ACS SyntheticBiology

Research Article

pubs.acs.org/synthbio

A Methodology to Annotate Systems Biology Markup Language Models with the Synthetic Biology Open Language

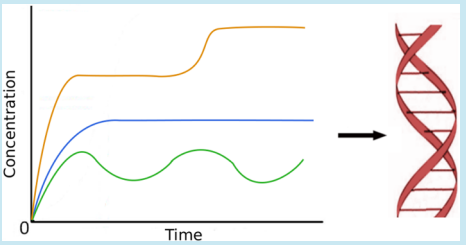
Nicholas Roehner*,† and Chris J. Myers*,‡

†Department of Bioengineering, University of Utah, Salt Lake City, Utah 84112, United States

‡Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, Utah 84112, United States

Supporting Information

ABSTRACT: Recently, we have begun to witness the potential of synthetic biology, noted here in the form of bacteria and yeast that have been genetically engineered to produce biofuels, manufacture drug precursors, and even invade tumor cells. The success of these projects, however, has often failed in translation and application to new projects, a problem exacerbated by a lack of engineering standards that combine descriptions of the structure and function of DNA. To address this need, this paper describes a methodology to connect the *systems biology markup language* (SBML) to the *synthetic biology open language* (SBOL), existing standards that describe biochemical models and DNA components, respectively. Our methodology involves first annotating SBML model elements such as species and reactions with SBOL DNA components. A graph is then constructed from the model, with vertices corresponding to elements within the model and edges corresponding to the cause-and-effect relationships between these elements. Lastly, the graph is traversed to assemble the annotating DNA components into a composite DNA component, which is used to annotate the model itself and can be referenced by other composite models and DNA components. In this way, our methodology can be used to build up a hierarchical library of models annotated with DNA components. Such a library is a useful input to any future genetic technology mapping algorithm that would automate the process of composing DNA components to satisfy a behavioral specification. Our methodology for SBML-to-SBOL annotation is implemented in the latest version of our *genetic design automation* (GDA) software tool, iBioSim.



KEYWORDS: *model annotation, SBML, SBOL, genetic design automation, iBioSim*

The last ten years have witnessed the potential of synthetic biology to improve a diversity of fields, including energy production, pollution control, chemical manufacturing, and medicine. In particular, we note the development of bacteria and yeast that have been genetically engineered to produce biofuels,[1,2] consume toxic waste,[3,4] manufacture drug precursors,[5] and even invade tumor cells.[6] The successes of these individual projects, however, have often failed in translation and application to new projects, a problem exacerbated by the lack of a genetic engineering framework based on standards that combine descriptions of the structure and function of DNA.[7−9] In order to provide a basis for future genetic engineering frameworks, we present a methodology for combining the *synthetic biology open language* (SBOL)[10,11] and *systems biology markup language* (SBML),[12] two existing standards that together can capture the structure and function of DNA.

Ideally, a synthetic biologist could design biological systems at a fairly high level of abstraction, focusing on a desired behavior rather than the DNA components used to implement said behavior. Such a synthetic biologist would use *genetic design automation* (GDA) software tools to automatically map abstract system designs to a standardized set of DNA components (synthesis) and simulate the mapped system designs to check that their behavior is correct (analysis). GDA decreases the expert knowledge required of a designer by encoding it into tools and the standardized data upon which these tools operate. GDA also provides a structure for evaluating design decisions on the behavior of quantitative models that can be specified and communicated unambiguously.

In order to enable GDA and the efficient exchange of functional data on DNA components, it is necessary that structural descriptions of genetic technologies be coupled with models for their behavior in a standardized manner. Existing standards such as GenBank[13] and SBML describe DNA sequences and biochemical models, respectively, but neither standard inherently makes the critical connection between DNA and models to enable simultaneous, modular composition of structure and function. Even the most widely used database for synthetic biology, the iGEM Registry of Standard Biological Parts,[14] does not offer an explicit, standardized means of linking DNA components to models.

SBOL, an emerging standard for synthetic biology with growing GDA tool support,[15−19] is well suited to the task of describing DNA components for association with models. As shown in Figure 1, SBOL currently captures the same sequence-oriented information found in a GenBank file, but unlike GenBank, it allows fully hierarchical annotation of DNA

Received: June 10, 2013

   dx.doi.org/10.1021/sb400066m | *ACS Synth. Biol.* XXXX, XXX, XXX−XXX

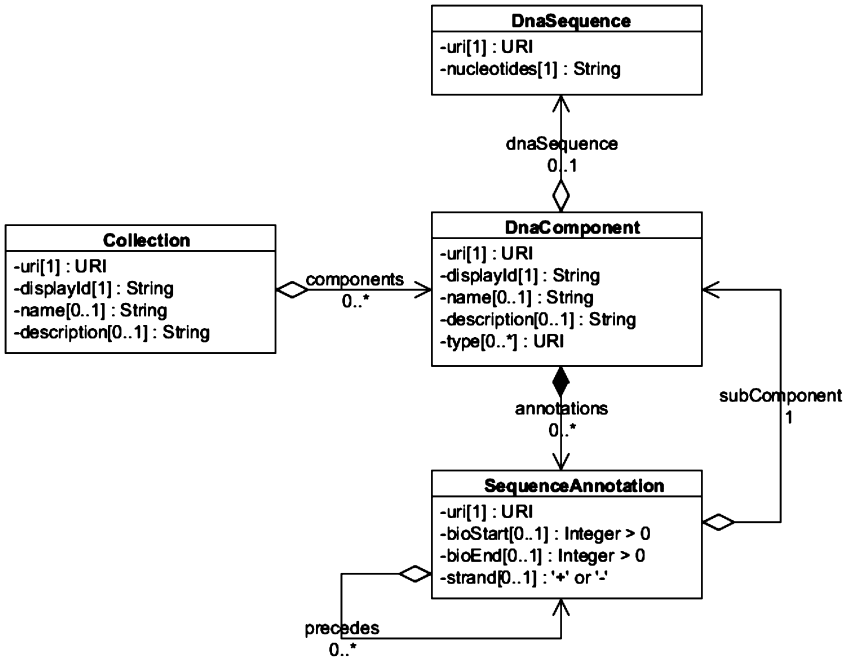


Figure 1. A Unified Modeling Language (UML) diagram for version 1.1 of SBOL.[10] The core data model consists of collections, DNA components, DNA sequences, and sequence annotations. Each of these object classes has a Unique Reference Identifier (URI).[20] DNA components are at the center of SBOL and have a display ID, name, description, Sequence Ontology (SO)[21] type, DNA sequence, and any number of sequence annotations. Note that a sequence annotation points to a DNA subcomponent, thereby enabling fully hierarchical annotation. Also, the precedes relationship of a sequence annotation can be used to indicate the relative ordering of DNA subcomponents.

components within DNA components, thereby mirroring the hierarchy and modularity of models that are useful for engineering systems. In regards to our requirements for GDA, all that SBOL lacks is an accompanying hierarchical description of the behavior of its DNA components.

Rather than create a new modeling standard within SBOL, this paper describes a methodology that supplies behavioral descriptions for SBOL using SBML. SBML is the preeminent standard for modeling biological systems and is supported by over 250 software tools, including our own software, iBioSim.[19,22] While the decision to use SBML may bias our behavioral descriptions toward reaction-based models (see Figure 2), it is also the case that SBML is an actively developed standard with formal procedures for its extension via packages, should the need arise for additional modeling formalisms to represent DNA component behavior. One such package recently released for testing is the *hierarchical model composition* (comp) package,[23] which is supported by iBioSim to enable hierarchical, modular design of genetic circuits. SBML also provides guidelines for annotating models with metadata, that is, data outside SBML that describes the elements of a SBML model, which is the key capacity for connecting SBML to SBOL.

