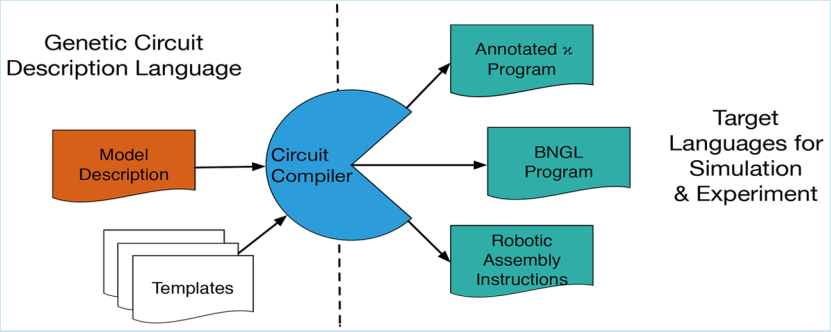
ACS SyntheticBiology
Research Article
Cite This: ACS Synth. Biol. 2018, 7, 2812−2823
pubs.acs.org/synthbio

A Genetic Circuit Compiler: Generating Combinatorial Genetic Circuits with Web Semantics and Inference

William Waites,*[,†] Göksel Mısırlı,[‡] Matteo Cavaliere,[§] Vincent Danos,[∥,†] and Anil Wipat[⊥]

[†]School of Informatics, University of Edinburgh, Edinburgh EH8 9YL, U.K.
[‡]School of Computing and Mathematics, Keele University, Newcastle ST5 5BG, U.K.
[§]School of Computing & Mathematics, Manchester Metropolitan University, Manchester M15 6BH, U.K.
[∥]École Normale Supérieure, Paris, CNRS, 75005 Paris, France
[⊥]School of Computing Science, Newcastle University, Newcastle upon Tyne NE1 7RU, U.K.

S *Supporting Information*



ABSTRACT: A central strategy of synthetic biology is to understand the basic processes of living creatures through engineering organisms using the same building blocks. Biological machines described in terms of parts can be studied by computer simulation in any of several languages or robotically assembled *in vitro*. In this paper we present a language, the Genetic Circuit Description Language (GCDL) and a compiler, the Genetic Circuit Compiler (GCC). This language describes genetic circuits at a level of granularity appropriate both for automated assembly in the laboratory and deriving simulation code. The GCDL follows Semantic Web practice, and the compiler makes novel use of the logical inference facilities that are therefore available. We present the GCDL and compiler structure as a study of a tool for generating κ-language simulations from semantic descriptions of genetic circuits.

KEYWORDS: *semantic web, inference, program generation, synthetic biology, genetic circuits*

Synthetic biology extends classical genetic engineering with concepts of modularity, standardization, and abstraction drawn largely from computer engineering. The goal is ambitious: to design complex biological systems, perhaps entire genomes, from first principles.[1] This enterprise has met with some success such as microbial drug synthesis,[2,3] production of new biofuels,[4] and alternative approaches to disease treatment.[5] However, most applications are still small and mostly designed manually.

The are several obstacles to designing more complex circuits. The design space of potential circuits is very large. Even when a design is chosen, there is large *a priori* uncertainty about what its behavior will be. In many cases the available information about molecular interactions in a cell is incomplete. A secondary obstacle is that designs can be brittle and very sensitive to the host environment in which they execute. In this context computational techniques become important for identifying biologically feasible solutions to problems of biological system synthesis. Beyond the challenges of the huge design space and associated uncertainties, writing these programs by hand is time-consuming and error prone, and there are very few tools available for verification and debugging them. Descriptions of models in terms of simulation code are tightly coupled to the language of the simulation program, and it may be difficult or impossible to use a different interpreter without completely rewriting the code.

We solve these problems by providing a high-level, modular, implementation-independent language for describing gene circuits called the Genetic Circuit Description Language (GCDL) and a compiler called Genetic Circuit Compiler (GCC). We use a strategy of contextual reasoning to obtain flexible output from this succinct input, and *templates* to support any number of output languages and modeling granularities. An overview of information flow through the compiler is shown in Figure 1. We demonstrate the utility of this approach by

Received: May 13, 2018
Published: November 8, 2018

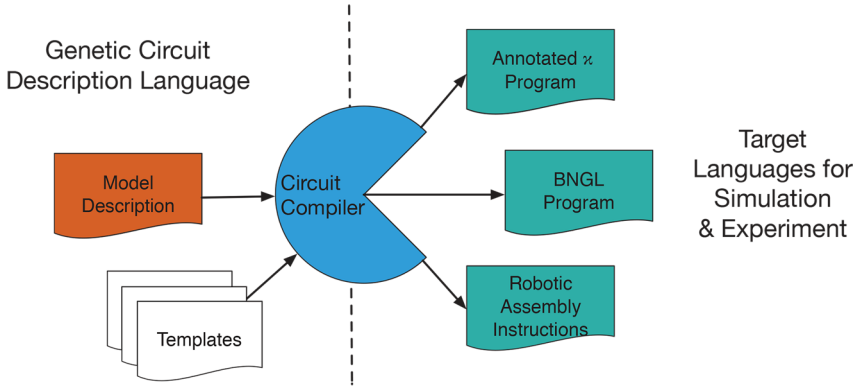


Figure 1. High-level data flow through the compiler. The compiler for synthetic gene circuits takes a model description written in GCDL and, using language-appropriate appropriate templates, creates code for simulation and laboratory assembly. We have implemented templates for annotated-κ for the KaSim software, and envision similar for the BNGL as well as SBOL.

describing, compiling, and simulating a complete genetic circuit, the well-known Elowitz repressilator.[6] The compiler and example code are available at https://github.com/rulebased/composition.

Code generation from this high-level description to a low-level language for simulation greatly reduces the scope for error in coding simulations. Because the language is implementation-independent, it is not tightly coupled to any particular interpreter or hardware. In this way GCDL facilitates *evergreen models*, models that are specified sufficiently well to be unambiguous but not so specifically that they can only be executed or constructed in one software package or environment.

Domain specific languages and examples of compilers processing these languages have previously been shown.[7−10] These languages are designed to allow for simulations using a particular methodology such as solving systems of ordinary differential equations or using Monte Carlo simulation. Unlike previous approaches, we emphasize the use of abstraction to facilitate *retargeting* or production of output suitable for different simulation environments and techniques as well as automated circuit assembly in the laboratory from a single description. Compiler targets are implemented using conditional inference, defining the semantics of the terms used in the description of the circuit in a way that is determined by the desired output type. The design of the compiler is general, and not limited to the present context of genetic circuits. The design is shown schematically in Figure 2.

The GCDL is an RDF[11] vocabulary and attendant inference rules, which facilitates gathering and collation of information about the constituent parts of a genetic circuit.[12] The output programs can be specialized to various languages, such as the KaSim flavor of κ,[13,14] BioNetGen's BNGL,[15,16] other representations such as SBOL,[17] or indeed whichever form is required by robotic laboratory equipment that assembles circuits *in vitro*. This output flexibility is accomplished using *templates* that use facts derived by inference rules[18] from the input model.

