



THE UNIVERSITY *of* EDINBURGH

Edinburgh Research Explorer

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

Citation for published version:

Waites, W, Misirli, G, Cavaliere, M, Danos, V & Wipat, A 2018, 'A Genetic Circuit Compiler: Generating Combinatorial Genetic Circuits with Web Semantics and Inference' *ACS Synthetic Biology*, vol. 7, no. 12, pp. 2812-2823. DOI: 10.1021/acssynbio.8b00201

Digital Object Identifier (DOI):

[10.1021/acssynbio.8b00201](https://doi.org/10.1021/acssynbio.8b00201)

Link:

[Link to publication record in Edinburgh Research Explorer](#)

Document Version:

Peer reviewed version

Published In:

ACS Synthetic Biology

General rights

Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy

The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.



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

William Waites, Goksel Misirli, Matteo Cavaliere, Vincent Danos, and Anil Wipat

ACS Synth. Biol., **Just Accepted Manuscript** • DOI: 10.1021/acssynbio.8b00201 • Publication Date (Web): 08 Nov 2018

Downloaded from <http://pubs.acs.org> on November 9, 2018

Just Accepted

“Just Accepted” manuscripts have been peer-reviewed and accepted for publication. They are posted online prior to technical editing, formatting for publication and author proofing. The American Chemical Society provides “Just Accepted” as a service to the research community to expedite the dissemination of scientific material as soon as possible after acceptance. “Just Accepted” manuscripts appear in full in PDF format accompanied by an HTML abstract. “Just Accepted” manuscripts have been fully peer reviewed, but should not be considered the official version of record. They are citable by the Digital Object Identifier (DOI®). “Just Accepted” is an optional service offered to authors. Therefore, the “Just Accepted” Web site may not include all articles that will be published in the journal. After a manuscript is technically edited and formatted, it will be removed from the “Just Accepted” Web site and published as an ASAP article. Note that technical editing may introduce minor changes to the manuscript text and/or graphics which could affect content, and all legal disclaimers and ethical guidelines that apply to the journal pertain. ACS cannot be held responsible for errors or consequences arising from the use of information contained in these “Just Accepted” manuscripts.

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*

[‡]*School of Computing and Mathematics, Keele University*

[¶]*School of Computing & Mathematics, Manchester Metropolitan University*

[§]*École Normale Supérieure, Paris, CNRS*

^{||}*School of Computing Science, Newcastle University*

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

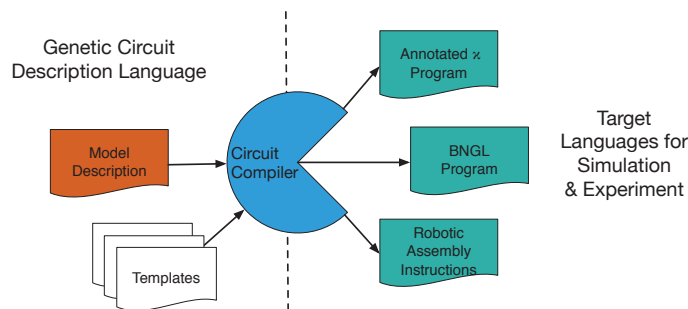
Semantic Web, Inference, Program Generation, Synthetic Biology, Genetic Circuits

Synthetic biology extends classical genetic engineering with concepts of modularity, standardisation, and abstraction drawn largely from computer engineering. The goal is ambitious: to design complex biological systems, perhaps entire genomes, from first principles¹. This enterprise has met with some success such as the microbial production of drug synthesis^{2,3}, new biofuels production⁴ and alternative approaches to disease treatment⁵. However, most applications are still small and mostly designed manually.

There 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 behaviour 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 modelling granularities. An overview of in-

1
2
3 formation flow through the compiler is shown in Figure 1. We demonstrate the utility
4 of this approach by describing, compiling and simulating a complete genetic circuit, the
5 well-known Elowitz repressilator⁶. The compiler and example code are available at <https://github.com/rulebased/composition>.
6
7
8
9



10
11
12
13
14
15
16
17
18
19
20
21
22 Figure 1: High-level data flow through the compiler. The compiler for synthetic gene circuits
23 takes a model description written in GCDL and, using language-appropriate appropriate
24 templates, creates code for simulation and laboratory assembly. We have implemented tem-
25 plates for annotated- κ for the KaSim software, and envision similar for the BNGL as well as
26 SBOL.
27
28
29

30 Code generation from this high-level description to a low-level language for simula-
31 tion greatly reduces the scope for error in coding simulations. Because the language is
32 implementation-independent it is not tightly coupled to any particular interpreter or hard-
33 ware. In this way GCDL facilitates *evergreen models*, models that are specified sufficiently
34 well to be unambiguous but not so specifically that they can only be executed or constructed
35 in one software package or environment.
36
37
38
39
40
41

42 Domain specific languages and examples of compilers processing these languages have
43 previously been shown^{7–10}. These languages are designed to allow for simulations using a
44 particular methodology such as solving systems of ordinary differential equations or using
45 Monte-Carlo simulation. Unlike previous approaches, we emphasise the use of abstraction to
46 facilitate *retargeting* or production of output suitable for different simulation environments
47 and techniques as well as automated circuit assembly in the laboratory from a single descrip-
48 tion. Compiler targets are implemented using conditional inference, defining the semantics
49 of the terms used in the description of the circuit in a way that is determined by the desired
50
51
52
53
54
55
56
57
58
59
60

output type. The design of the compiler is general, and not limited to the present context of genetic circuits. The design shown schematically in Figure 2.

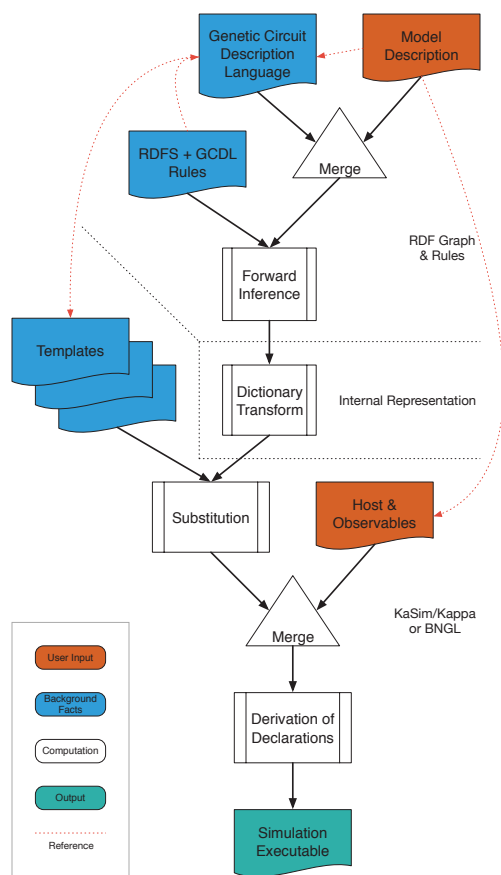


Figure 2: Detailed data flow through the compiler. This illustrates the use of inference to expand the GCDL model to derive consequent information appropriate to producing the next stage of output in the specific target language.