Earlier research on the problem of connecting biochemical models to DNA was carried out by Misirli et al., resulting in the *Model to Sequence Conversion* (MoSeC) software tool.[24] MoSeC can be used to automatically assemble a composite DNA sequence from a SBML model, provided that the model is composed of elements that are annotated with DNA sequences and structured in accordance with the *standard virtual parts* (SVP) approach.[25] Our SBML-to-SBOL methodology builds on the MoSeC approach by extending model annotation and sequence assembly to handle a more general class of SBML model. In particular, our methodology can be applied to hierarchical SBML models constructed using the comp package and to SBML models composed of elements that are not necessarily structured such as the SVPs used by MoSeC.

2. RESULTS AND DISCUSSION

Our methodology for SBML-to-SBOL annotation and assembly is implemented within our GDA software tool, iBioSim. An iBioSim user may import SBOL files into their project, browse and filter the DNA components from these files using the iBioSim SBOL browser, and select DNA components to annotate SBML models created or imported in iBioSim. Automated assembly of a composite DNA component from the annotated elments of a model is initiated when the model is saved. This composite component then annotates the model itself and is saved to a SBOL file of the user's choice in their project. If the user subsequently creates a hierarchical SBML model with a submodel element that references the first model, then the composite component annotating the first model becomes a subcomponent of the composite component assembled for the hierarchical model. In this way, the user may build up a hierarchy of SBML models that is mirrored by the hierarchy of the DNA components annotating these models.

To illustrate the complete process, consider a composite SBML model for a genetic toggle switch similar to those originally designed by Gardner et al.[26] The overall model for our toggle switch is composed of two submodels, one for a gene expressing TetR plus GFP and repressed by LacI (the LacI inverter) and one for a gene expressing LacI and repressed by TetR (the TetR inverter). Figure 2 displays one possible iBioSim representation of the SBML models for the toggle switch and its LacI and TetR inverters.

In order to better facilitate a comparison between our SBML-to-SBOL methodology and MoSeC, the LacI inverter model

dx.doi.org/10.1021/sb400066m | *ACS Synth. Biol.* XXXX, XXX, XXX−XXX

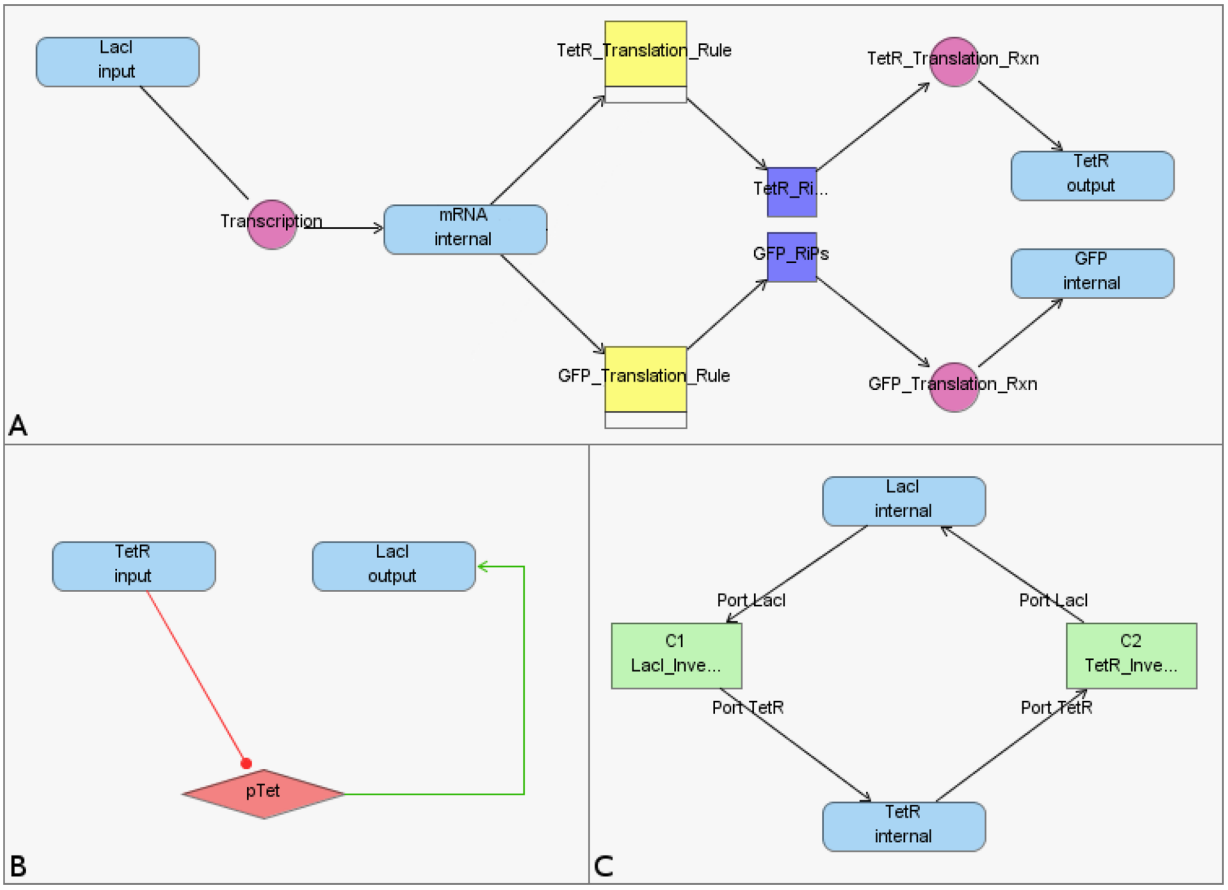


Figure 2. iBioSim representation of the SBML models for the (a) LacI inverter, (b) TetR inverter, and (c) genetic toggle switch. Blue ellipses are species, purple circles are reactions, yellow squares are rules, blue squares are parameters, green rectangles are submodels, and red diamonds are promoter elements that code for sets of reactions and species. Black arcs in the LacI inverter model show which species and parameters are inputs and outputs for which reactions and rules. The red arc in the TetR inverter model is a high-level modeling element that signifies repression of the promoter element pTet by the species TetR, while the green arc signifies production of the species LacI by the promoter element. Lastly, the labeled arcs in the toggle switch model are port mappings that specify that its LacI and TetR species are the same as those species in the LacI and TetR inverter submodels.

uses SVP-like elements. Our methodology, however, can also be applied to other more abstract models in which transcription and translation are combined into a single production reaction, such as those that iBioSim is capable of encoding with high-level modeling elements for promoters and positive and negative genetic regulation.[19] As an example of the foregoing, the TetR inverter model uses these special regulatory modeling elements. All SBML files created for the toggle switch and its LacI and TetR inverters can be found in the Supporting Information.

Model Annotation. The first step in our SBML-to-SBOL methodology is to annotate the elements of models at the lowest level of hierarchy, that is, those that contain no submodels. An iBioSim user may accomplish this step by first opening the appropriate editor for the SBML element they wish to annotate and clicking the "Associate SBOL" button. Doing so brings up a list of all DNA components currently annotating the chosen element. By allowing more than one DNA component to annotate a single element, our tool enables the user to group adjacent DNA components without necessarily having to include them as the subcomponents of a single composite component and thereby introduce superfluous hierarchy. This capability is useful for including DNA components such as terminators that may not have their behavior explicitly modeled but must be present for the design to function correctly. For the purpose of selecting DNA components to add to an element's list, our tool includes a SBOL browser (see Figure 3) that enables the user to filter project DNA components based on their SO type and/or any collections to which they belong.