We now proceed as follows. We begin with an overview of those aspects of synthetic biology and genetic engineering that are necessary to contextualize our work. Next, we explain the representation of this kind of genetic circuit model in GCDL, this is the main input to the compiler. In order to understand the desired output of the compiler, we then illustrate how these constructs are represented as rule-based code for the κ language simulator, KaSim. There follows a discussion of how the

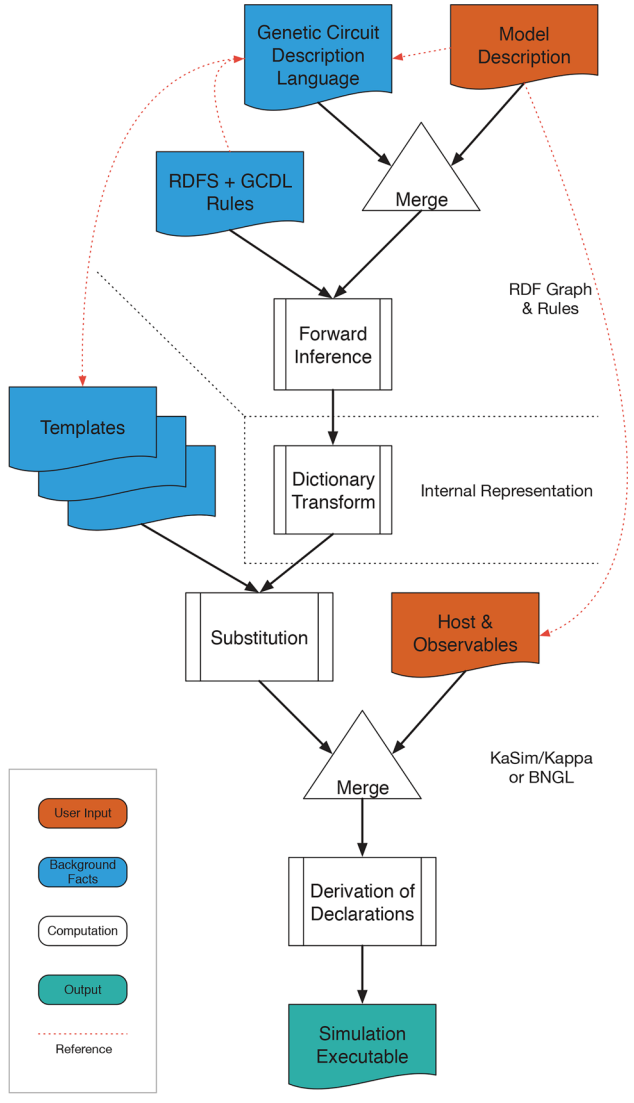


Figure 2. Detailed data flow through the compiler. This illustrates the use of inference to expand the GCDL model to derive consequent information appropriate to producing the next stage of output in the specific target language.

DOI: 10.1021/acssynbio.8b00201
ACS Synth. Biol. 2018, 7, 2812−2823

compiler infers the executable model from the input description. Finally, we discuss some possible uses and limitations of our technique.

BACKGROUND

Rule-Based Modeling of Genetic Processes. A weakness of reaction-based methods for modeling the processes of transcription, translation, and the production of chains of proteins is that they require chemical species for each bound state of the reagents. This in turn requires specification of reactions for each combination of these reagents. To solve this problem of needing combinatorially many reactions to describe substantially the same process, a generalization of reactions called *rules* are used.[19−21]

In the rule-based representation, *agents* correspond to reagents and they can have slots or *sites* that can be bound, or not. They can also have internal state. Unlike reactions that have no preconditions apart from the presence of the reagents, with rules, a configuration of the sites—bound in a particular way, bound in some way, unbound, or unspecified—is a precondition for the application of the rule. A rule may rearrange the bonds, creating or destroying them, without the need to invent new agents in order to represent different configurations of a given set of molecules.

The reader should note that the word *rule* is used in two distinct senses in this article. The first is as we have just described. The second is in the sense of *inference rule* as used in logic and in particular the way in which we deduce executable rule-based models from their declarative representations in RDF.

The κ Language. To briefly illustrate the essentials of rule-based modeling we will use the language of the Kappa simulation software, KaSim.[14] An agent declaration and rule expressing the formation of a polymer can be written as

```
%agent: A(d,u)

'binding' A(u[.]), A(d[.]) -> A(u[1]), A(d[1]) @k
```

We can gloss this as an agent with two sites, *u* and *d* for upstream and downstream, and a rule. The rule concerns two agent patterns one of which has an unbound upstream site, and the other an unbound downstream site, and the action of the rule is to bind them, the notation [1] denoting the bond. This process happens at some rate *k*.

The state of the other site of each agent is left unspecified, so implicit in this rule is the possibility that either or both the agents may already be bound to others and so part of arbitrarily long chains. In other words this expression covers not only two monomers joining together but an *n*-mer and an *m*-mer for arbitrary *n* and *m*. This is the essence of the expressive advantage that rule-based modeling provides. To express a similar concept using a reaction network would in fact require infinitely many reagents for every possible *n* (and *m*) and infinitely many reactions for every possible combination.

Biological Parts and Annotation. For efficiency, and economy of representation, we claim that the description of a computational model should include minimum information necessary for simulation. However, in order to use these models in an automated design process, additional metadata, or *annotations*, about the meaning of different modeling entities is needed.[12] Annotation facilitates the drawing of specific parts from a database such as the Virtual Parts Repository.[22] Models in that database are annotated with machine-readable metadata intended for combination into larger models. Myers and his colleagues have used annotations to derive simulatable models from descriptions of genetic circuits[23] and vice versa,[24] though these use reaction-based techniques and so inherit the poor scaling properties of that method.

To facilitate the *in silico* evaluation of potential synthetic gene circuits, a library of descriptions of genetic parts, together with their modular models is suggested in the literature.[22,25] These parts are intended to be large enough to have a particular meaning or function (*i.e.*, larger than individual base pairs) but not so large that they lack the flexibility to be recombined (*i.e.*, entire genes). Thus, we are concerned with coding sequences for particular proteins, promoters that, when activated, start the transcription process, operators that activate or suppress promoters according to whether they are bound or not by a given protein, and a small number of other objects. A sequence of these objects is a genetic circuit, and our goal is to have a good language for describing such sequences.

Annotation in this setting means machine-readable descriptions of entities of biological interest. This is done with statements, triples of the form (subject, predicate, object) according Semantic Web standards.[11,26] Entities are identified with Universal Resource Identifiers (URIs).[27] This provides the dual benefit of globally unique identifiers for entities and a built-in mechanism for retrieving more information about them providing that some care is taken to publish data according to best practises.[28,29] Large bodies of such information about biologically relevant information are published on the Web[30,31] and the use of Semantic Web standards for annotating our models allows us to express how an entity in a model description corresponds to a real world protein, or gene sequence or other entity.

The Semantic Web also affords us a technical advantage: inference rules. These can be either explicit as in Notation3[32,33] or implicit as in OWL Description Logics.[34,35] In either case this facility makes it possible, given a set of statements, to derive new statements according to inference rules. We use this to improve the ergonomics of our high-level language: while the compiler itself will make use, internally, of a large amount of information, we do not expect the user to supply it in painstaking detail. Rather, we allow the user to specify the minimum possible and provide rules to derive the necessary detail. Inference rules provide for both economy of representation for the high-level model description and flexibility for the different implementations.

A LANGUAGE FOR SYNTHETIC GENE CIRCUITS

This section describes the GCDL, the high-level language for describing genetic circuits made from standard biological parts.[22,25] We begin by stating the properties that we want in such a language and showing how we achieve them. There follows a synopsis of the vocabulary terms essential to the language. Finally, we illustrate salient language features applied to example circuits.

Desired Language Features. Our desired language features for high-level representation of a genetic circuit are as follows:

1. Sufficiency; there should be enough information to derive executable code for the circuit.

DOI: 10.1021/acssynbio.8b00201
ACS Synth. Biol. 2018, 7, 2812−2823

2. Identifiability; it should be possible to determine to which biological entities (DNA sequences, proteins) the representation refers.
3. Extensibility; it should be straightforward to add information or constructs that are not presently foreseen.
4. Generality; there should be no requirement that information about biological parts comes from any particular set or source.
5. Concision; there should be a minimum of extraneous detail or syntax.