The GCDL is an RDF¹¹ vocabulary and attendant inference rules which facilitates gathering and collation of information about the constituent parts of a genetic circuit¹². The output programs can be specialised to various languages, such as the KaSim flavour of κ ^{13,14}, BioNetGen's BNGL^{15,16}, other representations such as SBOL¹⁷ 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 rules¹⁸ from the input model.

We now proceed as follows. We begin with an overview of those aspects of synthetic

1
2
3 biology and genetic engineering that are necessary to contextualise our work. Next, we
4 explain the representation of this kind of genetic circuit model in GCDL, this is the main
5 input to the compiler. In order to understand the desired output of the compiler, we
6 then illustrate how these constructs are represented as rule-based code for the κ language
7 simulator, KaSim. There follows a discussion of how the compiler infers the executable
8 model from the input description. Finally, we discuss some possible uses and limitations of
9 our technique.
10
11
12
13
14
15
16
17
18

19 Background

20 Rule-based Modelling of Genetic Processes

21
22
23 A weakness of reaction-based methods for modelling the processes of transcription, trans-
24 lation and the production of chains of proteins is that they require chemical species for
25 each bound state of the reagents. This in turn requires specification of reactions for each
26 combination of these reagents. To solve this problem of needing combinatorially many re-
27 actions to describe substantially the same process, a generalisation of reactions called *rules*
28 are used^{19–21}.
29
30
31
32
33
34
35
36
37

38 In the rule-based representation, *agents* correspond to reagents and they can have slots
39 or *sites* that can be bound, or not. They can also have internal state. Unlike reactions which
40 have no preconditions apart from the presence of the reagents, with rules, a configuration
41 of the sites — bound in a particular way, bound in some way, unbound, or unspecified —
42 is a precondition for the application of the rule. A rule may re-arrange the bonds, creating
43 or destroying them, without the need to invent new agents in order to represent different
44 configurations of a given set of molecules.
45
46
47
48
49
50

51 The reader should note that the word *rule* is used in two distinct senses in this article.
52 The first is as we have just described. The second is in the sense of *inference rule* as used in
53 logic and in particular the way in which we deduce executable rule-based models from their
54
55
56
57
58
59
60

1
2
3 declarative representations in RDF.
4
5
6

7 **The κ Language**

8
9

10 To briefly illustrate the essentials of rule-based modelling we will use the language of the
11 Kappa simulation software, KaSim¹⁴. An agent declaration and rule expressing the formation
12 of a polymer can be written as,
13
14
15

```
16  
17 %agent: A(d,u)  
18  
19  
20 'binding' A(u[.]), A(d[.]) -> A(u[1]), A(d[1]) @k  
21  
22
```

23 We can gloss this as an agent with two sites, u and d for upstream and downstream, and
24 a rule. The rule concerns two agent patterns one of which has an unbound upstream site,
25 and the other an unbound downstream site, and the action of the rule is to bind them, the
26 notation $[1]$ denoting the bond. This process happens at some rate κ .
27
28
29
30

31 The state of the other site of each agent is left unspecified, so implicit in this rule is the
32 possibility that either or both the agents may already be bound to others and so part of
33 arbitrarily long chains. In other words this expression covers not only two monomers joining
34 together but an n -mer and an m -mer for arbitrary n and m . This is the essence of the
35 expressive advantage that rule-based modelling provides. To express a similar concept using
36 a reaction network would in fact require infinitely many reagents for every possible n (and
37 m) and infinitely many reactions for every possible combination.
38
39
40
41
42
43
44
45
46

47 **Biological Parts and Annotation**

48

49 For efficiency, and economy of representation, we claim that the description of a compu-
50 tational model should include minimum information necessary for simulation. However, in
51 order to use these models in an automated design process, additional metadata, or *annota-*
52 *tions*, about the meaning of different modelling entities is needed¹². Annotation facilitates
53
54
55
56
57
58
59
60

1
2
3 the drawing specific parts from a database such as the Virtual Parts Repository²². Models in
4 that database are annotated with machine-readable metadata intended for combination into
5 larger models. Myers and his colleagues have used annotations to derive simulatable mod-
6 els from descriptions of genetic circuits²³ and vice versa²⁴, though these use reaction-based
7 techniques and so inherit the poor scaling properties of that method.
8
9
10
11
12

13 To facilitate the *in silico* evaluation of potential synthetic gene circuits, a library of
14 descriptions of genetic parts, together with their modular models is suggested in^{22,25}. These
15 parts are intended to be large enough to have a particular meaning or function (i.e. larger
16 than individual base pairs) but not so large that they lack the flexibility to be recombined
17 (i.e. entire genes). Thus we are concerned with coding sequences for particular proteins,
18 promoters that, when activated, start the transcription process, operators that activate or
19 suppress promoters according to whether they are bound or not by a given protein, and a
20 small number of other objects. A sequence of these objects is a genetic circuit, and our goal
21 is to have a good language for describing such sequences.
22
23
24
25
26
27
28
29
30

31 Annotation in this setting means machine-readable descriptions of entities of biologi-
32 cal interest. This is done with statements, triples of the form (subject, predicate, object)
33 according Semantic Web standards^{11,26}. Entities are identified with Universal Resource Iden-
34 tifiers (URIs)²⁷. This provides the dual benefit of globally unique identifiers for entities and
35 a built-in mechanism for retrieving more information about them providing that some care
36 is taken to publish data according to best practises^{28,29}. Large bodies of such information
37 about biologically relevant information are published on the Web^{30,31} and the use of Seman-
38 tic Web standards for annotating our models allows us to express how an entity in a model
39 description corresponds to a real world protein, or gene sequence or other entity.
40
41
42
43
44
45
46
47
48

49 The Semantic Web also affords us a technical advantage: inference rules. These can be
50 either explicit as in Notation3^{32,33} or implicit as in OWL Description Logics^{34,35}. In either
51 case this facility makes it possible, given a set of statements, to derive new statements ac-
52 cording to inference rules. We use this to improve the ergonomics of our high-level language:
53
54
55
56
57
58
59
60

1
2
3 while the compiler itself will make use, internally, of a large amount of information, we do not
4 expect the user to supply it in painstaking detail. Rather, we allow the user to specify the
5 minimum possible and provide rules to derive the necessary detail. Inference rules provide
6 for both economy of representation for the high-level model description and flexibility for
7 the different implementations.
8
9
10
11
12