Let us consider annotating the elements of the LacI inverter and TetR inverter models from our toggle switch example. In the case of the LacI inverter model, let us annotate its transcription reaction with the LacI-regulated promoter BBa_I14032, both of its translation rules with the ribosome binding site (RBS) BBa_B0034, its TetR translation reaction with the coding sequence (CDS) BBa_C0040, and its GFP translation reaction with the CDS BBa_E0040 and the terminator BBa_B0015. As for the TetR inverter model, let us annotate its promoter element with the TetR-regulated promoter BBa_R0040 and its LacI species with the RBS BBa_B0034, the CDS BBa_C0012, and the terminator BBa_B0015. All of these DNA components can be obtained from the iGEM Registry,[14] which has recently been extended with the capacity to export parts in SBOL. To simplify these DNA components for other examples in this paper, let us also delete any sequence annotations they may have, thereby making them noncomposite DNA components. The resulting SBOL files can be found in the Supporting Information.

dx.doi.org/10.1021/sb400066m | ACS Synth. Biol. XXXX, XXX, XXX−XXX

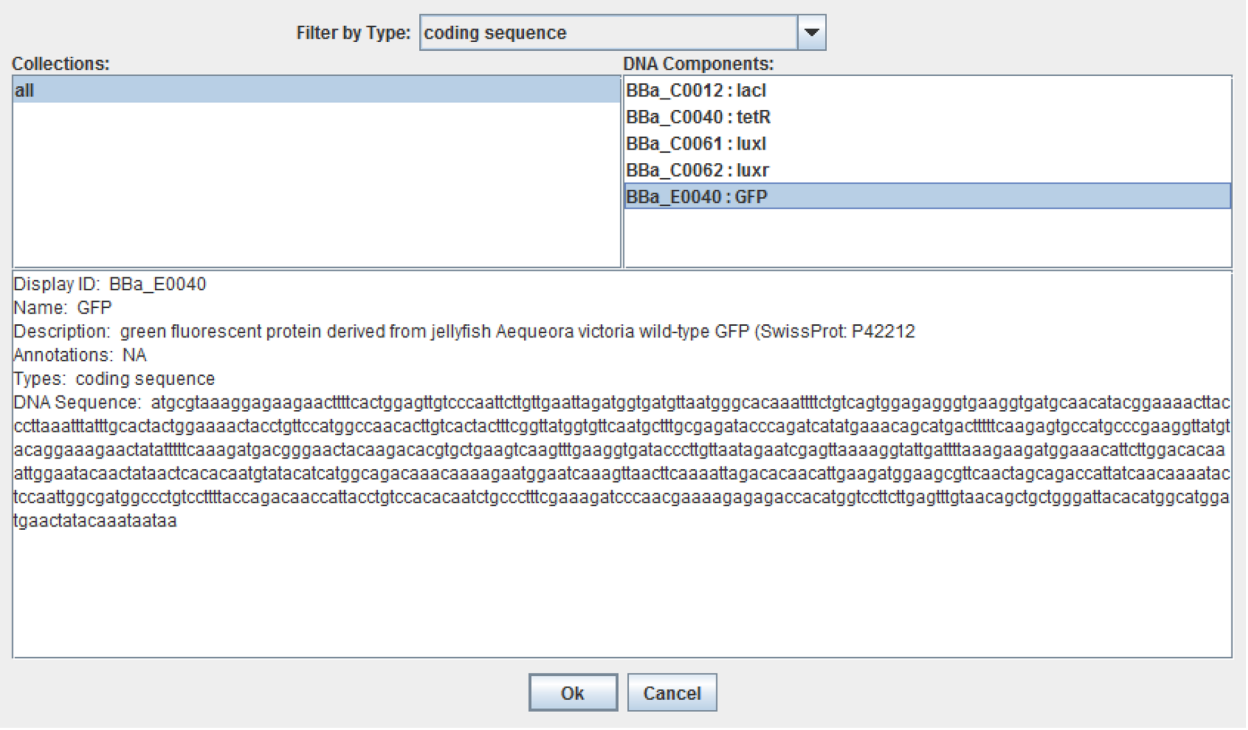


Figure 3. SBOL Browser view of the coding sequence for GFP. The filter-by-type combo box may be used to view all DNA components of a particular SO type within the project. The information displayed for a selected DNA component includes display ID, name, description, sequence annotations, SO types, and DNA sequence. In this case, GFP is not a composite DNA component and therefore is not annotated with any subcomponents.

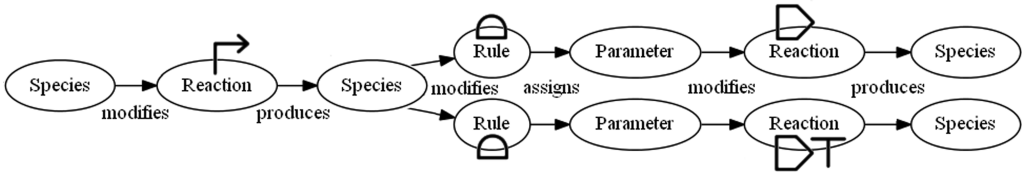


Figure 4. Graph constructed from the LacI inverter model. Each vertex of the graph has been labeled with the type of SBML element to which it correponds, while each edge has been labeled with the type of input/output relationship that exists between the SBML elements corresponding to the edge's origin and destination vertices. The DNA components stored at these vertices are shown using symbols taken from the SBOL visualization standard.[27] In particular, the bent arrow is a promoter, the half circle is a RBS, the box arrow is a CDS, and the "T" is a terminator.

DNA Component Assembly. The second step in our SBML-to-SBOL methodology is to assemble composite DNA components from the DNA components annotating the elements of our models. An iBioSim user may accomplish this step by simply saving their model, at which point iBioSim takes over the two processes required for composite DNA component assembly: graph construction and graph traversal. In a nutshell, graph construction involves the conversion of an annotated SBML model to a graph representation that captures the flow of information through the model, while graph traversal involves walking this graph and ordering its vertices such that their stored DNA components can be concatenated as subcomponents of a composite DNA component.

Graph Construction. During graph construction, vertices are created for each SBML element in a model, and any DNA components annotating a given element are stored at its corresponding vertex. Next, edges directed from one vertex to the next are created to capture the cause−and−effect relationships between elements of the model. This step results in edges pointing from the vertices for elements that represent quantities (such as species or parameters) to the vertices for elements that represent processes (such as reactions or rules) and vice versa. The direction of an edge between a vertex for a quantity element and a vertex for a process element is determined by whether the quantity element is an input or output of the process element. Figure 4 displays the graph constructed from the annotated LacI inverter model in our toggle switch example.

In the case of a hierarchical SBML model, graph construction must be modified to account for the presence of submodel elements. Whenever a vertex is created for a submodel element, if there are no DNA components directly annotating this element, then any DNA components annotating its referenced external model are stored at the vertex instead. In this way, DNA components that annotate models lower in the hierarchy are propagated upward to become subcomponents of DNA components that annotate models higher in the hierarchy.

Next, edge creation must be modified in order to connect the vertices for submodel elements to the vertices for other SBML elements. If a SBML element is marked to replace or be replaced by another element in an external model referenced by a submodel, then a pair of edges is added to connect the vertex

dx.doi.org/10.1021/sb400066m | ACS Synth. Biol. XXXX, XXX, XXX−XXX

for the marked element to the vertex for the submodel and vice versa. Edges are added in both directions between these vertices because the cause−and−effect relationships between elements at different levels of the modeling hierarchy cannot be known without combining the models and replacing or deleting their elements as marked. Flattening the hierarchy of models and associated DNA components in this manner, however, may not always be necessary or desirable. Figure 5 displays the graph constructed from the hierarchical toggle switch model following the assembly of composite DNA components for the LacI inverter and TetR inverter models.