The third and fourth requirements are readily met by using RDF as the underlying data model. The *open world* presumption[36] means that adding information as necessary is straightforward. The use of URIs[27] that can be dereferenced to obtain the required information means that information from different web-accessible databases can be obtained, mixed, and matched as desired. The use of URIs goes some way toward meeting the second requirement, albeit with some well-known caveats.[37]

The first and last of the desired features are the primary areas of innovation of the present work. We suggest (but do not require) the use of Turtle[38] or indeed Notation3[18] as the concrete surface syntax for writing models. This goes some way toward a representation that is intelligible by humans. Even then, we aim to minimize what needs to be written, and we do this using inference rules—if a needed fact can be derived from the model under the provided rule-set, it is unnecessary to write it explicitly in the model. Indeed it may even be undesirable to do so since it is a possible source of errors, for example some kinds of assertions may be correct in the context of some output types and incorrect in others. We aim for a minimal, yet complete under the inference rules, description of the model.

Vocabulary Terms. New terms introduced in this paper have the prefix gcc, which can be read as the "Genetic Circuit Compiler" vocabulary. The list of terms is reproduced in Table 1 and their complete definitions are given together with the accompanying rules in the Supporting Information. The GCDL is the union of terms from the *gcc* namespace with those from the Rule-Based Model Ontology (RBMO) that we previously defined,[39] together with terms from the Simple Knowledge Organization System (SKOS)[40] vocabulary, RDF Schema (RDFS),[35] and Resource Description Framework (RDF).[11]

Model Description. To illustrate the syntax of the high-level language, we use the well-known Elowitz repressilator shown diagrammatically in Figure 3a. The complete model can be found in the Supporting Information as well as distributed in the examples/ subdirectory of the compiler distribution. Also included with the compiler is a hand-assembled implementation of this circuit for comparison. A sample trace produced by generated program is shown in Figure 3b. Figure 4 shows a description of this the core of the model, in the GCDL. Some bibliographic metadata is included, using the standard Dublin Core[41] vocabulary, as well as a generic pointer (rdfs:seeAlso) to a publication about this model.

The term gcc:prefix is necessary in every model; it instructs the compiler that any entities that it creates should be created under the given prefix. Ultimately annotated rules will be generated for the low-level representation and the annotated entities require names. To give them names, a namespace is required, and this is how it is provided.

Next there is a gcc:include statement. This is a facility for including extra information in the low-level language. Extra information typically means rules for protein−protein interactions, which are beyond the scope of the current work, and as such it is simply supplied as a program fragment in the output language. This corresponds roughly to calling an assembly or machine language routine to perform a specialized task when programming a computer in a high-level language like C.

There follows initialization for specific variables. In this case these are the copy numbers for RNA polymerase molecules and ribosomes. These are denoted using rbmo:agent because of our choice to support rule-based modeling for greater generality than reaction-based methods. Finally, the circuit itself is specified. The argument or object is an rdf:List that simply contains identifiers for the parts, in order.

The circuit itself is now defined. However, at this juncture, we simply have a list of parts without having specified what they are or what their intended behavior is. To obtain a working model, we need more.

A Part Description. A simple example of a part description is shown in Figure 5. This is a coding sequence, as is clear from the type annotation on the part. It codes for a particular protein, specified with gcc:protein. This term is specific to proteins because under normal circumstances other kinds of part do not code for proteins. It is given a part symbol using gcc:part because the output language will not typically permit the use of URIs as identifiers, so this symbol *via* the implied skos:prefLabel[40] is what will appear instead. The protein produced by this coding sequence is also specified and linked using gcc:protein. It too is given a label using skos:prefLabel for the same reason, and its degradation rate is also specified with gcc:proteinDegradation-

Table 1. Selected Terms from the GCC Vocabulary

classes	
gcc:Part	Generic biological part
gcc:Operator	Operator
gcc:Promoter	Promoter
gcc:RibosomeBindingSite	Ribosome Binding Site
gcc:CodingSequence	Coding Sequence
gcc:Terminator	Terminator
gcc:Token	Token or symbol in a template
predicates	
gcc:include	Include a low-level model fragment
gcc:prefix	The prefix to use for generated annotations
gcc:init	Specifies initial copy numbers
gcc:part	Links a part to its token or symbol
gcc:overlaps	Indicates that two parts overlap (symmetric)
gcc:linear	Linear circuit type
gcc:circular	Circular circuit type
gcc:transcriptionFactor	Relates an operator to its transcription factor
gcc:transcriptionFactorBindingRate	Various rates
gcc:transcriptionFactorUnbindingRate	
gcc:rnapBindingRate	
gcc:rnapUnbindingRate	
gcc:rnapRNAUnbindingRate	
gcc:ribosomeBindingRate	
gcc:ribosomeRNAUnbindingRate	
gcc:ribosomeProteinUnbindingRate	
gcc:transcriptionInitiationRate	
gcc:transcriptionElongationRate	
gcc:translationElongationRate	
gcc:rnaDegradationRate	
gcc:proteinDegradationRate	

DOI: 10.1021/acssynbio.8b00201
ACS Synth. Biol. 2018, 7, 2812−2823

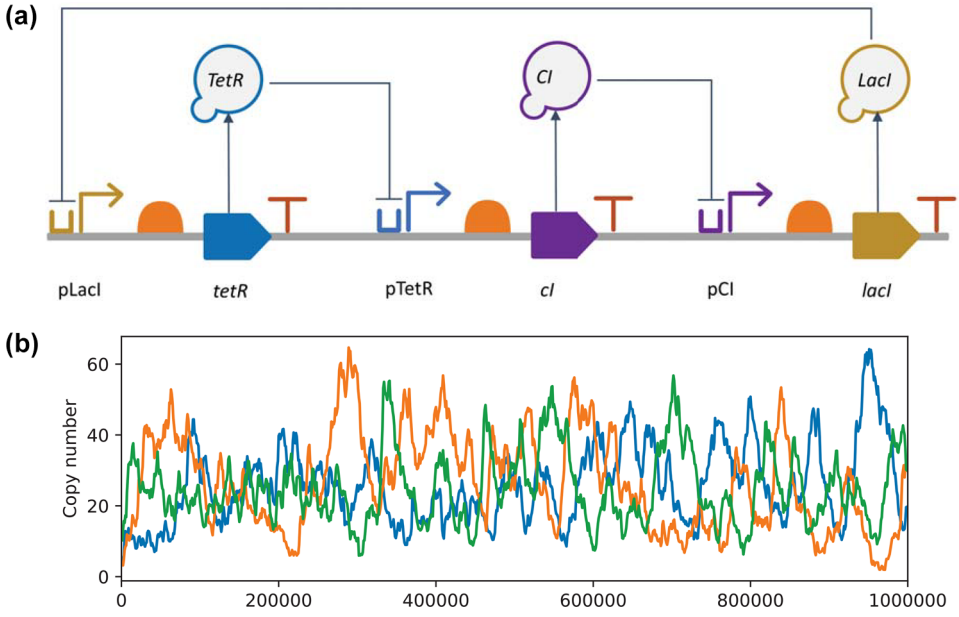


Figure 3. Diagram and sample simulation results of the Elowitz repressilator. (a) An example genetic circuit: the Elowitz repressilator. It is a negative feedback oscillator. The circuit is arranged linearly. Protein production and inhibitory protein−operator relationships are shown using the SBOL visual standard. (b) Sample simulation data from a program produced by the compiler showing the expected oscillations. Note in particular the relatively small copy numbers of the proteins for which stochastic simulation in the κ language is well suited.

```
## Model declaration
:m a rbmo:Model;
  ## bibliographic metadata
  dct:title "The Elowitz repressilator constructed from BioBrick parts";
  dct:description "Representation of the Elowitz repressilator given in the Kappa BioBricks Framework
       book chapter";
  rdfs:seeAlso <http://link.springer.com/protocol/10.1007/978-1-4939-1878-2_6>;
  gcc:prefix <http://id.inf.ed.ac.uk/rbm/examples/repressilator#>;
  ## include the host environment
  gcc:include <.../host.ka>;
  ## initialisations
  gcc:init
    [ rbmo:agent :RNAp; gcc:value 700 ],
    [ rbmo:agent :Ribosome; gcc:value 1000 ];
  ## The circuit itself, a list of parts
  gcc:linear (
    :R0040o :R0040p :B0034a :C0051 :B0011a
    :R0051o :R0051p :B0034b :C0012 :B0011b
    :R0010o :R0010p :B0034c :C0040 :B0011c
  ).
```

Figure 4. Example model for a synthetic gene circuit, Elowitz repressilator.

```
:C0012 a gcc:CodingSequence;
    gcc:label "Coding sequence for LacI";
    gcc:part "C0012";
    gcc:protein :P0010;
    gcc:proteinDegradationRate 0.0001.

:P0010 a gcc:Protein;
    bqbiol:is uniprot:P03023;
    skos:prefLabel "P0010";
    rdfs:label "LacI".
```

Figure 5. A coding sequence part description from the repressilator model. Notice how the coding sequence is linked to the protein that it codes for.