13 14 15 **A Language for Synthetic Gene Circuits** 16 17

18 This section describes the GCDL, the high-level language for describing genetic circuits made
19 from standard biological parts^{25 22}. We begin by stating the properties that we want in such
20 a language and showing how we achieve them. There follows a synopsis of the vocabulary
21 terms essential to the language. Finally, we illustrate salient language features applied to
22 example circuits.
23
24
25
26
27
28
29

30 31 **Desired Language Features** 32

33 Our desired language features for high-level representation of a genetic circuit are as follows,
34
35

- 36 1. sufficiency, there should be enough information to derive executable code for the circuit,
37
- 38 2. identifiability, it should be possible to determine to which biological entities (DNA
39 sequences, proteins) the representation refers,
40
41
- 42 3. extensibility, it should be straightforward to add information or constructs that are
43 not presently foreseen,
44
45
- 46 4. generality, there should be no requirement that information about biological parts
47 comes from any particular set or source, and
48
49
- 50 5. concision, there should be a minimum of extraneous detail or syntax.
51
52
53
54
55
56
57
58
59
60

1
2
3 The third and fourth requirements are readily met by using RDF as the underlying data
4 model. The *open world* presumption³⁶ means that adding information as necessary is
5 straightforward. The use of URIs²⁷ which can be dereferenced to obtain the required in-
6 formation means that information from different web-accessible databases can be obtained,
7 mixed and matched as desired. The use of URIs goes some way towards meeting the second
8 requirement, albeit with some well-known caveats³⁷.
9

10
11 The first and last of the desired features are the primary areas of innovation of the
12 present work. We suggest (but do not require) the use of Turtle³⁸ or indeed Notation3¹⁸ as
13 the concrete surface syntax for writing models. This goes some way towards a representation
14 that is intelligible by humans. Even then, we aim to minimise what needs to be written and
15 we do this using inference rules — if a needed fact can be derived from the model under
16 the provided rule-set, it is unnecessary to write it explicitly in the model. Indeed it may
17 even be undesirable to do so since it is a possible source of errors, for example some kinds of
18 assertions may be correct in the context of some output types and incorrect in others. We
19 aim for a minimal, yet complete under the inference rules, description of the model.
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34

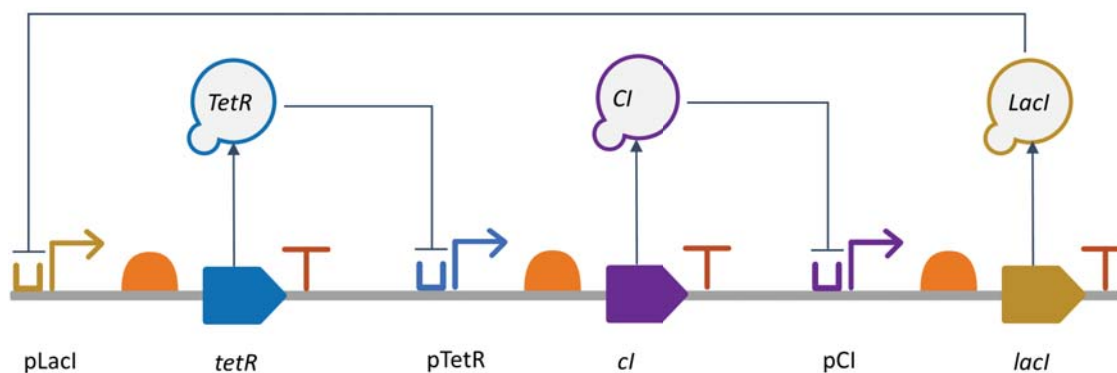
35 Vocabulary Terms

36
37 New terms introduced in this paper have the prefix `gcc` which can be read as the “Ge-
38 netic Circuit Compiler” vocabulary. The list of terms is reproduced in Table 3 and their
39 complete definitions are given together with the accompanying rules in the supplementary
40 materials. The GCDL is the union of terms from the `gcc` namespace with those from the
41 Rule-Based Model Ontology (RBMO) that we previously defined³⁹ together with terms from
42 the Simple Knowledge Organization System (SKOS)⁴⁰ vocabulary, RDF Schema (RDFS)³⁵
43 and Resource Description Framework (RDF)¹¹.
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

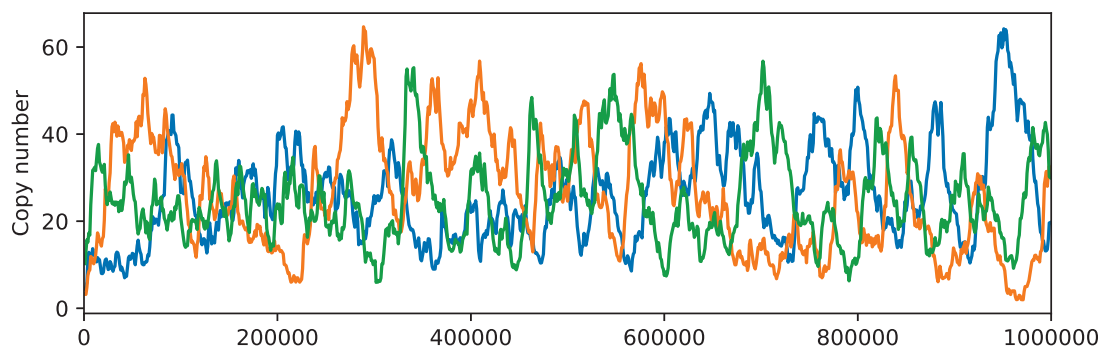
Table 3: Selected terms from the GCC vocabulary

Classes	
<code>gcc:Part</code>	Generic biological part
<code>gcc:Operator</code>	Operator
<code>gcc:Promoter</code>	Promoter
<code>gcc:RibosomeBindingSite</code>	Ribosome Binding Site
<code>gcc:CodingSequence</code>	Coding Sequence
<code>gcc:Terminator</code>	Terminator
<code>gcc:Token</code>	Token or symbol in a template

Predicates	
<code>gcc:include</code>	Include a low-level model fragment
<code>gcc:prefix</code>	The prefix to use for generated annotations
<code>gcc:init</code>	Specifies initial copy numbers
<code>gcc:part</code>	Links a part to its token or symbol
<code>gcc:overlaps</code>	Indicates that two parts overlap (symmetric)
<code>gcc:linear</code>	Linear circuit type
<code>gcc:circular</code>	Circular circuit type
<code>gcc:transcriptionFactor</code>	Relates an operator to its transcription factor
<code>gcc:transcriptionFactorBindingRate</code>	Various rates
<code>gcc:transcriptionFactorUnbindingRate</code>	
<code>gcc:rnapBindingRate</code>	
<code>gcc:rnapUnbindingRate</code>	
<code>gcc:rnapRNAUnbindingRate</code>	
<code>gcc:ribosomeBindingRate</code>	
<code>gcc:ribosomeRNAUnbindingRate</code>	
<code>gcc:ribosomeProteinUnbindingRate</code>	
<code>gcc:transcriptionInitiationRate</code>	
<code>gcc:transcriptionElongationRate</code>	
<code>gcc:translationElongationRate</code>	
<code>gcc:rnaDegradationRate</code>	
<code>gcc:proteinDegradationRate</code>	