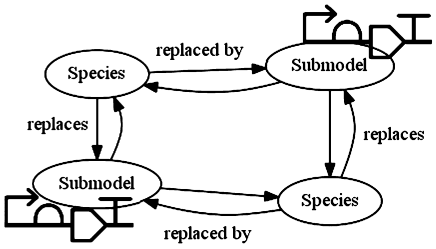


Figure 5. Graph constructed from the hierarchical toggle switch model. Each pair of edges in the graph has been labeled with the model composition relationship between a species element in this model and a species element in the external LacI inverter or TetR inverter model that is referenced by a submodel element in this model. The composite DNA components stored at the submodel vertices are shown by placing SBOL visual symbols for their subcomponents along a solid line.

Flattening of a hierarchical model prior to graph construction is necessary when doing so would change the set of genetic constructs resulting from graph traversal. For example, the set of assembled genetic constructs would change when an unannotated element in one model is marked to replace an annotated element belonging to another model that is lower in the hierarchy. Hence, our overarching strategy when assembling a composite DNA component for a hierarchical model is to construct two graphs, one for the model as it stands and one for its flattened version, then traverse both graphs and compare the partial and complete genetic constructs assembled thereof. If the constructs match with respect to the URIs of their noncomposite subcomponents, then our method annotates the hierarchical model using the DNA component assembled from the unflattened model. Otherwise, our method uses the DNA component assembled from the flattened model. In this way, our methodology seeks to preserve the hierarchy of assembled DNA components, whenever possible, though not at the expense of the set of genetic constructs implied by the cause-and-effect relationships at the lowest level of the modeling hierarchy.

Graph Traversal. During graph traversal, a *depth-first search* (DFS) is performed to order the vertices of the graph such that their stored DNA components may be concatenated to form a valid partial or complete genetic construct. A basic DFS starts at a vertex with no incoming edges and follows outgoing edges until a vertex with no outgoing edges is encountered. The search then backtracks until it can follow an edge not previously taken. This process repeats until all vertices in the graph have been encountered and ordered accordingly. However, because it is desirable for the ordering of our vertices to correspond to a valid ordering of their stored DNA components, our graph traversal cannot rely on a DFS alone, since the choices made by the DFS at branches in the graph are uninformed. For example, there are two possible vertex orderings that a DFS could produce when applied to the LacI inverter graph in Figure 4, only one of which corresponds to a valid genetic construct.

Consequently, our method needs an arbiter to determine when the DFS has chosen a path that leads to an invalid genetic construct. For this purpose, our method uses *deterministic finite automata* (DFAs) constructed from *regular expressions*. In computer science, a regular expression is a common means of specifying a *regular language*, or collection of patterns, while a DFA is a class of state machine used to match inputs against the regular language underlying its construction. Previously, it has been shown that a context-free language composed of DNA component types can be used to verify synthetic genetic constructs, and it has been speculated that simpler regular languages would be sufficient for this task as well.[28]

The default regular expression of iBioSim is *promoter*, $(RBS,CDS)^{+}terminator^{+}$, which specifies that DNA components are to be assembled into genetic constructs composed of a promoter followed by one or more RBS−CDS pairs and one or more terminators. At the start of graph traversal, either this default expression or a custom expression is translated into several DFAs as described in the Methods section. During graph traversal, these DFAs process the SO types of the noncomposite DNA components stored at each vertex encountered. Whenever an invalid genetic construct would be assembled, backtracking is initiated to find a valid solution if possible. Figure 6 shows a sample DFA translated directly from iBioSim's default regular expression for a complete genetic construct.

When our methodology is applied to the toggle switch example, two separate graph traversals are performed on the hierarchical and flattened versions of the toggle switch model. Since the toggle switch components assembled in this way are identical, the component chosen to annotate the toggle switch model is that which possesses the greatest degree of hierarchy, that is, that assembled from the hierarchical version of the model. This composite toggle switch component contains both

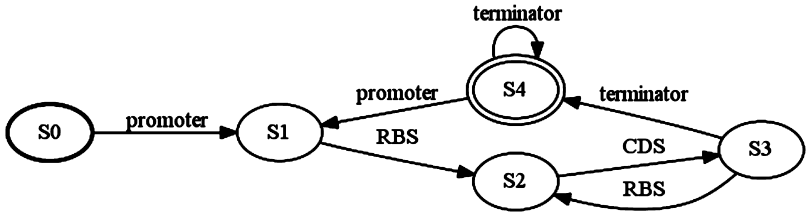


Figure 6. DFA translated from iBioSim's default regular expression for a complete genetic construct. State S0 is the start state. If an input matches the label of an outgoing edge of the current state, then a transition is made from this state to the edge destination. If there is no match for an input, then the DFA rejects the entire input sequence. The DFA only accepts an input sequence if it ends in the accept state S4.

dx.doi.org/10.1021/sb400066m | *ACS Synth. Biol.* XXXX, XXX, XXX−XXX

LacI inverter and TetR inverter subcomponents, which in turn contain non−composite subcomponents such as promoters, terminators, etc. By comparison, the other (also composite) toggle switch component contains these noncomposite subcomponents, but not the intermediate LacI inverter or TetR inverter components. The composite SBOL and annotated SBML resulting from the application of our methodology to the toggle switch can be found in the Supporting Information.

Discussion. This paper describes a methodology for annotating SBML models with SBOL, which is implemented it in our GDA software tool, iBioSim. Our SBML-to-SBOL methodology builds on the results of previous software tools such as MoSeC by extending the processes of graph construction and traversal, which form the basis for model-to-DNA component assembly. In particular, this paper enhances graph construction to handle hierarchical SBML models built via the comp package and a greater variety of SBML elements such as submodels and events. This paper also supplements graph traversal with an arbiter in the form of a DFA constructed from a regular expression for patterns of SO types, thereby providing the user with a means of programming our methodology to recognize a greater variety of genetic constructs. By allowing the user to customize the source of arbitration for graph traversal, our SBML-to-SBOL methodology is more flexible when it comes to assembling DNA components into a variety of genetic constructs.

Still, there is one major assumption that dictates the class of genetic constructs that it is possible to assemble via our methodology. Namely, our method assumes that the partial ordering of the modeling elements that describe the behavior of a genetic construct, as inferred from the cause-and-effect relationships between these elements, is equivalent to the partial ordering of the DNA components that annotate these elements, from which a total ordering or sequence of these DNA components that is a valid genetic construct may be determined. By means of this assumption, however, our methodology can be used to efficiently create a hierarchical library of SBML models annotated with SBOL DNA components, a useful input to any future genetic technology mapping algorithm that would automate the process of composing DNA components to satisfy a behavioral specification.

Our future research on model-to-DNA component annotation and assembly involves working within the SBOL Developers Group to develop a modeling extension for connecting back from SBOL elements to models written in SBML and other modeling standards. Currently, a BioBricks Foundation Request for Comments (BBF RFC)[29] describing the modeling extension is being written. Upon its completion, the RFC will be made publicly available and submitted for approval to the SBOL Developers Group as a whole.

Once it is integrated into SBOL, the modeling extension will set a precedent for SBOL designs to serve as the central hub for attaching nonsequence related information on DNA components and the devices and systems to which they belong. In the case of the SBOL modeling extension, this information will be models for system behavior, the languages of these models (e.g., SBML, CellML[30]), their frameworks (e.g., ordinary differential equation, stochastic), the roles they play (e.g., simulation, specification), and lists of interactions between system components that are derivable from these models. Other extensions will introduce different types of information such as host context and measurements.