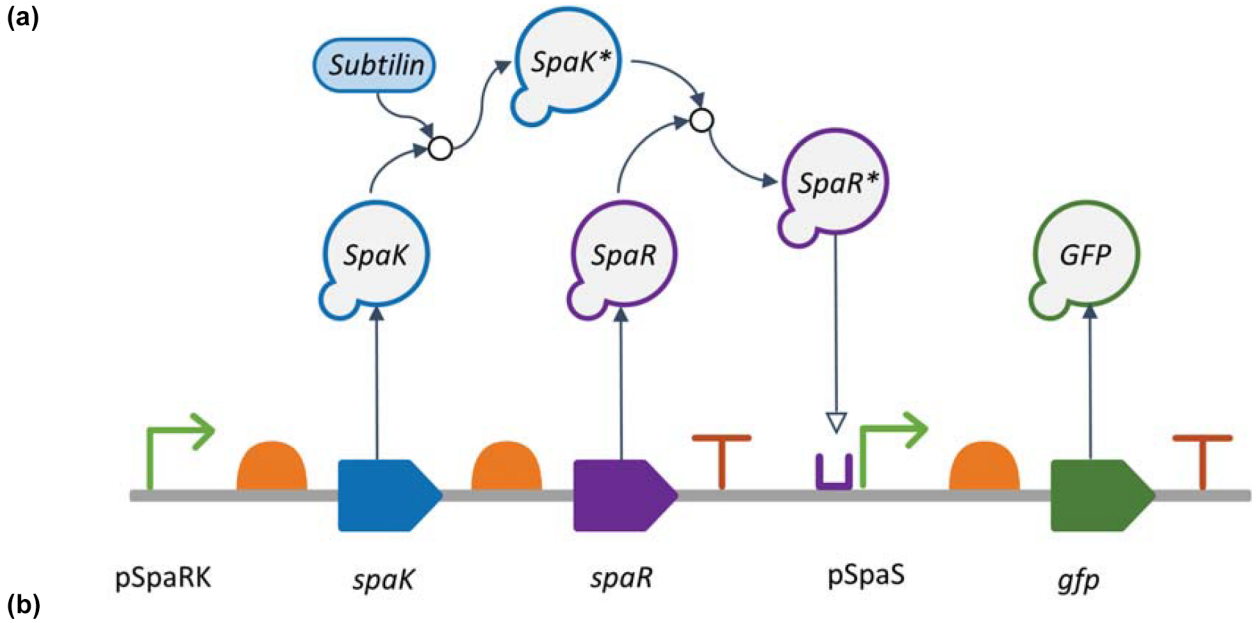
Rate. It is equally possible to specify the rates for transcription and translation in a similar manner though not shown here. In

DOI: 10.1021/acssynbio.8b00201
ACS Synth. Biol. 2018, 7, 2812−2823

```
:R0040o a gcc:Operator;
  rdfs:label "TetR activated operator";
  gcc:part "R0040o";
  gcc:transcriptionFactor :P0040;
  gcc:transcriptionFactorBindingRate 0.01;
  gcc:transcriptionFactorUnbindingRate 0.01.

:R0040p a gcc:Promoter;
  rdfs:label "TetR repressible promoter";
  gcc:part "R0040p";
  gcc:rnapBindingRate
    [
      gcc:upstream ([ a rbmo:BoundState;
                      rbmo:stateOf :R0040o ]);
      gcc:value 7e-7
    ], [
      gcc:upstream ([ a rbmo:UnboundState;
                      rbmo:stateOf :R0040o ]);
      gcc:value 0.0007
    ].
```

Figure 6. An operator and promoter from the repressilator model. The binding rates for the promoter depend on the state of the adjacent operator.




```
:m a rbmo:Model;
    dct:title "Subtilin Receiver Two-Component System";
    gcc:include <.../subtilin-host.ka>;
    gcc:linear (
        :pSpaRK :RBSa :spaK :RBSb :spaR :Ta
        :pSpaS  :RBSc :gfp :Tb
    ).
```

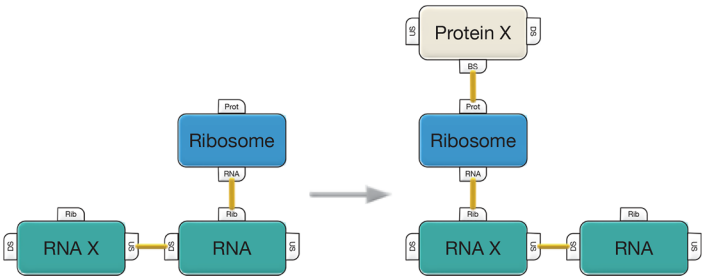
Figure 7. Representations of the Subtilin Receiver model. (a) Diagram of the subtilin genetic circuit. The figure shows the multirelay phosphorylation, and hence the activation, of SpaR TFs to induce the downstream gene expression. As a result, GFP reporter proteins are produced in the presence of Subtilin molecules. (b) Corresponding semantic model.

practice, rates are known primarily from experiment, and this is an important reason to have accessible databases or repositories of part specifications.

Importantly, following the practice in our previous paper on rule annotation,[39] a weak identity assertion is made with identifiers in external databases for the parts. This uses bqbiol:is instead of owl:sameAs because the strong replacement semantics (Leibniz's Law[42]) of the latter can yield unwanted inferences when terms are not used perfectly rigorously.[37] This weaker identify assertion permits the identification of the :P0010 in the example with the identifier for the protein in the well-known UniProt[31] database.

A More Complex Part Description. A more involved example demonstrating how an operator−promoter combina-

DOI: 10.1021/acssynbio.8b00201
ACS Synth. Biol. 2018, 7, 2812−2823




```
'coding-sequence-translation'
    RNA(type{X}, us[2], bs[.]),
    RNA(ds[2], bs[1]),
    Ribosome(rna[1], protein[.]),
    .
->
    RNA(type{X}, us[2], bs[1]),
    RNA(ds[2], bs[.]),
    Ribosome(rna[1], protein[3]),
    P(type{X}, bs[3])
@k
```

Figure 8. Translation of the RNA segment corresponding to a coding sequence to produce a protein.

tion is encoded is shown in Figure 6. Here we have an operator with the rates for binding and unbinding of the transcription factor specified explicitly. If the operator is bound by the transcription factor, the neighboring promoter is repressed—an RNA polymerase will not be able to bind. By contrast if the operator is unbound, the promoter will accept binding of RNA polymerase easily and frequently. The language supports an arbitrary amount of operator context for operators and promoters enabling the specification of complex regulatory structures such as combinatorial logic gates[43−45] and some forms of cooperative binding.

The transcription factor is specified by using gcc:transcriptionFactor to refer to the protein that will turn the operator on or off. Like gcc:protein for coding sequences, the term is unique to operators.

The promoter comes next and it is the most complex part to specify. Because the rate for binding of RNA polymerase depends on the state of the operator, two rates must be specified. States of the nearby parts are specified using the rbmo vocabulary, which makes available the full range of expressiveness for rule-based output languages. For generality, a list of parts, upstream or downstream on the DNA strand may be specified along with their states. This enables a promoter to be controlled by two or more operators. The rate itself in this case is given with gcc:value for each case.

