August 2011.
- [5] J. Beal, K. Usbeck, and B. Benyo. On the evaluation of space-time functions. *The Computer Journal*, 2012. .
- [6] J. Beal, R. Weiss, D. Densmore, A. Adler, E. Appleton, J. Babb, S. Bhatia, N. Davidsohn, T. Haddock, J. Loyall, et al. An end-to-end workflow for engineering of biological networks from high-level specifications. *ACS Synthetic Biology*, 1(8):317–331, 2012.
- [7] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013.
- [8] L. Bilitchenko, A. Liu, S. Cheung, E. Weeding, B. Xia, M. Leguia, J. C. Anderson, and D. Densmore. Eugene: A domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PLoS ONE*, 6(4):e18882, 2011.
- [9] J. Bonnet, P. Subsoontorn, and D. Endy. Rewritable digital data storage in live cells via engineered control of recombination directionality. *PNAS*, 109(23):8884–8889, Jun 2012.
- [10] B. Canton, A. Labno, and D. Endy. Refinement and standardization of synthetic biological parts and devices. *Nature Biotechnology*, 26(7):787–793, 2008.
- [11] D. Chandran, F. Bergmann, and H. Sauro. Tinkercell: modular cad tool for synthetic biology. *J. of Biological Engineering*, 3(1):19, 2009.
- [12] C. Conboy, J. Braff, and D. Endy. Definitions and measures of performance for standard biological parts. Technical Report <http://hdl.handle.net/1721.1/31335>, MIT, March 2006.
- [13] M. J. Czar, Y. Cai, and J. Peccoud. Writing dna with genocad. *Nucleic Acids Research*, 37(Web Server issue):W40–W47, 2009.
- [14] N. Davidsohn, J. Beal, A. Adler, F. Yaman, Y. Li, Z. Xie, and R. Weiss. Accurate predictions of genetic circuit behavior from part characterization and modular composition. In *5th IWBDA*, July 2013.
- [15] T. Ellis, X. Wang, and J. Collins. Diversity-based, model-guided construction of synthetic gene networks with predicted functions. *Nature biotechnology*, 27(5):465–471, 2009.
- [16] M. Elowitz and S. Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767):335–338, 2000.
- [17] T. S. Gardner, C. R. Cantor, and J. J. Collins. Construction of a genetic toggle switch in *escherichia coli*. *Nature*, 403(6767):339–342, 2000.
- [18] T. S. Ham, Z. Dmytriv, H. Plahar, J. Chen, N. J. Hillson, and J. D. Keasling. Design, implementation and practice of jbei-ice: an open source biological part registry platform and tools. *Nucleic acids research*, 40(18):e141–e141, 2012.
- [19] N. J. Hillson, R. Rosengarten, and J. D. Keasling. j5 dna assembly design automation software. *ACS Synthetic Biology*, 1(1), 2012.
- [20] S. Hooshangi, S. Thiberge, and R. Weiss. Ultrasensitivity and noise propagation in a synthetic transcriptional cascade. *PNAS*, 102(10):3581, 2005.
- [21] L. V. Huynh, A. Tsoukalas, M. Koppe, and I. Tagkopoulos. Sbrone: A scalable optimization and module matching framework for automated biosystems design. *ACS Synthetic Biology*, February 2013.
- [22] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.
- [23] A. S. Khalil, T. K. Lu, C. J. Bashor, C. L. Ramirez, N. C. Pyenson, J. K. Joung, and J. J. Collins. A synthetic biology framework for programming eukaryotic transcription functions. *Cell*, 150(3):647–658, August 2012.
- [24] M. A. Marchisio and J. Stelling. Automatic design of digital synthetic gene circuits. *PLoS Comput Biology*, 7(2):e1001083, 02 2011.
- [25] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.
- [26] MIT Proto. MIT Proto. software available at <http://proto.bbn.com/>, Retrieved June 15th, 2013.
- [27] R. Morbitzer, P. Romer, J. Boch, and T. Lahaye. Regulation of selected genome loci using de novo-engineered transcription activator-like effector (tale)-type transcription factors. *Proc Natl Acad Sci*, 107(50):21617–21622, December 2010.
- [28] V. K. Mutalik, J. C. Guimaraes, G. Cambray, C. Lam, M. J. Christoffersen, Q.-A. Mai, A. B. Tran, M. Paull, J. D. Keasling, A. P. Arkin, and D. Endy. Precise and reliable gene expression via standard transcription and translation initiation elements. *Nature Methods*, 10:354–360, April 2013.
- [29] M. Pedersen and A. Phillips. Towards programming languages for genetic engineering of living cells. *Journal of the Royal Society Interface*, 6(4):S437–S450, 2009.
- [30] P. E. M. Purnick and R. Weiss. The second wave of synthetic biology: from modules to systems. *Nature Reviews Molecular Cell Biology*, 10(6):410–422, 2009.
- [31] S. Richardson, S. Wheelan, R. Yarrington, and J. Boeke. Genedesign: rapid, automated design of multikilobase synthetic genes. *Genome Res.*, 16(4):550–6, April 2006.
- [32] M. Viroli, J. Beal, and K. Usbeck. Operational semantics of Proto. *Science of Computer Programming*, 2012. .
- [33] R. Weiss. *Cellular Computation and Communications using Engineered Genetic Regulatory Networks*. PhD thesis, MIT, 2001.
- [34] R. Weiss, G. E. Homsy, and T. F. Knight Jr. Toward in vivo digital circuits. In *Evolution as Computation*, pages 275–295. Springer, 2002.
- [35] B. Xia, S. Bhatia, B. Bubenheim, M. Dadgar, D. Densmore, and J. C. Anderson. Developers and user’s guide to clotho v2.0 a software platform for the creation of synthetic biological systems. *Methods in Enzymology*, 498:97–135, 2011.