The end goal is to enable different GDA software tools to exchange a single genetic design, yet allow each tool to focus on the aspects of the design that are most relevant to that tool's purpose. For example, iBioSim would be used to associate SBOL designs with SBML models to create a library of modeled designs, synthesize from the library a composite SBOL design associated with a composite SBML model to meet a behavioral specification, and finally verify the behavior of the composite design through simulation and model checking of its SBML against the specification. The composite SBOL design could then be sent to another tool to perform other steps of the genetic design process such as the determination of a physical assembly protocol.

METHODS

Model Annotation. Similar to the MoSeC approach, our SBML-to-SBOL methodology leverages the *resource description framework* (RDF)[31] annotations serialized in *extensible markup language* (XML)[32] and follows the guidelines for such annotations as given by the creators of SBML. Figure 7 displays the typical format for our SBML-to-SBOL annotations. Each annotation specifies a SBML element such as a model, species, reaction, rule, parameter, event, or submodel as its subject and a list of DNA components as its object.

```
<SBML_ELEMENT + + + metaid="SBML_META_ID" + + + >
  <annotation>
    <ModelToSBOL xmlns="http://sbolstandard.org/modeltosbol/1.0#">
      <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
               xmlns:mts="http://sbolstandard.org/modeltosbol/1.0#">
        <rdf:Description rdf:about="#SBML_META_ID">
          <mts:DNAComponents>
            <rdf:Seq>
              <rdf:li rdf:resource="DNA_COMPONENT_URI"/>
              . . .
            </rdf:Seq>
          </mts:DNAComponents>
        </rdf:Description>
      </rdf:RDF>
    </ModelToSBOL>
  </annotation>
</SBML_ELEMENT>
```

Figure 7. General form for a RDF/XML SBML-to-SBOL annotation on a SBML element. The first line of XML is that of the SBML element, which contains a meta ID among other information as indicated by +. Next is the SBML-to-SBOL annotation, with its first two lines containing namespaces that distinguish it from other types of annotations and indicate how it should be processed. The actual content of the annotation is highlighted in red, green, and blue. The red line is the subject of the annotation, which is a SBML element identified using its meta ID. The green line is the predicate of the annotation, which indicates that the subject is associated with DNA components. The blue line is the object of the annotation, which is an ordered sequence of URIs identifying a list of DNA components.

DNA Component Assembly. Algorithm 1 lies at the root of the process for automatically assembling composite DNA components with iBioSim. The algorithm implements and composes the aforementioned processes of graph construction and graph traversal, selecting a sequence of starting vertices VS for graph traversal. The algorithm also initiates the process of constructing a sequence of DFAs, $D = \langle DC, DP, DS, DT \rangle$, which is a necessary input for recognizing genetic constructs during graph traversal.

dx.doi.org/10.1021/sb400066m | ACS Synth. Biol. XXXX, XXX, XXX−XXX

Algorithm 1: Assemble Composite DNA Component

Input: Regular expression u and SBML model $M = \langle S, R, L, P, N, C\rangle$ where S is a set of species, R is a set of reactions, L is a set of rules, P is a set of global parameters, N is a set of events, and C is a set of submodels
Output: Composite DNA component k

$\langle G, \mu\rangle \leftarrow$ CONSTRUCT-GRAPH(M)
if $|C| > 0$ **then**
  $G \leftarrow$ CONSTRUCT-GRAPH-FROM-SUBMODELS(G, μ, M)
$VS \leftarrow \langle\rangle$
for $v \in V$ **do**
  if $predecessors(v) = \emptyset$ **then**
    $VS \leftarrow VS||v$
$D \leftarrow$ CONSTRUCT-DFAS(u)
$VO \leftarrow$TRAVERSE-GRAPH($G, VS, \langle\rangle, \langle\rangle, \langle\rangle, \{\}, D$)
for $vo \in VO$ **do**
  $addSubComponents(k, dnaComponents(vo))$
return k

Primitive routines called by Algorithm 1 include the vertex routines *predecessors* and *dnaComponents*, as well as, the DNA component routine *addSubComponents*. The routine *predecessors* returns all vertices that precede a given vertex in the graph G, while *dnaComponents* returns all DNA components that are stored at the given vertex during graph construction. Lastly, *addSubComponents* takes a list of DNA components as input and makes them subcomponents of the given DNA component.

Graph Construction. Algorithms 2 and 3 handle the process of graph construction. Algorithm 2 initiates graph

Algorithm 2: Construct Graph

Input: SBML model $M = \langle S, R, L, P, N, C\rangle$ where S is a set of species, R is a set of reactions, L is a set of rules, P is a set of global parameters, N is a set of events, and C is a set of submodels
Output: Graph $G = \langle V, E\rangle$ and mapping $\mu : X \rightarrow V$

$V \leftarrow \emptyset$
$E \leftarrow \emptyset$
$X = S \cup R \cup L \cup P \cup N$
for $x \in X$ **do**
  $dnaComponents(v) \leftarrow$ PARSE-ANNOTATION($annotation(x)$)
  $V \leftarrow V \cup v$
  $\mu(x) \leftarrow v$
for $r \in R$ **do**
  for $s \in reactants(r) \cup modifiers(r)$ **do**
    $E \leftarrow E \cup \langle\mu(s), \mu(r)\rangle$
  for $x \in$ PARSE-IDENTIFIERS($kineticLawMath(r)$) **do**
    $E \leftarrow E \cup \langle\mu(x), \mu(r)\rangle$
  for $s \in products(r)$ **do**
    $E \leftarrow E \cup \langle\mu(r), \mu(s)\rangle$
for $l \in L$ **do**
  if $isAssignmentRule(l) \vee isRateRule(l)$ **then**
    for $x \in$ PARSE-IDENTIFIERS($math(l)$) **do**
      $E \leftarrow E \cup \langle\mu(x), \mu(l)\rangle$
    $E \leftarrow E \cup \langle\mu(l), \mu(variable(l))\rangle$
for $n \in N$ **do**
  for $x \in$ PARSE-IDENTIFIERS($triggerMath(n)$)$\cup$PARSE-IDENTIFIERS($delayMath(n)$)$\cup$PARSE-IDENTIFIERS($priorityMath(n)$)$\cup$PARSE-IDENTIFIERS($eventAssignmentMaths(n)$) **do**
    $E \leftarrow E \cup \langle\mu(x), \mu(n)\rangle$
  for $x \in eventAssignments(n)$ **do**
    $E \leftarrow E \cup \langle\mu(n), \mu(x)\rangle$
return $\langle G, \mu\rangle$

construction from the nonhierarchical elements of the input SBML model and creates a mapping from these elements to their corresponding graph vertices to assist in edge creation. Algorithm 3, on the other hand, finishes graph construction from any submodel elements, if they are present.