Host and Protein−Protein Interactions. The language can also support protein−protein interactions in a basic way. To see why these are useful, consider an example from the engineering of a bacterial communication system where the subtilin molecule is used to control population level dynamics. Cells have the receiver device[22,46] to sense the existence of subtilin, and the reporter device to initiate downstream cellular processes (Figures 7a and 7b). In the subtilin receiver, the interactions among the proteins produced by translation and the operator−promoters are mediated by a cascade reaction initiated by the subtilin molecule. Subtilin combines to phosphorylate the *SpaK* protein, which in turn phosphorylates the *SpaR* protein that finally binds to the promoter that controls the emission of a fluorescent green protein.

While the genetic circuit can straightforwardly be described similarly to the previous repressilator example, the protein−protein interactions cannot. We do not attempt here to model these interactions in the GCDL though a future extension could do so. Instead we simply allow for inclusion of the relevant program, as a file in the output language (in this case κ-language). It is possible to supply arbitrary code in the low-level language using the gcc:include term. This facility makes it feasible to represent such genetic circuits, which depend strongly on the host environment in order to operate.

Protein Fusion. It is also worth noting that this example illustrates that in the high-level language it is immediately possible to represent devices that produce chains of proteins. This is known as protein fusion and is interesting for some applications.[47] A chain of proteins is produced by adding adjacent (and appropriate) coding sequences. It is enough to simply list the coding sequences in the circuit; nothing else need be done.

Other Parts. The descriptions for the other kinds of biological parts, terminators, coding sequences, follow a similar pattern. There are terms for specifying the rates for the rules in which they participate, and a few specialized terms according to the function of the specific part. It is possible to find out the available terms by inspecting the gcc vocabulary included in the Supporting Information.

OUTPUT REPRESENTATION

We now briefly consider the form of the output representation. By using different templates, the compiler can produce output in different languages. We focus on rule-based representations here and use the language of the KaSim simulator[14] for concrete illustration as it is widely adopted for stochastic simulation of rule-based models.[48] The rule-based modeling approach is merely outlined here and follows that used in Kappa BioBricks Framework KBBF[48] closely. We stress that though output as executable program in the KaSim language is demonstrated here, alternative rule-based representations like BioNetGen are equally possible as are descriptions in a language like SBOL as input to an experimental process in the laboratory. A more detailed account of the modeling methodology and corresponding output can be found in the Supporting Information.

The real work of modeling the transcription and translation machinery is done with sliding rules. Figure 8 shows how this works for the creation of a protein from a coding sequence. This is our first example of a rule where though the adjacent part figures explicitly in the rule, its *type* does not. It is sufficient to know that it is a piece of RNA. In this case, two pieces of RNA are involved, the part that is central to this rule corresponds to the coding sequence for *X*. It is adjacent to another piece of RNA, and the ribosome slides from one to the other (to the left, where sliding on DNA happens, as we will see next, to the right) and in the process, emits a protein of type *X*.

DOI: 10.1021/acssynbio.8b00201
ACS Synth. Biol. 2018, 7, 2812−2823

```
## Auto-generated generic part {{ name }}
{% include "header.ka" %}
{% import "context.ka" as context with context %}
{% import "meta.ka" as meta with context %}

{% include "transcription_elongation.ka" %}
{% include "transcription_termination.ka" %}
{% include "translation_chain.ka" %}
{% include "translation_elongation.ka" %}
{% include "translation_termination.ka" %}
{% include "host_maintenance.ka" %}
```

```
{% set rule = "%s-translation-chain" % name %}
//
//^ :{{ rule }} a rbmo:Rule;
//^   bqbiol:isVersionOf go:GO:0006415;
{{ meta.rule() }}{# #}
//^   rdfs:label "{{ name }} formation of \
//^               translational chains, due to \
//^               gene fusion or leakiness of \
//^               stop codons".
// {{ name }} formation of translational chains,
// due to gene fusion or leakiness of stop codons
//
'{{ rule }}' \
    RNA(ds[2], bs[1]), \
    Ribosome(rna[1], protein[3]), \
    RNA(type{{ curly(name) }}, us[2], bs[.]), \
    Protein(ds[.], bs[3]), . \
-> \
    RNA(ds[2], bs[.]), \
    Ribosome(rna[1], protein[3]), \
    RNA(type~{{ name }}, us[2], bs[1]), \
    P(ds[4], bs[.]), \
    P(type{{ curly(name) }}, us[4], bs[3]) \
    @{{ translationElongationRate }}
```

Figure 9. Template examples. On top is the template for a generic part, and it references several other templates, one of which, translation_chain.ka, is reproduced on bottom.

GENETIC CIRCUIT COMPILER

Having described the GCDL in some detail, we now briefly sketch our implementation of the compiler. Many compiler implementations are possible; ours innovatively combines the logical inference that is native to the semantic web with the use of templates to generate the target program. The templates define standard models for each type of part in a given output language. Different output languages or model granularities are achieved by choosing a different set of templates. The overall information flow through the compiler is illustrated in Figure 1.

Our strategy is to first gather all the input statements and background facts that are asserted by the various vocabularies in use. In the first inference step, standard RDF rules are used to make available consequent facts that will be needed to produce the ultimate result. The result is a program in a language such as κ and not RDF, and which uses local variable names and not URIs, so the materialized facts are transformed into a suitable internal representation. Substitution into templates is done next, and finally the result is postprocessed to derive any remaining program directives that are only knowable once the complete circuit is assembled.

It is interesting to consider that the entire compiler can be thought of as implementing a kind of inference quite different from what is commonly used with the Semantic Web. The consequent, the executable model, is in a different language from the antecedent, the declarative description. Through the use of embedding annotations, however, the original model is nevertheless carried through to the output, and is unambiguously recoverable. There is thus an arrow from the space of declarative models in RDF to the space of annotated executable models. There is an arrow in the other direction that forgets the executable part and retains the declarative part. In an important sense, the two representations contain the same information, only that the executable model has more materialized detail in order that it may be run.

Semantic Inference. The input from the user is the model description in the high-level language as described above. This description uses terms from, and makes reference to the gcc and rbmo vocabularies. The *meaning* of these terms, in the context of deriving an equivalent version of the program in the low-level language, is given by the companion inference rules. This is a somewhat subtle concept so let us illustrate what it means. Consider the statement

DOI: 10.1021/acssynbio.8b00201
ACS Synth. Biol. 2018, 7, 2812−2823

```
gcc:transcriptionFactor a gcc:Token;
    skos:prefLabel "transcriptionFactor".
gcc:transcriptionFactorBindingRate a gcc:Token;
    skos:prefLabel "transcriptionFactorBindingRate";
    gcc:default 1.0.
gcc:transcriptionFactorUnbindingRate a gcc:Token;
    skos:prefLabel "transcriptionFactorUnbindingRate";
    gcc:default 1.0.

gcc:Operator rdfs:subClassOf gcc:Part;
    gcc:kappaTemplate rbmt:operator.ka;
    gcc:bnglTemplate rbmt:operator.bngl;
    gcc:tokens
        gcc:transcriptionFactor,
        gcc:transcriptionFactorBindingRate,
        gcc:transcriptionFactorUnbindingRate.
```

Figure 10. Specification in the gcc vocabulary of a gcc:Operator and associated terms.

```
:R0040a a gcc:Operator.
```

This statement gives the type of :R0040a as gcc:Operator.