(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 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 supplementary materials 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 Core⁴¹ 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

```

1  ## Model declaration
2  :m a rbmo:Model;
3
4  ## bibliographic metadata
5  dct:title "The Elowitz repressilator constructed from BioBrick parts";
6  dct:description "Representation of the Elowitz repressilator given in the Kappa BioBricks Framework
7  book chapter";
8  rdfs:seeAlso <http://link.springer.com/protocol/10.1007/978-1-4939-1878-2_6>;
9  gcc:prefix <http://id.inf.ed.ac.uk/rbm/examples/repressilator#>;
10 ## include the host environment
11 gcc:include <.../host.ka>;
12 ## initialisations
13 gcc:init
14   [ rbmo:agent :RNAp; gcc:value 700 ],
15   [ rbmo:agent :Ribosome; gcc:value 1000 ];
16 ## The circuit itself, a list of parts
17 gcc:linear (
18   :R0040o :R0040p :B0034a :C0051 :B0011a
19   :R0051o :R0051p :B0034b :C0012 :B0011b
20   :R0010o :R0010p :B0034c :C0040 :B0011c
21 ).

```

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

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 specialised task when programming a computer in a high-level language like C.

There follows initialisation 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 modelling for greater generality than reaction-based methods. Finally, the circuit itself is specified. The argument, or object is an `rdf:List` which 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 behaviour is. To obtain a working model, we need more.

```
1
2
3
4 :C0012 a gcc:CodingSequence;
5   gcc:label "Coding sequence for LacI";
6   gcc:part "C0012";
7   gcc:protein :P0010;
8   gcc:proteinDegradationRate 0.0001.
9
10 :P0010 a gcc:Protein;
11   bqbiol:is uniprot:P03023;
12   skos:prefLabel "P0010";
13   rdfs:label "LacI".
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
```

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.

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:proteinDegradationRate`. It is equally possible to specify the rates for transcription and translation in a similar manner though not shown here. In 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 annotation³⁹, 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' Law⁴²) of the latter can yield unwanted inferences when terms are not used perfectly rigorously³⁷. This weaker identify assertion permits the identification of the `:P0010` in the example with the identifier for the protein in the well-known UniProt³¹ database.

A More Complex Part Description

A more involved example demonstrating how an operator-promoter combination 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 neighbouring 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.

```
: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.

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

1
2
3 list of parts, upstream or downstream on the DNA strand may be specified along with their
4 states. This enables a promoter to be controlled by two or more operators. The rate itself
5 in this case is given with `gcc:value` for each case.
6
7
8
9

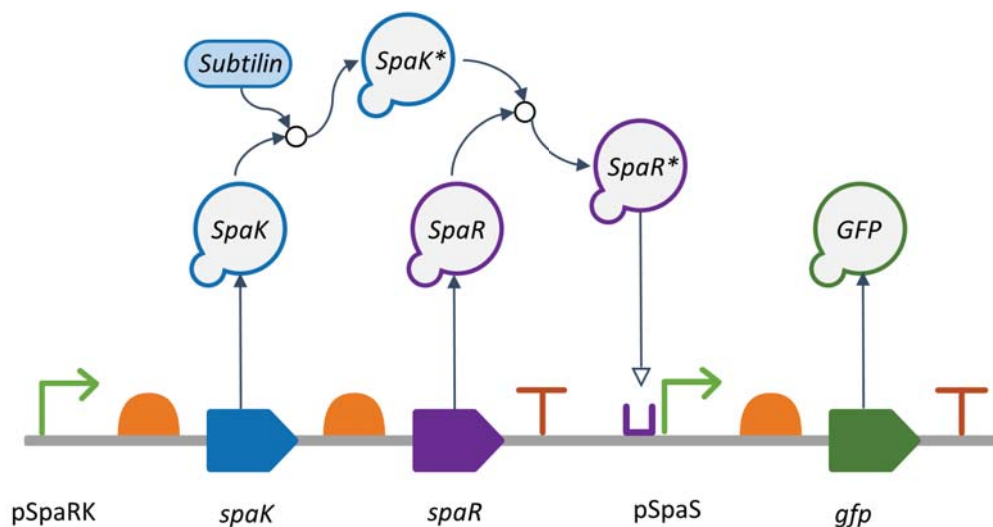
10 **Host and Protein-Protein Interactions**

11
12
13 The language can also support protein–protein interactions in a basic way. To see why these
14 are useful, consider an example from the engineering of a bacterial communication system
15 where the subtilin molecule is used to control population level dynamics. Cells have the
16 receiver device ^{22,46} to sense the existence of subtilin, and the reporter device to initiate
17 downstream cellular processes (Figures 7a and 7b). In the subtilin receiver, the interactions
18 among the proteins produced by translation and the operator-promoters are mediated by
19 a cascade reaction initiated by the subtilin molecule. Subtilin combines to phosphorylate
20 the *SpaK* protein, which in turn phosphorylates the *SpaR* protein that finally binds to the
21 promoter that controls the emission of a fluorescent green protein.
22
23
24
25
26
27
28
29
30

31 While the genetic circuit can straightforwardly be described similarly to the previous
32 repressilator example, the protein–protein interactions cannot. We do not attempt here to
33 model these interactions in the GCDL though a future extension could do so. Instead we
34 simply allow for inclusion of the relevant program, as a file in the output language (in this
35 case κ -language). It is possible to supply arbitrary code in the low-level language using the
36 `gcc:include` term. This facility makes it feasible to represent such genetic circuits which
37 depend strongly on the host environment in order to operate.
38
39
40
41
42
43
44
45
46

47 **Protein Fusion**

48
49
50 It is also worth noting that this example illustrates that in the high-level language it is
51 immediately possible to represent devices that produce chains of proteins. This is known as
52 protein fusion and is interesting for some applications⁴⁷. A chain of proteins is produced by
53 adding adjacent (and appropriate) coding sequences. It is enough to simply list the coding
54
55
56
57
58
59
60



(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.