Functions called by Algorithms 2 and 3 include PARSE–ANNOTATION, PARSE–IDENTIFIERS, and LOAD–EXTERNAL–MODEL. The PARSE–ANNOTATION function takes as input a RDF/XML annotation and determines if it is a SBML-to-SBOL annotation. If so, then the function resolves

Algorithm 3: Construct Graph From Submodels

Input: Graph $G = \langle V, E\rangle$, mapping $\mu : X \rightarrow V$, and SBML model $M = \langle S, R, L, P, N, C\rangle$ where S is a set of species, R is a set of reactions, L is a set of rules, P is a set of global parameters, N is a set of events, and C is a set of submodels
Output: Graph $G = \langle V, E\rangle$

for $c \in C$ **do**
  $dnaComponents(v) \leftarrow$ PARSE-ANNOTATION($annotation(c)$)
  if $|dnaComponents(v)| = 0$ **then**
    $W \leftarrow$ LOAD-EXTERNAL-MODEL(c)
    $dnaComponents(v) \leftarrow$ PARSE-ANNOTATION($annotation(W)$)
  $V \leftarrow V \cup \{v\}$
  $\mu(c) \leftarrow v$
$X = S \cup R \cup L \cup P \cup N$
for $x \in X$ **do**
  for $c \in submodels(replacements(x)) \cup submodel(replacedBy(x))$ **do**
    $E \leftarrow E \cup \langle\mu(x), \mu(c)\rangle$
    $E \leftarrow E \cup \langle\mu(c), \mu(x)\rangle$
return G

the list of URIs specified as the annotation object and returns the corresponding list of DNA components. The PARSE–IDENTIFIERS function takes as input a MathML object and returns the SBML elements corresponding to the identifiers (not operators or numbers) found in the MathML. Lastly, the LOAD–EXTERNAL–MODEL function takes as input a submodel, determines its corresponding external model definition, and returns the external model defined.

Primitive routines called by Algorithms 2 and 3 are indicated with italicized text and belong to graph vertices as well as SBML elements. The routines belonging to SBML elements return a variety of data on these elements as described in the SBML specification,[33] while the vertex routine *dnaComponents* returns the list of DNA components currently stored at the given vertex.

DFA Construction. While this paper does not present the algorithmic details of DFA construction, it does briefly describe the process here. DFA *DC* recognizes patterns of SO types corresponding to complete genetic constructs. It is constructed by first quantifying the user's regular expression for a complete genetic construct with the + (one or more) operator and then converting the quantified regular expresssion to a DFA using a method similar to that of McNaughton and Yamada.[34] For example, the regular expression $p(r,c)^+t^+$ would be quantified as $(p(r,c)^+t^+)^+$ and then converted to a DFA that can recognize patterns such as *p, r, c, r, c, t, t* and *p, r, c, t, p, r, c, t*.

DFA *DT*, however, also recognizes patterns for partial genetic constructs that end with the same type of DNA component as a complete genetic construct. It is constructed by first expanding the user's quantified expression with a nonstandard operator and then converting the expanded expression to a DFA as before. For example, the quantified expression $(p(r,c)^+t^+)^+$ would be expanded to $(p(r, c)^+t^+|(r, c|c)(r, c)^*t^+|t, t^*)(p(r, c)^+t^+)^*$ and then converted to a DFA that can recognize patterns such as *r, c, r, c, t, t* and *r, c, t, p, r, c, t*. The full semantics for our expansion operator can be found in the Supporting Information.

Unlike *DT*, DFA *DS* also recognizes patterns for partial genetic constructs that begin rather than end with the same type of DNA component as a complete genetic construct. It is constructed in the same manner as *DT*, with the exception that the ordering of the user's quantified expression is reversed prior to its expansion. Hence, the resulting DFA can recognize patterns such as *c, r, c, r, p* and *c, r, p, t, c, r, p*. When *DS* is used during graph traversal, its input SO types are reversed to ensure proper construct recognition.

Lastly, DFA *DP* also recognizes patterns for partial genetic constructs with no restrictions on starting or terminal DNA

dx.doi.org/10.1021/sb400066m | ACS Synth. Biol. XXXX, XXX, XXX–XXX

component types. It is constructed by making a copy of *DT* and marking every one of its states as accepting. The resulting DFA can recognize patterns such as *r, c, t, t, p* and *r, c, t, p, r, c, t, p*.

Graph Traversal. Algorithms 4, 5, 6, 7, and 8 handle the process of graph traversal. Algorithm 4 composes the other graph traversal algorithms to find a sequence of ordered vertices *VO* with a particular property. Namely, when the DNA components stored at each vertex in *VO* are concatenated to form a composite DNA component, the result is one or more valid genetic constructs.

Algorithm 4: Traverse Graph

Input: Graph $G = \langle V, E\rangle$, sequence of starting vertices VS, sequence of current vertices VC, sequence of local vertices VL, sequence of ordered vertices VO, set of global vertices VG, and sequence of DFAs $D = \langle DC, DP, DS, DT\rangle$

Output: Sequence of ordered vertices VO

```
while |VS| > 0 do
    if |VC| = 0 then
        VC ← VC||VS_0
    while |VC| > 0 do
        if runDFA(DP, types(VC_0)) then
            runDFA(DT, types(VC_0))
            VL ← VL||VC_0
            VN ← DETERMINE-NEXT-VERTICES(VS, VC, VL, VG, DP, DT)
            VC ← sub(VC, 1, |VC| − 1)
            if |VN| = 0 then
                while |VC| > 0 ∧ VC_0 ∈ VL do
                    VC ← sub(VC, 1, |VC| − 1)
            else if |VN| = 1 then
                VC ← VN_0||VC
            else if |VN| > 1 then
                return TRAVERSE-BRANCHES(G, VS, VC, VN, VL, VO, VG, D)
        else
            return ⟨⟩
    reset(DP)
    reset(DT)
    if ¬ORDER-LOCAL-VERTICES(VL, VO, DS, DT) then
        return ⟨⟩
    VG ← VL
    VL ← ⟨⟩
    while |VS| > 0 ∧ VS_0 ∈ VG do
        VS ← sub(VS, 1, |VS| − 1)
if |VG| < |V| then
    return TRAVERSE-CYCLES(G, VO, VG, D)
else
    return VO
```

Algorithm 4 finds *VO* using three sequences of vertices and one set of vertices to manage a series of DFSs. Each vertex in the sequence *VS* is the starting point for a DFS that potentially results in a valid genetic construct. *VS* is initially populated with vertices that have no incoming edges but is later expanded with vertices that follow the boundary of a genetic construct (as determined in Algorithm 5).

Algorithm 5: Determine Next Vertices

Input: Sequence of starting vertices VS, sequence of current vertices VC, sequence of local vertices VL, set of global vertices VG, partial construct DFA DP, and terminal construct DFA DT

Output: Sequence of next vertices VN

```
VN ← ⟨⟩
for vn ∈ successors(VC_0) do
    VP ← predecessors(vn) \ VC_0
    if |VP| = 0 ∨ (¬inStartState(DT) ∧ ¬inAcceptState(DT)) ∨
    (VP ⊆ VL ∧ vn ∉ VL) ∨ (VP ⊆ VG ∧ vn ∉ VG) then
        if |types(vn)| > 0 ∧
        ((inAcceptState(DT) ∧ ¬checkDFA(DT, firstType(vn))) ∨
        ¬checkDFA(DP, firstType(vn))) then
            VS ← VS||vn
        else if vn ∉ VL then
            VN ← VN||vn
return VN
```

Next, sequence *VC* stores the vertices that are currently under consideration during a DFS from a given starting vertex. Though only one vertex from *VC* is considered at a time (the first vertex in the sequence, VC_0), *VC* may contain more than one vertex when the graph branches, and it becomes necessary to keep track of the root vertices for the branches that are not immediately traversed in a depth-first fashion. When a vertex from *VC* is considered, it is replaced with its successors as determined by Algorithm 5. When *VC* becomes empty, it is repopulated using the next vertex from *VS*, which signifies the start of a new DFS.

The sequence *VL* stores off each vertex removed from *VC*, thereby keeping a record of the order in which the vertices that are local to the current DFS have been visited. At the conclusion of a given DFS, *VL* is composed with the sequence of ordered vertices *VO* in accordance with Algorithm 6. *VL* is also added to the set of globally visited vertices *VG*, which keeps track of the vertices that have been visited by previous DFSs. *VG* is used at the end of Algorithm 4 to determine whether any vertices have yet to be visited, in which case Algorithm 8 is required to order them. Finally, *VL* and *VG* are also used to prune previously visited vertices from *VC* and *VS*, respectively.