The implications of this statement allows to identify the correct template to use for this part, found from information provided by the gcc vocabulary. Indeed, as a background fact, we have

```
gcc:Operator gcc:kappaTemplate rbmt:operator.ka.
```

or in other words that a gcc:Operator corresponds to the template rbmt:operator.ka. We also have an inference rule, provided with the gcc vocabulary that says

```
{ ?part a [ gcc:kappaTemplate ?template ] } => { ?part gcc:kappaTemplate ?template }.
```

In the Notation 3[32,33] language this means that "for all ?parts that has a type that corresponds to a kappa ?template, that ?part itself corresponds to that ?template". Alternatively,

$$\text{type}(p, x) \wedge \text{kappa}(x, t) \rightarrow \text{kappa}(p, t)$$

It would have been perfectly possible to explicitly write what template should be used for each part in the high-level model description. That is not desirable because it would leak implementation details of the compiler into what ought to be an implementation-independent declarative description.

The above rule, and others like it serve to elaborate the high-level description into a more detailed version suitable for the next stage of the compiler and relieve the user of the need to supply the extra details. All implications that can be drawn under the rdfs inference rules and the gcc specific rules are drawn and become part of the in the in-memory RDF storage as the transitive closure of the rules (given the background facts and the provided model facts).

Internal Representation. The output of the first stage of the compiler contains all the information necessary to completely describe the output, but it is not in a convenient form for providing to the template rendering engine. Our implementation choice for the compiler is the Jinja2[49] rendering engine. This means that the appropriate data structure is a dictionary or associative list that can be processed natively by these tools without need of external library. The required internal representation is built up by querying the in-memory RDF storage for the specific information required by the templates.

Our implementation does not require modification when new terms are added to the vocabulary and templates. To add support for a new kind of part it is necessary to write a new template for it and possibly add some terms to the vocabulary but does not require changing the compiler software itself. What makes this possible are the inference rules described in the previous section. The queries on the RDF storage that produce the internal representation are posed in terms of the *consequents* of the inference rules rather than the specific form of input.

Template Substitution. The templates that produce the bulk of the low-level output are written in the well-known Jinja2 language. This language is commonly used for the server-side generation of web pages. KaSim or BNGL programs are not web pages but they are text documents and Jinja2 is well suited to generating them. It has a notion of inclusion and inheritance that is useful for handling the variations among the different kinds of parts, which typically differ in the rules for one or two of the interactions in which they participate with the others being identical. We provide a total of 15 templates for KaSim, of which there are top-level templates for each of the five distinct types of biological part defined in the gcc vocabulary as well as a generic part template, five templates implementing functionality shared among parts, and five consisting of supporting boilerplate required by KaSim.

A full description of the facilities provided by Jinja2 is beyond the scope of this paper, but a flavor is given in Figure 9, which shows an example of a template for a generic part (not having specific functionality like a promoter or operator might) demonstrating substitution of the name variable derived from annotation, and include statements referencing several other templates, one of which is reproduced and shows the KaSim code that is produced.

We use specific terms for defining the rates for the rules in which biological parts are involved, and a few other terms according to the function of the biological part of interest. It is possible to find the available terms out by inspecting the gcc vocabulary provided in the Supporting Information.

A fragment of the gcc vocabulary is reproduced in Figure 10. Though this exposes some implementation detail, it is useful to understand the relationships between the various terms used to describe models. This is also important when supplying customized templates.

There are gcc:Tokens, so named because they correspond to tokens in the low-level language that are replaced. Each must

DOI: 10.1021/acssynbio.8b00201
ACS Synth. Biol. 2018, 7, 2812−2823

have a preferred label that gives the literal token. In cases where there exists a sensible default value, this is given with gcc:default. The purpose of these statements is to act as a bridge between the fully materialized RDF representation of the model and the templates that require substitution of locally meaningful names.

For each kind of part (such as the gcc:Operator in the example in Figure 10), there are two main annotations that are necessary. For each machine-readable low-level language, a template is specified. The gcc:tokens annotations give the tokens that are pertinent to this kind of part. These must be specified in the high-level model or allowed to take on their default values. In addition to documenting the requirements of the templates for each kind of part, these statements are, "operationalized" and used by the compiler. They can equally well be used to check that a supplied high-level model is sufficiently complete and well-formed to produce an output program.

Derivation of Declarations. The KaSim language requires forward declaration of the type signatures of agents. This is by design[50] so that the simulator can check that agents are correctly used where they appear in patterns in the rules. While this design choice can help a modeler that is writing a simulation program in the low-level language by hand, to assist in finding mistakes and typographical errors, it is not possible to know *a priori* what these declarations should be in the present context. The correct declarations for DNA, RNA, and Protein depend on the complete set of parts that make up the model so their correct declarations cannot be in any template for an individual part.

To solve this issue, the compiler implements a postprocessing step. The rules that are produced by instantiating the templates for each part are concatenated together with any explicitly supplied rules and then the whole is parsed. The use of each agent in each rule in this rule-set is assumed to be correct by construction. From there a declaration that covers each use of each agent is built up.

Initialization. At this final stage of the compiler, all rules are present, both supplied by the user for the host environment and implied by the parts that form the genetic circuits, and all declarations are also present. What is missing is the statement that creates an initial copy of the DNA sequence itself, with each upstream−downstream bond present. This information is, of course, available in the definition of the circuit, and so an appropriate %init statement, creating a single instance of the DNA sequence with correct linkages between the agent-parts is produced and added to the output. The low-level program is finally complete and ready to be executed.

DISCUSSION

We have presented a language, the GCDL for describing genetic circuits and our compiler for generating simulation executables from it. We have made the case that the succinctness of the GCDL affords the user the benefit of describing the salient aspects of these circuits free of extraneous detail, that this reduces the potential for user error inherent in detailed coding of molecular interactions, and that this approach also affords flexibility in choosing the simulation or experimental methodology for the model. We have further developed the argument that modularity in modeling of genetic circuits has similar benefits of modularity in high-level programming languages, namely, encapsulation and clarity. We now consider some of the limitations and benefits of our design choices and explore some areas ripe for future research.

It is important to understand the correctness and verification properties of the compiler and the GCDL. The GCDL is an RDF-based language and models are typically written in Turtle. The syntax[11,38] on a concrete level is well-defined and models that are badly formed will be rejected. The standard templates are documented in machine-readable form in the GCDL vocabulary. Annotations that are required for a given part type also cause the model to be rejected by the compiler if they are not present. But the compiler does not perform verification in terms of how the parts are composed. Users are free to choose any DNA parts and in any order. For example, a model that includes a coding sequence part without preceding promoter and ribosome binding site parts is allowed, though and the resulting model would emit no protein agents and perhaps not be very interesting. Verifying whether a given circuit expressed in the GCDL is accepted by a parts grammar;[7,51] verification of part sequences is out of scope for the compiler but could be the subject of future work.

The expressive power afforded by the design choice of modularity—fixing the level of abstraction for a model—comes at a cost. Biological parts are considered as atomic units. While it is straightforward to model complex mechanisms like combinatorial logic operators and cooperative binding it is not straightforward to mix models in terms of the part abstraction with models of the underlying substrate. Phenomena that inherently involve the physical or chemical structure of the DNA molecule or the shape of a protein cannot be modeled directly, and we are restricted to simply asserting that they occur or not at some rate. Similarly, while parts that share nucleotide sequences and may overlap can be marked as such with the gcc:overlaps term, this has no effect on the modeling. If the fact of parts overlapping is significant in the behavior of the circuit, those parts are not modular and that would break the abstraction barrier. Such an annotation can, however, be used when selecting parts for assembly *in vitro*. Parts for which overlap is functionally significant can also be treated as an atomic unit with a suitable template. The modeling abstraction, once chosen, is fixed. This is by design, in order that models so expressed remain tractable.

Similar reasoning applies to optimization of DNA sequences. This is not our focus in the present work. Here, our main goal is to capture the dynamics of genetic circuit designs and to automate the process of model generation. Hence, deriving final DNA sequences encoding the behaviors captured in models is not our focus, and related research can indeed be incorporated in the future.[52] Because the language is based on RDF, custom user based data can be stored as annotations[39] to facilitate later optimization.