```

: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
).

```

(b) Corresponding semantic model.

Figure 7: Representations of the Subtilin Receiver model.

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 specialised terms according to the function of the specific part. It is possible to find the available terms out by inspecting the gcc vocabulary included in the supplementary materials.

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 simulator¹⁴ for concrete illustration as it is widely adopted for stochastic simulation of rule-based models⁴⁸. The rule-based modelling 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 modelling methodology and corresponding output can be found in the supplementary materials.

The real work of modelling 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,

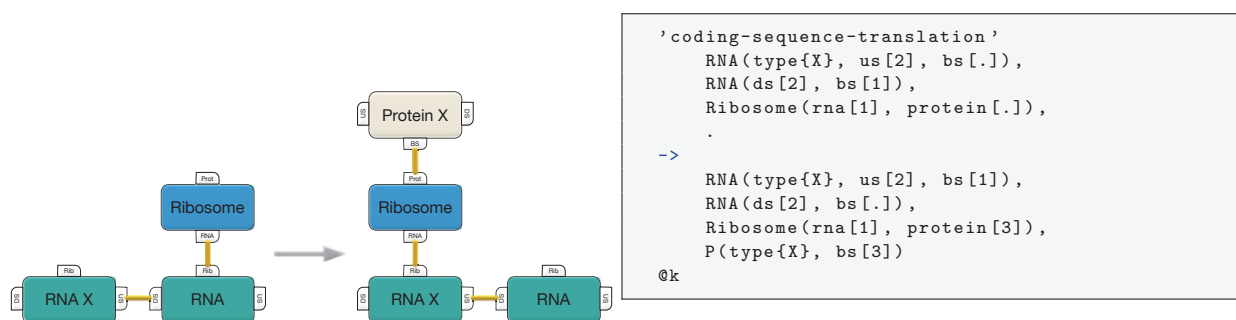


Figure 8: Translation of the RNA segment corresponding to a coding sequence to produce a protein.

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 .

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 materialised facts are transformed into a suitable internal representation. Substitution into templates is done next, and finally the result is post-processed 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 materialised 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,

```
: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 an `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

1
2
3 declarative description.
4

5 The above rule, and others like it serve to elaborate the high-level description into a more
6 detailed version suitable for the next stage of the compiler and relieve the user of the need
7 to supply the extra details. All implications that can be drawn under the `rdfs` inference
8 rules and the `gcc` specific rules are drawn and become part of the in the in-memory RDF
9 storage as the transitive closure of the rules (given the background facts and the provided
10 model facts).
11
12
13
14
15
16
17

18 **Internal Representation**

19 The output of the first stage of the compiler contains all the information necessary to com-
20 pletely describe the output, but it is not in a convenient form for providing to the template
21 rendering engine. Our implementation choice for the compiler is the Jinja2⁴⁹ rendering en-
22 gine. This means that the appropriate data-structure is a dictionary or associative list that
23 can be processed natively by these tools without need of external library. The required
24 internal representation is built up by querying the in-memory RDF storage for the specific
25 information required by the templates.
26
27
28
29
30
31
32
33
34

35 Our implementation does not require modification when new terms are added to the
36 vocabulary and templates. To add support for a new kind of part it is necessary to write
37 a new template for it and possibly add some terms to the vocabulary but does not require
38 changing the compiler software itself. What makes this possible are the inference rules
39 described in the previous section. The queries on the RDF storage that produce the internal
40 representation are posed in terms of the *consequents* of the inference rules rather than the
41 specific form of input.
42
43
44
45
46
47
48
49
50

51 **Template Substitution**

52 The templates that produce the bulk of the low-level output are written in the well-known
53 Jinja2 language. This language is commonly used for the server-side generation of web pages.
54
55
56
57
58
59
60

1
2
3 KaSim or BNGL programs are not web pages but they are text documents and Jinja2 is
4 well suited to generating them. It has a notion of inclusion and inheritance that is useful for
5 handling the variations among the different kinds of parts, which typically differ in the rules
6 for one or two of the interactions in which they participate with the others being identical.
7 We provide a total of 15 templates for KaSim, of which there are top-level templates for
8 each of the five distinct types of biological part defined in the `gcc` vocabulary as well as a
9 generic part template, five templates implementing functionality shared among parts, and
10 five consisting of supporting boilerplate required by KaSim.

11
12 A full description of the facilities provided by Jinja2 is beyond the scope of this paper, but
13 a flavour is given in Figure 9 which shows an example of a template for a generic part (not
14 having specific functionality like a promoter or operator might) demonstrating substitution
15 of the `name` variable derived from annotation, and include statements referencing several other
16 templates, one of which is reproduced and shows the KaSim code that is produced.

17
18 We use specific terms for defining the rates for the rules in which biological parts are
19 involved, and a few other terms according to the function of the biological part of interest.
20 It is possible to find the available terms out by inspecting the `gcc` vocabulary provided in
21 the supplementary materials..

22
23 A fragment of the `gcc` vocabulary is reproduced in Figure 10. Though this exposes some
24 implementation detail, it is useful to understand the relationships between the various terms
25 used to describe models. This is also important when supplying customised templates.

26
27 There are `gcc:Tokens`, so named because they correspond to tokens in the low-level
28 language that are replaced. Each must have a preferred label that gives the literal token. In
29 cases where there exists a sensible default value, this is given with `gcc:default`. The purpose
30 of these statements is to act as a bridge between the fully materialised RDF representation
31 of the model and the templates that require substitution of locally meaningful names.

32
33 For each kind of part (such as the `gcc:Operator` in the example in Figure 10), there
34 are two main annotations that are necessary. For each machine-readable low-level language,
35
36

```

1
2
3
4 ## Auto-generated generic part {{ name }}
5 {% include "header.ka" %}
6 {% import "context.ka" as context with context %}
7 {% import "meta.ka" as meta with context %}
8
9 {% include "transcription_elongation.ka" %}
10 {% include "transcription_termination.ka" %}
11 {% include "translation_chain.ka" %}
12 {% include "translation_elongation.ka" %}
13 {% include "translation_termination.ka" %}
14 {% include "host_maintenance.ka" %}

```

```

15 {% set rule = "%s-translation-chain" % name %}
16 //
17 //^ :{{ rule }} a rbmo:Rule;
18 //^  bqiol:isVersionOf go:GO:0006415;
19 {{ meta.rule() }}{# #}
20 //^  rdfs:label "{{ name }} formation of \
21 //^          translational chains, due to \
22 //^          gene fusion or leakiness of \
23 //^          stop codons".
24 // {{ name }} formation of translational chains,
25 // due to gene fusion or leakiness of stop codons
26 //
27 '>{{ rule }}' \
28   RNA(ds[2], bs[1]), \
29   Ribosome(rna[1], protein[3]), \
30   RNA(type{{ curly(name) }}, us[2], bs[.]), \
31   Protein(ds[.], bs[3]), . \
32 -> \
33   RNA(ds[2], bs[.]), \
34   Ribosome(rna[1], protein[3]), \
35   RNA(type~{{ name }}, us[2], bs[1]), \
36   P(ds[4], bs[.]), \
37   P(type{{ curly(name) }}, us[4], bs[3]) \
38   @{{ 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.

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, “operationalised” 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.