Algorithm 6: Traverse Branches

Input: Graph $G = \langle V, E\rangle$, sequence of starting vertices VS, sequence of current vertices VC, sequence of next vertices VN, sequence of local vertices VL, sequence of ordered vertices VO, set of global vertices VG, and sequence of DFAs $D = \langle DC, DP, DS, DT\rangle$

Output: Sequence of ordered nodes VBO

```
for i ← 0...|VN| − 1 do
    for j ← i...|VN| − 1 do
        VN ← VN||VN_i
        VN ← sub(VN, 0, i)||sub(VN, i + 1, |VN| − i)
        VBO ← TRAVERSE-
        GRAPH(G, copy VS, VN || copy VC, copy VL, copy VO,
        copy VG, copy D)
        if |VBO| > 0 then
            return VBO
return ⟨⟩
```

Primitive routines called by Algorithm 4 include the DFA routines *runDFA* and *reset*, as well as the vertex routine *types* and sequence routine *sub*. The routine *runDFA* takes a sequence of strings as input, transitions the given DFA to a new state accordingly, and returns a Boolean indicating whether the new state is accepting. When called with the partial construct DFA *DP*, this routine determines if an invalid genetic construct would be formed, in which case Algorithm 4 terminates and returns nothing.

The input for *runDFA* is supplied by *types*, which returns the SO type strings for the list of DNA components stored at a given vertex. The routine *reset*, on the other hand, returns the given DFA to its start state and is called at the end of each DFS. Lastly, *sub* returns a subsequence of the given sequence that starts at the indicated index and has the indicated length. This routine is used during Algorithm 4 to effectively delete the first element in a sequence by replacing the sequence with a subsequence that starts at index one and has a length equal to that of the sequence minus one.

As noted previously, Algorithm 5 determines which vertices succeeding the current vertex VC_0 are added to *VC* or *VS* for future consideration. The algorithm accomplishes this task by testing two sets of conditions. The first set of conditions checks for whether a successor vertex *vn* can and should be visited by another DFS, while the second set checks for whether *vn* should be the starting point for a new DFS.

dx.doi.org/10.1021/sb400066m | ACS Synth. Biol. XXXX, XXX, XXX−XXX

In the first set of conditions, if *vn* has predecessors other than $V\ C_0$, then one of two conditions must be true for the current DFS to continue. Either the current DFS must be in the middle of a genetic construct, or all other predecessors of *vn* must have already been visited during the current DFS or other DFSs. The latter condition guarantees that *vn* is visited at least once, while the former prevents *vn* from being visited more than once unless it and its successive vertices potentially store DNA components that appear in more than one genetic construct.

For example, two different promoters could promote the transcription of the same RBS−CDS−terminator combination. Even though this combination should appear in two different locations on DNA, it could annotate a single element of the model such as a species that represents the mRNA resulting from transcription at both promoters, or perhaps a reaction that represents the translation of said mRNA into protein. Hence, using our methodology, the vertex resulting from such a modeling element would have to be visited twice in order to obtain two separate instances of the RBS−CDS−terminator combination on DNA.

In the second set of conditions, if *vn* stores any DNA components, then there is a chance that it marks the beginning of a new genetic construct and should be added to *VS* instead of *VC*. This is the definitely the case if the current DFS is at the end of a genetic construct and the first DNA component stored at *vn* has a SO type other than that of a DNA component found at the end of a complete genetic construct. If the current DFS is not at the end of a genetic construct and the first component at *vn* would lead to an invalid construct, then it is at least possible that *vn* starts a new genetic construct. When neither of these conditions are true, *vn* is added to *VC*, provided that it has not already been visited by the current DFS.

Primitive routines called by Algorithm 5 that have not been described elsewhere include the DFA routines *inAcceptState*, *inStartState*, and *checkDFA*, as well as, the vertex routine *firstType*. Routines *inAcceptState* and *inStartState* return Booleans indicating whether the given DFA is in its accept state or start state, respectively. The routine *checkDFA* functions similarly to *runDFA*, but only returns a Boolean indicating whether the given DFA would be in an accept state after processing a list of input strings and does not actually transition the DFA to a post−input state. Finally, *firstType* returns the SO type of the first DNA component in a list stored at the given vertex.

Once Algorithm 5 has determined which vertices will be added to *VC*, if there are more than one then Algorithm 6 handles the process of finding the order in which these next vertices should be added to *VC* so that a valid genetic construct is obtained. The algorithm achieves this task by permuting the sequence of next vertices *VN* and recursively calling Algorithm 4 with copies of relevant data structures (in case the call fails and the next permutation must be tried).

When the current DFS terminates, Algorithm 7 handles the process of composing the sequence of vertices ordered by the DFS (*VL*) with the sequence of vertices ordered by previous DFSs (*VO*). The algorithm solves this problem using the starting and terminal construct DFAs *DS* and *DT* to determine whether *VL* should be concatenated at the beginning or end of *VO*. While complete genetic constructs can generally be concatenated in any order without introducing cis-interactions between DNA components within these constructs, care must be taken when composing partial genetic constructs. For instance, when composing a single promoter with a RBS−CDS−terminator combination that it should not cis-regulate, the promoter must be placed immediately after the RBS−CDS−terminator combination.

Algorithm 7: Order Local Vertices

Input: Sequence of local vertices VL, sequence of ordered vertices VO, starting construct DFA DS, and terminal construct DFA DT
Output: Sequence of ordered vertices VO

```
if |types(VL)| = 0 ∨ |types(VO)| = 0 ∨
(checkDFA(DS, types(VL)) ∧ checkDFA(DT, types(VO))) then
    VO ← VO||VL
else if checkDFA(DS, types(VO)) ∧ checkDFA(DT, types(VL)) then
    VO ← VL||VO
else
    return FALSE
return TRUE
```

Lastly, Algorithm 8 is responsible for traversing any vertices that are unvisited by the primary graph traversal, that is, vertices that belong to isolated, strongly connected subgraphs. These are cyclic portions of the graph that lack vertices with zero incoming edges to serve as natural starting points for a traversal and that are not reachable from other portions of the graph. The algorithm solves the problem of visiting these cycles by first identifying within them potential starting vertices that store a first DNA component that shares a SO type with a first component in a complete genetic construct. Next, the algorithm tries these starting vertices one at a time by recursively calling Algorithm 4 with copies of relevant data structures in case a given starting vertex does not produce a valid solution. If no valid solutions can be produced in this way, then all remaining vertices within the cycles are tried as starting vertices in the aforementioned manner.

Algorithm 8: Traverse Cycles

Input: Graph $G = \langle V, E\rangle$, sequence of ordered vertices VO, set of global vertices VG, and sequence of DFAs $D = \langle DC, DP, DS, DT\rangle$
Output: Sequence of ordered vertices VCO

```
VC ← V \ VG
VCS ← ∅
for vc ∈ VC do
    if firstType(vc) ⊆ startingTypes(DC) then
        VCS ← VCS ∪ vc
for i ← 0...1 do
    for vcs ∈ VCS do
        VCO ← TRAVERSE-GRAPH(G, ⟨vcs⟩, ⟨⟩, ⟨⟩, copy VO, copy VG, copy D)
        if |VCO| > 0 then
            return VCO
    VCS ← VC \ VCS
return ⟨⟩
```

■ ASSOCIATED CONTENT

Supporting Information

SBOL file and SBOL-annotated SBML files for the genetic toggle switch and its subcomponents, as well as a PDF file describing an expansion operator for regular expressions. This material is available free of charge *via* the Internet at http://pubs.acs.org/.