We do, however, envision optimization of circuits at the level of abstraction that we have chosen, and derivation of circuits to a given specification. A method for doing this, which we only sketch here, is to define a suitable fitness or distance measure on the output of simulations with respect to the desired specification. A starting candidate circuit is chosen, constructed from a given library of parts, and measured. Parts of the circuit are swapped, added or removed at random, subject to the constraint that the circuit remains well formed according to an operon grammar[7,51] and the new circuit is measured with respect to the specification. If the result is better than the previous circuit, the change is accepted, and the process is repeated until a locally optimal solution is found. This evolutionary algorithm approach is in contrast to the approach of assembling all possible circuits *in vitro* seen elsewhere[53−56] and is likely to be less efficient in cases where the desired behavior of the circuit can be measured simply, such as by

DOI: 10.1021/acssynbio.8b00201
ACS Synth. Biol. 2018, 7, 2812−2823

detecting the production of a fluorescent protein. However, for cases that may be more difficult to measure *in vitro* such as oscillations or more complex outputs it can be more straightforward to measure the output and compare to the specification when done *in silico*.

Currently, the templates that we have supplied only handle single stranded genetic constructs. Parts are composed using upstream and downstream bonds to create chains of DNA sequences, and our framework currently does not consider whether the other strand is free or not regarding the elongation RNAP or the binding of molecules and so on. One reason why we have chosen to support the single-stranded case first is simplicity. Another is that databases of models for double-stranded parts are not available. Adding support for this in templates, and developing a library of suitable parts is another topic for future research.

Here, we presented the application of rule-based models and Semantic Web technologies to automate the design of genetic circuits. Representations of cellular activities were captured using modular rules to support scalability of designs. The automation process is facilitated by the GCDL high level language, which is built upon the Semantic Web and is used to describe genetic circuits. Furthermore, we presented a compiler that generates rule-based executable models from the high-level description. The implementation of the compiler is notable in its use of semantic inference, and the language is sophisticated enough to support several classes of complex regulatory mechanisms. Despite the expressive power afforded by this approach, the language maintains a succinctness and simplicity that we hope will be a boon to those modeling genetic circuits in silico. The implicit modularity in our rule-based approach and the high-level language presented will be beneficial to synthetic biologists to model complex regulatory relationships through the use of widely adopted standards.

■ ASSOCIATED CONTENT

Supporting Information

The Supporting Information is available free of charge on the ACS Publications website at DOI: 10.1021/acssynbio.8b00201.

Extended output representation discussion; The genetic circuit compiler language; Additional inference rules for GCC; Complete model of the Elowitz repressilator (PDF)

■ AUTHOR INFORMATION

Corresponding Author

*E-mail: wwaites@inf.ed.ac.uk.

ORCID

William Waites: 0000-0002-7759-6805

Göksel Mısırlı: 0000-0002-2454-7188

Anil Wipat: 0000-0001-7310-4191

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

Thanks to Michael Korbakov for porting the original Haskell implementation of the agent declaration code to Python. We would also like to thank the anonymous reviewers, whose feedback has resulted in a much improved manuscript. W.W., M.C., G.M., A.W., and V.D. acknowledge the support from the Engineering and Physical Sciences Research Council (EPSRC) grant EP/J02175X/1 and from UK Research Councils' Synthetic Biology for Growth program. W.W. and V.D. acknowledge the European Union's Seventh Framework Programme for research, technological development and demonstration grant number 320823 (to V.D. and W.W.). W.W. also acknowledges support from the National Academies Keck Futures Initiative of the National Academy of Sciences award number NAKFI CB12.

■ REFERENCES

(1) Baldwin, G. (2012) *Synthetic Biology: A Primer*, John Wiley & Sons.

(2) Paddon, C. J., et al. (2013) High-level semi-synthetic production of the potent antimalarial artemisinin. *Nature (London, U. K.) 496*, 528−532.

(3) Galanie, S., Thodey, K., Trenchard, I. J., Filsinger Interrante, M., and Smolke, C. D. (2015) Complete biosynthesis of opioids in yeast. *Science (Washington, DC, U. S.) 349*, 1095−1100.

(4) Ferry, M. S., Hasty, J., and Cookson, N. A. (2012) Synthetic biology approaches to biofuel production. *Biofuels 3*, 9−12.

(5) Ruder, W. C., Lu, T., and Collins, J. J. (2011) Synthetic Biology Moving into the Clinic. *Science (Washington, DC, U. S.) 333*, 1248−1252.

(6) Elowitz, M. B., and Leibler, S. (2000) A synthetic oscillatory network of transcriptional regulators. *Nature (London, U. K.) 403*, 335−338.

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

(8) Beal, J., Phillips, A., Densmore, D., and Cai, Y. (2011) *Design and Analysis of Biomolecular Circuits: Engineering Approaches to Systems and Synthetic Biology* (Koeppl, H., Setti, G., di Bernardo, M., and Densmore, D., Eds.) pp 225−252, Springer, New York.

(9) Cai, Y., Beal, J., Phillips, A., and Densmore, D. (2011) *Design and Analysis of Biomolecular Circuits*, pp 225−252, SpringerLink.

(10) Hallinan, J. S., Gilfellon, O., Misirli, G., and Wipat, A. (2014) Tuning receiver characteristics in bacterial quorum communication: An evolutionary approach using standard virtual biological parts. In *2014 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology*, pp 1−8.

(11) Cyganiak, R., Carroll, J., and Lanthaler, M. (2014) *RDF 1.1 Concepts and Abstract Syntax*: W3C Recommendation, World Wide Web Consortium, https://www.w3.org/TR/rdf11-concepts/.

(12) Neal, M. L., Cooling, M. T., Smith, L. P., Thompson, C. T., Sauro, H. M., Carlson, B. E., Cook, D. L., and Gennari, J. H. (2014) A reappraisal of how to build modular, reusable models of biological systems. *PLoS Comput. Biol. 10*, e1003849.

(13) Danos, V., Feret, J., Fontana, W., Harmer, R., and Krivine, J. (2007) In *CONCUR 2007—Concurrency Theory* (Caires, L., and Vasconcelos, V. T., Eds.) pp 17−41, Lecture Notes in Computer Science 4703, Springer, Berlin, DOI: 10.1007/978-3-540-74407-8_3.

(14) Krivine, J., and Feret, J. (2017) *KaSim*, http://dev.executableknowledge.org/.

(15) Blinov, M. L., Faeder, J. R., Goldstein, B., and Hlavacek, W. S. (2004) BioNetGen: software for rule-based modeling of signal transduction based on the interactions of molecular domains. *Bioinformatics 20*, 3289−3291.

(16) Harris, L. A., Hogg, J. S., Tapia, J.-J., Sekar, J. A. P., Gupta, S., Korsunsky, I., Arora, A., Barua, D., Sheehan, R. P., and Faeder, J. R. (2016) BioNetGen 2.2: advances in rule-based modeling. *Bioinformatics 32*, 3366−3368.

(17) Galdzicki, M., Clancy, K. P., Oberortner, E., Pocock, M., Quinn, J. Y., Rodriguez, C. A., Roehner, N., Wilson, M. L., Adam, L., and Anderson, J. C. (2014) The Synthetic Biology Open Language (SBOL) provides a community standard for communicating designs in synthetic biology. *Nat. Biotechnol. 32*, 545.

(18) Berners-Lee, T. (2005) *An RDF Language for the Semantic Web*, Design Issues.