```
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: The specification in the `gcc` vocabulary of an `gcc:Operator` and associated terms.

Derivation of Declarations

The KaSim language requires forward declaration of the type signatures of agents. This is by design⁵⁰ 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 modeller 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 post-processing 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.

Initialisation

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

1
2
3 are also present. What is missing is the statement that creates an initial copy of the DNA
4 sequence itself, which each upstream–downstream bond present. This information is, of
5 course, available in the definition of the circuit, and so an appropriate `%init` statement,
6 creating a single instance of the DNA sequence with correct linkages between the agent-
7 parts is produced and added to the output. The low-level program is finally complete and
8 ready to be executed.
9
10
11
12
13
14
15
16

17 Discussion

18
19
20 We have presented a language, the GCDL for describing genetic circuits and our compiler
21 for generating simulation executables from it. We have made the case that the succinctness
22 of the GCDL affords the user the benefit of describing the salient aspects of these circuits
23 free of extraneous detail, that this reduces the potential for user error inherent in detailed
24 coding of molecular interactions, and that this approach also affords flexibility in choosing
25 the simulation or experimental methodology for the model. We have further developed the
26 argument that modularity in modelling of genetic circuits has similar benefits of modularity
27 in high-level programming languages, namely encapsulation and clarity. We now consider
28 some of the limitations and benefits of our design choices and explore some areas ripe for
29 future research.
30
31
32
33
34
35
36
37
38
39

40 It is important to understand the correctness and verification properties of the compiler
41 and the GCDL. The GCDL is an RDF-based language and models are typically written in
42 Turtle. The syntax^{11 38} on a concrete level is well defined and models that are badly formed
43 will be rejected. The standard templates are documented in machine-readable form in the
44 GCDL vocabulary. Annotations that are required for a given part type also cause the model
45 to be rejected by the compiler if they are not present. But the compiler does not perform
46 verification in terms of how the parts are composed. Users are free to choose any DNA
47 parts and in any order. For example, a model that includes a coding sequence part without
48
49
50
51
52
53
54
55
56
57
58
59
60

1
2
3 preceding promoter and ribosome binding site parts is allowed, though and the resulting
4 model would emit no protein agents and perhaps not be very interesting. Verifying whether
5 a given circuit expressed in the GCDL is accepted by a parts grammar^{7,51} verification of part
6 sequences is out of scope for the compiler but could be the subject of future work.
7
8
9

10
11 The expressive power afforded by the design choice of modularity — fixing the level
12 of abstraction for a model — comes at a cost. Biological parts are considered as atomic
13 units. While it is straightforward to model complex mechanisms like combinatorial logic
14 operators and cooperative binding it is not straightforward to mix models in terms of the
15 part abstraction with models of the underlying substrate. Phenomena that inherently involve
16 the physical or chemical structure of the DNA molecule or the shape of a protein cannot be
17 modelled directly and we are restricted to simply asserting that they occur or not at some
18 rate. Similarly, while parts which share nucleotide sequences and may overlap can be marked
19 as such with the `gcc:overlaps` term, this has no effect on the modelling. If the fact of parts
20 overlapping is significant in the behaviour of the circuit, those parts are not modular and
21 that would break the abstraction barrier. Such an annotation can, however, be used when
22 selecting parts for assembly *in vitro*. Parts for which overlap is functionally significant can
23 also be treated as an atomic unit with a suitable template. The modelling abstraction, once
24 chosen, is fixed. This is by design, in order that models so expressed remain tractable.
25
26
27
28
29
30
31
32
33
34
35
36
37
38

39 Similar reasoning applies to optimisation of DNA sequences. This is not our focus in the
40 present work. Here, our main goal is to capture the dynamics of genetic circuit designs and
41 to automate the process of model generation. Hence, deriving final DNA sequences encoding
42 the behaviours captured in models is not our focus, and related research can indeed be
43 incorporated in the future⁵². Because the language is based on RDF, custom user based
44 data can be stored as annotations³⁹ to facilitate later optimisation.
45
46
47
48
49
50

51 We do, however, envision optimisation of circuits at the level of abstraction that we have
52 chosen, and derivation of circuits to a given specification. A method for doing this, which
53 we only sketch here, is to define a suitable fitness or distance measure on the output of
54
55
56
57
58
59
60

1
2
3 simulations with respect to the desired specification. A starting candidate circuit is chosen,
4 constructed from a given library of parts, and measured. Parts of the circuit are swapped,
5 added or removed at random, subject to the constraint that the circuit remains well formed
6 according to an operon grammar^{7,51} and the new circuit is measured with respect to the
7 specification. If the result is better than the previous circuit, the change is accepted, and
8 the process is repeated until a locally optimal solution is found. This evolutionary algorithm
9 approach is in contrast to the approach of assembling all possible circuits *in vitro* seen
10 elsewhere^{53–56} and is likely to be less efficient in cases where the desired behaviour of the
11 circuit can be measured simply, such as by detecting the production of a fluorescent protein.
12 However for cases that may be more difficult to measure *in vitro* such as oscillations or more
13 complex outputs it can be more straightforward to measure the output and compare to the
14 specification when done *in silico*.

15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
Currently, the templates that we have supplied only handle single stranded genetic con-
structs. 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

1
2
3 this approach, the language maintains a succinctness and simplicity that we hope will be a
4 boon to those modelling genetic circuits in silico. The implicit modularity in our rule-based
5 approach and the high-level language presented will be beneficial to synthetic biologists to
6 model complex regulatory relationships through the use of widely adopted standards.
7
8
9

10
11
12 **Acknowledgements** Thanks to Michael Korbakov for porting the original Haskell im-
13 plementation of the agent declaration code to Python. We would also like to thank the
14 anonymous reviewers, whose feedback has resulted in a much improved manuscript.
15
16
17
18

19
20 **Funding** W.W., M.C., G.M., A.W., and V.D. acknowledge the support from the Engineer-
21 ing and Physical Sciences Research Council (EPSRC) grant EP/J02175X/1 and from UK
22 Research Councils' Synthetic Biology for Growth programme. W.W. and V.D. acknowledge
23 the European Union's Seventh Framework Programme for research, technological develop-
24 ment and demonstration grant number 320823 (to V.D. and W.W.). W.W. also acknowledges
25 support from the National Academies Keck Futures Initiative of the National Academy of
26 Sciences award number NAKFI CB12.
27
28
29
30
31
32
33

34
35 **Conflicts of Interest** The authors have no competing interests.
36
37
38

39 Resources

40
41
42
43 Rule-Based Modelling Ontology <http://purl.org/rbm/rbmo>
44 Genetic Circuit Description Language <http://purl.org/rbm/comp>
45 Rule-Based Modelling Examples <https://github.com/rulebased/rbmo>
46 Genetic Circuit Compiler Software <https://github.com/rulebased/composition>
47 Kappa BioBricks Framework Software <https://github.com/sstucki/lms-kappa>
48
49
50
51
52
53
54
55
56
57
58
59
60