■ AUTHOR INFORMATION

Corresponding Authors

*E-mail: n.roehner@utah.edu.
*E-mail: myers@ece.utah.edu.

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

We acknowledge the members of the SBOL Developers Group, in particular Goksel Misirli and Anil Wipat, for helpful feedback

on the material presented in this paper. This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0916042 and CCF-1218095. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

(1) Atsumi, S., and Liao, J. C. (2008) Metabolic engineering for advanced biofuels production from *Escherichia coli*. *Curr. Opin. Biotechnol. 19*, 414−419.
(2) Keasling, J. D., and Chou, H. (2008) Metabolic engineering delivers next-generation biofuels. *Nat. Biotechnol. 26*, 298−299.
(3) Brazil, G. M., Kenefick, L., Callanan, M., Haro, A., de Lorenzo, V., Dowling, D. N., and O'Gara, F. (1995) Construction of a rhizosphere pseudomonad with potential to degrade polychlorinated biphenyls and detection of bph gene expression in the rhizosphere. *Appl. Environ. Microbiol. 61*, 1946−1952.
(4) Cases, I., and de Lorenzo, V. (1995) Genetically modified organisms for the environment: Stories of success and failure and what we have learned from them. *Int. Microbiol. 8*, 213−222.
(5) Ro, D.-K., Paradise, E. M., Ouellet, M., Fisher, K. J., Newman, K. L., Ndungu, J. M., Ho, K. A., Eachus, R. A., Ham, T. S., Kirby, J., Chang, M. C. Y., Withers, S. T., Shiba, Y., Sarpong, R., and Keasling, J. D. (2006) Production of the antimalarial drug percursor artemisinic acid in engineered yeast. *Nature 440*, 940−943.
(6) Anderson, J. C., Clarke, E. J., and Arkin, A. P. (2006) Environmentally controlled invasion of cancer cells by engineering bacteria. *J. Mol. Biol. 355*, 619−627.
(7) Endy, D. (2005) Foundations for engineering biology. *Nature 438*, 449−453.
(8) Arkin, A. (2008) Setting the standard in synthetic biology. *Nat. Biotechnol. 26*, 771−774.
(9) Peccoud, J., Anderson, J. C., D. Chandran, D. D., Galdzicki, M., Lux, M. W., Rodriquez, C. A., Stan, G.-B., and Sauro, H. M. (2011) Essential information for synthetic DNA sequences. *Nat. Biotechnol. 29*, 22.
(10) Galdzicki, M.et al. Synthetic Biology Open Language (SBOL) Version 1.1.0. BBF RFC 87, 2012; DOI: 1721.1/73909.
(11) Galdzicki, M. (2013) SBOL: A community standard for communicating designs in synthetic biology. *Figshare*, DOI: 10.6084/m9.figshare.762451.
(12) Hucka, M., et al. (2003) The Systems Biology Markup Language (SBML): A medium for representation and exchange of biochemical network models. *Bioinformatics 19*, 524−531.
(13) Bilofsky, H. S., and Christian, B. (1988) The GenBank genetic sequence data bank. *Nucleic Acids Res. 16*, 1861−1863.
(14) iGEM Registry. 2003; http://parts.igem.org.
(15) Chandran, D., Bergmann, F. T., and Sauro, H. M. (2009) TinkerCell: Modular CAD tool for synthetic biology. *J. Biol. Eng. 3*, 19.
(16) Czar, M. J., Cai, Y. Z., and Peccoud, J. (2009) Writing DNA with GenoCAD. *Nucleic Acids Res. 37*, W40−W47.
(17) Densmore, D.; Van Devender, A.; Johnson, M.; Sritanyaratana, N. (2009) A platform-based design environment for synthetic biological systems. *The Fifth Richard Tapia Celebration of Diversity in Computing Conference: Intellect, Initiatives, Insight, and Innovations* New York, pp 24−29.
(18) Chen, J., Densmore, D., Ham, T. S., Keasling, J. D., and Hillson, N. J. (2012) DeviceEditor visual biological CAD canvas. *J. Biol. Eng. 6*, 1.
(19) Madsen, C., Myers, C., Patterson, T., Roehner, N., Stevens, J., and Winstead, C. (2012) Design and test of genetic circuits using iBioSim. *IEEE Des. Test 29*, 32−39.
(20) Berners-Lee, T.; Fielding, R.; Masinter, L. (2005) *Uniform Resource Identifier (URI): Generic Syntax*, IETF RFC 3986, The Internet Society; http://tools.ietf.org/html/rfc3986.
(21) Eilbeck, K., Lewis, S. E., Mungall, C. J., Yandell, M., Stein, L., Durbin, R., and Ashburner, M. (2005) The Sequence Ontology: A tool for the unification of genome annotations. *Genome Biology 6*, R44.
(22) Myers, C. J., Barker, N., Jones, K., Kuwahara, H., Madsen, C., and Nguyen, N.-P. D. (2009) iBioSim: A tool for the analysis and design of genetic circuits. *Bioinformatics 25*, 2848−2849.
(23) Smith, L. P.; Hucka, M.; Hoops, S.; Finney, A.; Ginkel, M.; Myers, C. J.; Moraru, I.; Liebermeister, W. SBML Level 3 package specification: Hierarchical Model Composition. *SBML Package Specification*, 2012; http://sbml.org/Documents/Specifications/SBML_Level_3/Packages/Hierarchica_Model_Composition_(comp) (accessed May 24, 2012).
(24) Misirli, G., Halliman, 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.
(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) Gardner, T. S., Cantor, C. R., and Collins, J. J. (2000) Construction of a genetic toggle switch in *Escherichia coli*. *Nature 403*, 339−342.
(27) Quinn, J.; Beal, J.; Bhatia, S.; Cai, P.; Chen, J.; Clancy, K.; Hillson, N. J.; Galdzicki, M.; Maheshwari, A.; Umesh, P.; Pocock, M.; Rodriguez, C.; Stan, G.-B.; Endy, D. Synthetic Biology Open Language Visual (SBOL Visual), Version 1.0.0. BBF RFC 93, 2013; DOI: 1721.1/78249.
(28) 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*, 2760−67.
(29) The BioBricks Foundation: RFC. 2008; http://openwetware.org/wiki/The_BioBricks_Foundation:RFC (accessed Aug. 8, 2013).
(30) Hedley, W. J., Nelson, M. R., Bellivant, D. P., and Nielsen, P. F. (2001) A short introduction to CellML. *Phil. Trans. R. Soc. Lond. A 359*, 1073−1089.
(31) Lassila, O.; Swick, R. RDF/XML syntax specification (revised). W3C Recommendation, 2004; http://www.w3.org/TR/rdf-syntax-grammar/ (accessed May 28, 2013).
(32) Bray, T.; Paoli, J.; Sperber-McQueen, C. M.; Maler, E.; Yergeau, F. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation, 2008; http://www.w3.org/TR/xml/ (accessed May 28, 2013).
(33) Hucka, M.; Bergmann, F. T.; Hoops, S.; Keating, S. M.; Sahle, S.; Schaff, J. C.; Smith, L. P.; Wilkinson, D. J. The Systems Biology Markup Language (SBML): language specification for Level 3 Version 1 Core. SBML Specification, 2010; http://sbml.org/Documents/Specifications#SBML_Level_3_Version_1_Core (accessed May 25, 2012).
(34) McNaughton, R., and Yamada, H. (1960) Regular expression and state graphs for automata. *IEEE Trans. Comput. EC−9*, 39−47.