(19) Hlavacek, W. S., Faeder, J. R., Blinov, M. L., Perelson, A. S., and Goldstein, B. (2003) The complexity of complexes in signal transduction. *Biotechnol. Bioeng. 84*, 783−794.
(20) Danos, V., and Laneve, C. (2004) Formal molecular biology. *Theor. Comput. Sci. 325*, 69−110.
(21) Danos, V., Feret, J., Fontana, W., Harmer, R., and Krivine, J. (2008) *Formal Methods in Systems Biology*, pp 103−122, Springer.
(22) Misirli, G., Hallinan, J., and Wipat, A. (2014) Composable Modular Models for Synthetic Biology. *J. Emerg. Technol. Comput. Syst. 11*, 22:1−22:19.
(23) Roehner, N., Zhang, Z., Nguyen, T., and Myers, C. J. (2015) Generating Systems Biology Markup Language Models from the Synthetic Biology Open Language. *ACS Synth. Biol. 4*, 873−879. PMID: 25822671.
(24) Nguyen, T., Roehner, N., Zundel, Z., and Myers, C. J. (2016) A Converter from the Systems Biology Markup Language to the Synthetic Biology Open Language. *ACS Synth. Biol. 5*, 479−486. PMID: 26696234.
(25) Cooling, M. T., Rouilly, V., Misirli, G., Lawson, J., Yu, T., Hallinan, J., and Wipat, A. (2010) Standard virtual biological parts: a repository of modular modeling components for synthetic biology. *Bioinformatics 26*, 925−931.
(26) van Harmelen, F., and McGuinness, D. L. (2004) *OWL Web Ontology Language*, W3C Recommendation, World Wide Web Consortium.
(27) Masinter, L., Berners-Lee, T., and Fielding, R. T. (2005) *RFC3896 - Uniform Resource Identifier (URI): Generic Syntax*, Request for Comments, https://tools.ietf.org/html/rfc3986.
(28) Hyland, B., Atemezing, G., and Villazón-Terrazas, B. (2014) *Best Practices for Publishing Linked Data*, W3C Working Group Note, World Wide Web Consortium.
(29) Sauermann, L., Cyganiak, R., and Völkel, M. (2011) *Cool URIs for the Semantic Web*, W3C Interest Group Note, World Wide Web Consortium, https://www.w3.org/TR/cooluris/.
(30) Ashburner, M., et al. (2000) Gene Ontology: tool for the unification of biology. *Nat. Genet. 25*, 25−29.
(31) Consortium, U., et al. (2008) The universal protein resource (UniProt). *Nucleic Acids Res. 36*, D190−D195.
(32) Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y., and Hendler, J. (2008) N3logic: A logical framework for the world wide web. *Theor Pract. Log.Prog. 8*, 249−269.
(33) Berners-Lee, T., and Connolly, D. (2011) *Notation3 (N3): A readable RDF syntax*; W3C Team Submission, World Wide Web Consortium.
(34) Horrocks, I. (2005) *OWL: A Description Logic Based Ontology Language*, pp 1−4, Logic Programming.
(35) Brickley, D., and Guha, R. V. (2014) *RDF Schema 1.1*; W3C Recommendation, World Wide Web Consortium, https://www.w3.org/TR/rdf-schema/.
(36) Drummond, N., and Shearer, R. (2006) The Open World Assumption. eSI Workshop: The Closed World of Databases meets the Open World of the Semantic Web.
(37) Halpin, H., Hayes, P. J., McCusker, J. P., McGuinness, D. L., and Thompson, H. S. (2010) When owl:sameAs Isn't the Same: An Analysis of Identity in Linked Data. In *The Semantic Web—ISWC 2010*; pp 305−320, DOI: 10.1007/978-3-642-17746-0_20.
(38) Prud'hommeaux, E., and Carothers, G. (2014) *RDF 1.1 Turtle*; W3C Recommendation, World Wide Web Consortium, https://www.w3.org/TR/turtle/.
(39) Misirli, G., Cavaliere, M., Waites, W., Pocock, M., Madsen, C., Gilfellon, O., Honorato-Zimmer, R., Zuliani, P., Danos, V., and Wipat, A. (2016) Annotation of rule-based models with formal semantics to enable creation, analysis, reuse and visualization. *Bioinformatics 32*, btv660.
(40) Miles, A., Matthews, B., Wilson, M., and Brickley, D. (2005) SKOS Core: Simple knowledge organisation for the Web. In *International Conference on Dublin Core and Metadata Applications*, pp 3−10.
(41) Kunze, J. A., and Baker, T. (2007) *RFC5013 - The Dublin Core Metadata Element Set*; Request for Comments, https://tools.ietf.org/html/rfc5013.
(42) Forrest, P. (2016) In *The Stanford Encyclopedia of Philosophy* (Zalta, E. N., Ed.) Metaphysics Research Lab, Stanford University.
(43) Cox, R. S., Surette, M. G., and Elowitz, M. B. (2007) Programming gene expression with combinatorial promoters. *Mol. Syst. Biol. 3*, 145.
(44) Wang, B., Kitney, R. I., Joly, N., and Buck, M. (2011) Engineering modular and orthogonal genetic logic gates for robust digital-like synthetic biology. *Nat. Commun. 2*, 508.
(45) Sanchez, A., Garcia, H. G., Jones, D., Phillips, R., and Kondev, J. (2011) Effect of promoter architecture on the cell-to-cell variability in gene expression. *PLoS Comput. Biol. 7*, e1001100.
(46) Bongers, R. S., Veening, J.-W., Wieringen, M. V., Kuipers, O. P., and Kleerebezem, M. (2005) Development and Characterization of a Subtilin-Regulated Expression System in Bacillus subtilis: Strict Control of Gene Expression by Addition of Subtilin. *Appl. Environ. Microbiol. 71*, 8818−8824.
(47) Yu, K., Liu, C., Kim, B.-G., and Lee, D.-Y. (2015) Synthetic fusion protein design and applications. *Biotechnol. Adv. 33*, 155−164.
(48) Wilson-Kanamori, J., Danos, V., Thomson, T., and Honorato-Zimmer, R. (2015) In *Computational Methods in Synthetic Biology* (Marchisio, M. A., Ed.) pp 105−135, Methods in Molecular Biology 1244, Springer, New York, DOI: 10.1007/978-1-4939-1878-2_6.
(49) Ronacher, A. (2008) *Jinja2 (The Python Template Engine)*, http://jinja.pocoo.org.
(50) Feret, J., and Krivine, J. (2015) personal correspondence.
(51) Cai, Y., Hartnett, B., Gustafsson, C., and Peccoud, J. (2007) A syntactic model to design and verify synthetic genetic constructs derived from standard biological parts. *Bioinformatics 23*, 2760−2767.
(52) Misirli, G., Hallinan, J. S., Yu, T., Lawson, J. R., Wimalaratne, S. M., Cooling, M. T., and Wipat, A. (2011) Model annotation for synthetic biology: automating model to nucleotide sequence conversion. *Bioinformatics 27*, 973−979.
(53) Guet, C. C., Elowitz, M. B., Hsing, W., and Leibler, S. (2002) Combinatorial synthesis of genetic networks. *Science (Washington, DC, U. S.) 296*, 1466−1470.
(54) Menzella, H. G., Reid, R., Carney, J. R., Chandran, S. S., Reisinger, S. J., Patel, K. G., Hopwood, D. A., and Santi, D. V. (2005) Combinatorial polyketide biosynthesis by de novo design and rearrangement of modular polyketide synthase genes. *Nat. Biotechnol. 23*, 1171.
(55) Smanski, M. J., Bhatia, S., Zhao, D., Park, Y., Woodruff, L. B., Giannoukos, G., Ciulla, D., Busby, M., Calderon, J., and Nicol, R. (2014) Functional optimization of gene clusters by combinatorial design and assembly. *Nat. Biotechnol. 32*, 1241.
(56) Cress, B. F., Toparlak, O. D., Guleria, S., Lebovich, M., Stieglitz, J. T., Englaender, J. A., Jones, J. A., Linhardt, R. J., and Koffas, M. A. (2015) CRISPathBrick: modular combinatorial assembly of type II-A CRISPR arrays for dCas9-mediated multiplex transcriptional repression in E. coli. *ACS Synth. Biol. 4*, 987−1000.

DOI: 10.1021/acssynbio.8b00201
ACS Synth. Biol. 2018, 7, 2812−2823