References

1. Baldwin, G. *Synthetic biology: A primer*; John Wiley & Sons, 2012.
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. In *Design and Analysis of Biomolecular Circuits: Engineering Approaches to Systems and Synthetic Biology*; Koepl, H., Setti, G., di Bernardo, M., and Densmore, D., Eds.; Springer New York: New York, NY, 2011; pp 225–252.
9. Cai, Y., Beal, J., Phillips, A., and Densmore, D. *Design and Analysis of Biomolecular Circuits*; SpringerLink, 2011; pp 225–252.
10. Hallinan, J. S., Gilfellon, O., Misirli, G., and Wipat, A. Tuning receiver characteristics in bacterial quorum communication: An evolutionary approach using standard virtual

- 1
2
3 biological parts. 2014 IEEE Conference on Computational Intelligence in Bioinformatics
4 and Computational Biology. 2014; pp 1–8.
5
6
7
- 8 11. Cyganiak, R., Carroll, J., and Lanthaler, M. *RDF 1.1 Concepts and Abstract Syntax*;
9 W3C Recommendation, 2014.
10
11
12
- 13 12. Neal, M. L., Cooling, M. T., Smith, L. P., Thompson, C. T., Sauro, H. M., Carlson, B. E.,
14 Cook, D. L., and Gennari, J. H. (2014) A reappraisal of how to build modular, reusable
15 models of biological systems. *PLoS Comput. Biol.* *10*, e1003849.
16
17
18
- 19 13. Danos, V., Feret, J., Fontana, W., Harmer, R., and Krivine, J. In *CONCUR 2007 –*
20 *Concurrency Theory*; Caires, L., and Vasconcelos, V. T., Eds.; Lecture Notes in Com-
21 puter Science 4703; Springer Berlin Heidelberg, 2007; pp 17–41, DOI: 10.1007/978-3-
22 540-74407-8_3.
23
24
25
26
- 27 14. Krivine, J., and Feret, J. KaSim. 2017; <http://dev.executableknowledge.org/>.
28
29
30
- 31 15. Blinov, M. L., Faeder, J. R., Goldstein, B., and Hlavacek, W. S. (2004) BioNetGen:
32 software for rule-based modeling of signal transduction based on the interactions of
33 molecular domains. *Bioinformatics* *20*, 3289–3291.
34
35
36
37
- 38 16. Harris, L. A., Hogg, J. S., Tapia, J.-J., Sekar, J. A. P., Gupta, S., Korsunsky, I., Arora, A.,
39 Barua, D., Sheehan, R. P., and Faeder, J. R. (2016) BioNetGen 2.2: advances in rule-
40 based modeling. *Bioinformatics* *32*, 3366–3368.
41
42
43
44
- 45 17. Galdzicki, M., Clancy, K. P., Oberortner, E., Pocock, M., Quinn, J. Y., Rodriguez, C. A.,
46 Roehner, N., Wilson, M. L., Adam, L., and Anderson, J. C. (2014) The Synthetic Biology
47 Open Language (SBOL) provides a community standard for communicating designs in
48 synthetic biology. *Nat. Biotechnol.* *32*, 545.
49
50
51
52
- 53 18. Berners-Lee, T. *An RDF language for the Semantic Web*; Design Issues, 2005.
54
55
56
57
58
59
60

- 1
2
3 19. Hlavacek, W. S., Faeder, J. R., Blinov, M. L., Perelson, A. S., and Goldstein, B. (2003)
4 The complexity of complexes in signal transduction. *Biotechnol. Bioeng.* *84*, 783–794.
5
6
7
- 8 20. Danos, V., and Laneve, C. (2004) Formal molecular biology. *Theor. Comput. Sci.* *325*,
9 69–110.
10
11
- 12 21. Danos, V., Feret, J., Fontana, W., Harmer, R., and Krivine, J. *Formal methods in systems*
13 *biology*; Springer, 2008; pp 103–122.
14
15
16
- 17 22. Misirli, G., Hallinan, J., and Wipat, A. (2014) Composable Modular Models for Synthetic
18 Biology. *J. Emerg. Technol. Comput. Syst.* *11*, 22:1–22:19.
19
20
- 21 23. Roehner, N., Zhang, Z., Nguyen, T., and Myers, C. J. (2015) Generating Systems Biology
22 Markup Language Models from the Synthetic Biology Open Language. *ACS Synth. Biol.*
23 873–879, PMID: 25822671.
24
25
26
27
- 28 24. Nguyen, T., Roehner, N., Zundel, Z., and Myers, C. J. (2016) A Converter from the
29 Systems Biology Markup Language to the Synthetic Biology Open Language. *ACS Synth.*
30 *Biol.* *5*, 479–486, PMID: 26696234.
31
32
33
34
35
- 36 25. Cooling, M. T., Rouilly, V., Misirli, G., Lawson, J., Yu, T., Hallinan, J., and Wipat, A.
37 (2010) Standard virtual biological parts: a repository of modular modeling components
38 for synthetic biology. *Bioinformatics* *26*, 925–931.
39
40
41
42
- 43 26. van Harmelen, F., and McGuinness, D. L. *OWL Web Ontology Language*; W3C Recom-
44 mendation, 2004.
45
46
- 47 27. Masinter, L., Berners-Lee, T., and Fielding, R. T. *RFC3896 – Uniform Resource Iden-*
48 *tifier (URI): Generic Syntax*; Request for Comments, 2005.
49
50
51
- 52 28. Hyland, B., Ateazing, G., and Villazón-Terrazas, B. *Best Practices for Publishing*
53 *Linked Data*; W3C Working Group Note, 2014.
54
55
56
57

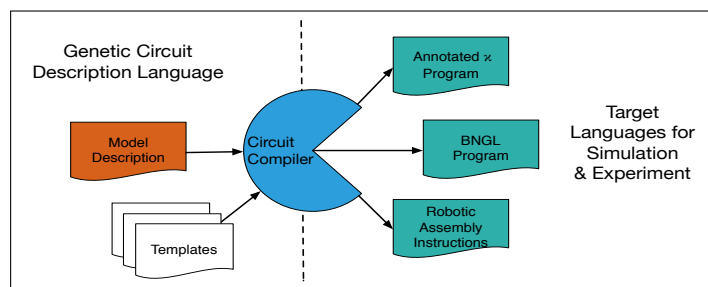
- 1
- 2
- 3
- 4 29. Sauermann, L., Cyganiak, R., and Völkel, M. *Cool URIs for the semantic web*; 2011.
- 5
- 6 30. Ashburner, M. et al. (2000) Gene Ontology: tool for the unification of biology. *Nat.*
- 7 *Genet.* 25, 25–29.
- 8
- 9
- 10 31. Consortium, U., and others, (2008) The universal protein resource (UniProt). *Nucleic*
- 11 *Acids Res.* 36, D190–D195.
- 12
- 13
- 14 32. Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y., and Hendler, J. (2008) N3logic: A
- 15 logical framework for the world wide web. *Theor. Pract. Log. Prog.* 8, 249–269.
- 16
- 17 33. Berners-Lee, T., and Connolly, D. *Notation3 (N3): A readable RDF syntax*; W3C Team
- 18 Submission, 2011.
- 19
- 20 34. Horrocks, I. OWL: A Description Logic Based Ontology Language. *Logic Programming.*
- 21 2005; pp 1–4.
- 22
- 23 35. Brickley, D., and V., G. R. *RDF Schema 1.1*; W3C Recommendation, 2014.
- 24
- 25 36. Drummond, N., and Shearer, R. The open world assumption. eSI Workshop: The Closed
- 26 World of Databases meets the Open World of the Semantic Web. 2006.
- 27
- 28 37. Halpin, H., Hayes, P. J., McCusker, J. P., McGuinness, D. L., and Thompson, H. S. When
- 29 owl:sameAs Isn't the Same: An Analysis of Identity in Linked Data. *The Semantic Web*
- 30 – ISWC 2010. 2010; pp 305–320, DOI: 10.1007/978-3-642-17746-0_20.
- 31
- 32 38. Prud'hommeaux, E., and Carothers, G. *RDF 1.1 Turtle*; W3C Recommendation, 2014.
- 33
- 34 39. Misirli, G., Cavaliere, M., Waites, W., Pocock, M., Madsen, C., Gilfellow, O., Honorato-
- 35 Zimmer, R., Zuliani, P., Danos, V., and Wipat, A. (2015) Annotation of rule-based
- 36 models with formal semantics to enable creation, analysis, reuse and visualization. *Bioin-*
- 37 *formatics* btv660.
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60

- 1
2
3
4 40. Miles, A., Matthews, B., Wilson, M., and Brickley, D. (2005) SKOS Core: Simple knowl-
5 edge organisation for the Web. *International Conference on Dublin Core and Metadata*
6 *Applications 0*, 3–10.
7
8
9
10 41. Kunze, J. A., and Baker, T. *RFC5013 – The Dublin Core Metadata Element Set*; Request
11 for Comments, 2007.
12
13
14
15 42. Forrest, P. In *The Stanford Encyclopedia of Philosophy*, winter 2016 ed.; Zalta, E. N.,
16 Ed.; Metaphysics Research Lab, Stanford University, 2016.
17
18
19
20 43. Cox, R. S., Surette, M. G., and Elowitz, M. B. (2007) Programming gene expression
21 with combinatorial promoters. *Mol. Syst. Biol.* *3*, 145.
22
23
24
25 44. Wang, B., Kitney, R. I., Joly, N., and Buck, M. (2011) Engineering modular and or-
26 thogonal genetic logic gates for robust digital-like synthetic biology. *Nat. Commun.* *2*,
27 508.
28
29
30
31 45. Sanchez, A., Garcia, H. G., Jones, D., Phillips, R., and Kondev, J. (2011) Effect of
32 promoter architecture on the cell-to-cell variability in gene expression. *PLoS Comput.*
33 *Biol.* *7*, e1001100.
34
35
36
37
38 46. Bongers, R. S., Veening, J.-W., Wieringen, M. V., Kuipers, O. P., and Kleerebezem, M.
39 (2005) Development and Characterization of a Subtilin-Regulated Expression System
40 in *Bacillus subtilis*: Strict Control of Gene Expression by Addition of Subtilin. *Appl.*
41 *Environ. Microbiol.* *71*, 8818–8824.
42
43
44
45
46
47 47. Yu, K., Liu, C., Kim, B.-G., and Lee, D.-Y. (2015) Synthetic fusion protein design and
48 applications. *Biotechnol. Adv.* *33*, 155–164.
49
50
51
52 48. Wilson-Kanamori, J., Danos, V., Thomson, T., and Honorato-Zimmer, R. In *Compu-*
53 *tational Methods in Synthetic Biology*; Marchisio, M. A., Ed.; Methods in Molecular
54 Biology 1244; Springer New York, 2015; pp 105–135, DOI: 10.1007/978-1-4939-1878-2.6.
55
56
57
58
59
60

- 1
2
3 49. Ronacher, A. Jinja2 (The Python Template Engine). 2008; <http://jinja.pocoo.org>.
4
5
6 50. Feret, J., and Krivine, J. Personal correspondence. 2015.
7
8
9 51. Cai, Y., Hartnett, B., Gustafsson, C., and Peccoud, J. (2007) A syntactic model to
10 design and verify synthetic genetic constructs derived from standard biological parts.
11 *Bioinformatics* 23, 2760–2767.
12
13
14
15 52. Misirli, G., Hallinan, J. S., Yu, T., Lawson, J. R., Wimalaratne, S. M., Cooling, M. T.,
16 and Wipat, A. (2011) Model annotation for synthetic biology: automating model to
17 nucleotide sequence conversion. *Bioinformatics* 27, 973–979.
18
19
20
21 53. Guet, C. C., Elowitz, M. B., Hsing, W., and Leibler, S. (2002) Combinatorial synthesis
22 of genetic networks. *Science (Washington, DC, U. S.)* 296, 1466–1470.
23
24
25
26
27 54. Menzella, H. G., Reid, R., Carney, J. R., Chandran, S. S., Reisinger, S. J., Patel, K. G.,
28 Hopwood, D. A., and Santi, D. V. (2005) Combinatorial polyketide biosynthesis by de
29 novo design and rearrangement of modular polyketide synthase genes. *Nat. Biotechnol.*
30 23, 1171.
31
32
33
34
35
36 55. Smanski, M. J., Bhatia, S., Zhao, D., Park, Y., Woodruff, L. B., Giannoukos, G.,
37 Ciulla, D., Busby, M., Calderon, J., and Nicol, R. (2014) Functional optimization of
38 gene clusters by combinatorial design and assembly. *Nat. Biotechnol.* 32, 1241.
39
40
41
42
43 56. Cress, B. F., Toparlak, O. D., Guleria, S., Lebovich, M., Stieglitz, J. T., Englaender,
44 J. A., Jones, J. A., Linhardt, R. J., and Koffas, M. A. (2015) CRISPathBrick:
45 modular combinatorial assembly of type II-A CRISPR arrays for dCas9-mediated mul-
46 tiplex transcriptional repression in *E. coli*. *ACS Synth. Biol.* 4, 987–1000.
47
48
49
50
51
52
53
54
55
56
57
58
59
60

For Table of Contents Use Only

With a box:



Without a box:

