ACS SyntheticBiology

pubs.acs.org/synthbio Research Article

GOLDBAR: A Framework for Combinatorial Biological Design

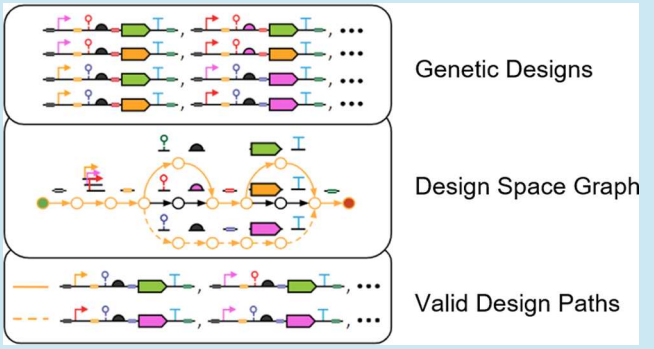
Published as part of ACS Synthetic Biology special issue "IWBDA 2023".

Nicholas Roehner,* James Roberts, Andrei Lapets, Dany Gould, Vidya Akavoor, Lucy Qin, D. Benjamin Gordon, Christopher Voigt, and Douglas Densmore

Cite This: *ACS Synth. Biol.* 2024, 13, 2899–2911 Read Online

ACCESS | Metrics & More | Article Recommendations | Supporting Information

ABSTRACT: With the rise of new DNA part libraries and technologies for assembling DNA, synthetic biologists are increasingly constructing and screening combinatorial libraries to optimize their biological designs. As combinatorial libraries are used to generate data on design performance, new rules for composing biological designs will emerge. Most formal frameworks for combinatorial design, however, do not yet support formal comparison of design composition, which is needed to facilitate automated analysis and machine learning in massive biological design spaces. To address this need, we introduce a combinatorial design framework called GOLDBAR. Compared with existing frameworks, GOLDBAR enables synthetic biologists to intersect and merge the rules for entire classes of biological designs to extract common design motifs and infer new ones. Here, we demonstrate the application of GOLDBAR to refine/validate design spaces for TetR-homologue transcriptional logic circuits, verify the assembly of a partial *nif* gene cluster, and infer novel gene clusters for the biosynthesis of rebeccamycin. We also discuss how GOLDBAR could be used to facilitate grammar-based machine learning in synthetic biology.



KEYWORDS: *biological design, combinatorial engineering, design automation, regular grammar, genetic design, machine learning*

1. INTRODUCTION

Many engineered biological systems such as genetic circuits and biosynthetic pathways lack highly accurate mechanistic models to predict their behavior and performance from their structure and environmental context. To work around this gap in modeling, synthetic biologists are increasingly constructing and screening combinatorial libraries that vary the structural organization of system parts to find combinations that perform well.[1−9] This combinatorial approach to engineering biological systems has been fueled by the development of new libraries of DNA parts[10−22] and new techniques and technologies for assembling DNA parts into libraries of DNA constructs.[23−37]

Despite these advancements in laboratory techniques and technology, the adoption of formal frameworks and software tools for combinatorial biological design has lagged behind. In particular, several frameworks and tools for combinatorial design have been developed in the past decade, including those based on constraints,[38] regular grammars,[39] and context-free grammars,[40] but none of them have risen to the level of a standard within the greater synthetic biology community. Possible explanations for this trend include the difficulty in developing easy-to-use software for biological design and the relative immaturity of standardization in synthetic biology as a whole.[41] Another possible explanation is that current frameworks and tools for combinatorial design do not yet address some of the most pressing challenges facing synthetic biologists, including the refinement of massive combinatorial biological design spaces, the inference of rules for biological design, and the sharing and storage of these spaces and rules.

As combinatorial libraries continue to be used to generate data on which designs perform well and which do not, new rules for biological design will continue to emerge.[42] In many laboratories, however, the state of the art in combinatorial design still involves specifying libraries as collections of files adhering to legacy bioinformatics formats such as FASTA and GenBank, from which it can be difficult to infer the existence of shared design motifs and common design rules without significant preprocessing of the data (for example, with algorithms such as BLAST). To the extent that rules exist to describe library structure, they are commonly documented

Received: April 26, 2024
Revised: August 3, 2024
Accepted: August 6, 2024
Published: August 20, 2024

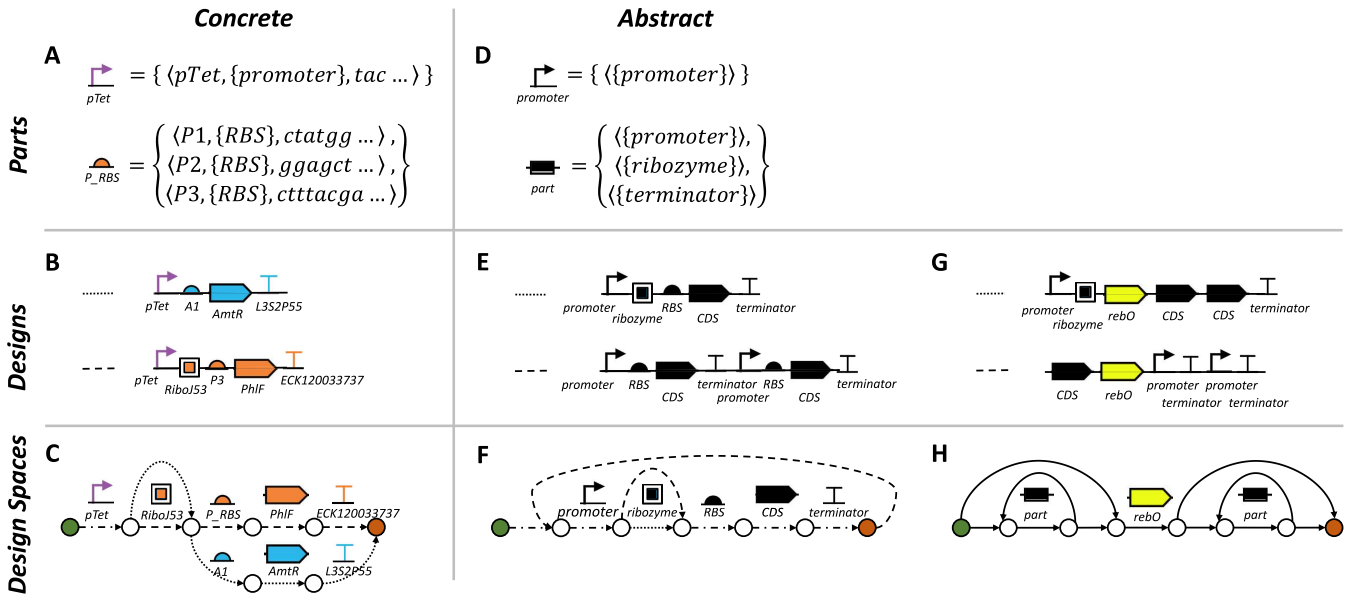


Figure 1. (A) Examples of sets of concrete parts represented using colored glyphs from the Synthetic Biology Open Language (SBOL) Visual 2 standard.[45] Shown here is a set containing the concrete promoter pTet. Also shown is a set of concrete RBS parts P1, P2, and P3. (B) Examples of two concrete designs for single TU constructs. In these designs, each SBOL Visual glyph represents a single concrete part rather than a set of parts. (C) Graph representation of a concrete construct space that encodes these concrete designs. Green nodes are "start" nodes, and red nodes are "accept" nodes. Each edge is labeled with an SBOL Visual glyph that represents a set of concrete parts. Designs encoded by this graph are obtained by enumerating valid paths through the graph from a start node to an accept node and concatenating one part from each edge of a path. For example, the dashed path encodes the design for the orange TU construct with a ribozyme. (D) Examples of sets of abstract parts represented using black glyphs from the SBOL Visual 2 standard.[45] Shown here is a set containing an abstract promoter. Also shown is a set containing an abstract promoter, ribozyme, and terminator. (E) Examples of abstract designs for monocistronic TU and polycistronic TU architectures. Each SBOL Visual glyph in these designs represents a single abstract part. (F) Graph representation of an abstract architecture space that encodes these abstract designs. For example, the dashed path encodes the design for the polycistronic TU architecture. (G) Examples of abstract designs for TU constructs containing single copies of the *rebO* CDS. (H) Graph representation of an abstract rule space that encodes both of these abstract designs in accordance with the rule "contains at least one copy of the *rebO* CDS.".

with prose in methods section of a manuscript or they are implicitly encoded as entries in an electronic spreadsheet. For example, a spreadsheet might contain lists of parts from which a designer intends to select individual entries and concatenate them into different permutations to design a library of constructs. Unfortunately, neither of these encodings tends to facilitate the direct application of software for automated design and machine learning.

Biological design rules must be formally encoded for several reasons. First, formal encoding enables design rules to be shared unambiguously. Second, formally encoded design rules can be consistently reasoned over and used in proofs for the refinement and validation of existing designs and the composition of new designs. Finally, formally encoded rules can be more directly processed and modified by computers in an automated manner and can be used to structure or generate features for machine learning models. Automation is critical for navigating and learning from biological design spaces given that even a modest DNA parts library of five promoters, five ribosome binding sites (RBSs), five coding sequences (CDSs), and five terminators can implement over six billion possible 5-gene cluster designs when selecting parts without replacement and composing them in different orders and orientations ($5^4 \times 4^4 \times 3^4 \times 2^4 \times 1^4 \times 2^5$). In order for a framework to effectively support automation and machine learning for combinatorial biological design, we believe that it should exhibit at least three characteristics.

First, it should be representative of synthetic biology. In other words, the framework should be capable of representing structural motifs commonly found in biological designs such as transcriptional units (TUs), operons, and gene clusters. It should also be capable of representing concepts common to engineering design in general, such as abstract design templates that can be filled in and refined. Representation impacts the class of designs that a user can generate and validate within a framework. Second, the specifications for design motifs within the framework should be compact. That is, they should summarize patterns found across multiple designs without requiring the same amount of memory to individually encode the said designs. Compactness can impact the ease with which specifications are written in the framework. Third, different specifications made using the same framework should be comparable to each other. Ideally, the framework should provide methods for intersecting, merging, and otherwise comparing specified motifs in order to determine which designs they have in common and to infer new designs based on their similarity. Comparability is a key enabler for standardization and fundamental applications to support the reuse of designs including design composition, refinement, and validation.

At present, there is no design framework for synthetic biology that exhibits all three of these characteristics to the degree that we believe is necessary to support automation and machine learning for combinatorial design. In particular, there exist frameworks based on constraints[38,43] and context-free grammars[40] that can compactly specify a wide range of biological design motifs, but these frameworks do not support formal comparison of said motifs. For example, the

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911

combinatorial extension of the synthetic biology open language (SBOL)[38,43] permits users to specify combinatorial designs as templates with constraints between typed subcomponents (promoter, terminator, etc.) and different options for replacing these subcomponents to derive an individual design. This standard for representing combinatorial designs, however, does not include operators for intersecting or merging them in order to compare them or generate novel motifs. Previously, we developed a framework based on regular grammar that supported the intersection of biological design spaces,[39] but even this framework was incapable of directly representing certain engineering concepts adopted by synthetic biology, most notably abstraction. Without abstraction, it is impossible to refer to abstract parts that exist outside of a known library (e.g., "any promoter"), which prohibits more generalized composition and validation of design spaces.

In order to support automation and machine learning for combinatorial biological design, we present a formal framework called GOLDBAR (Grammars for cOmbinatoriaL Design and Bio-Assembly Revision). We have enhanced the expressiveness of GOLDBAR compared to that of our previous framework for combinatorial biological design by adding the ability to represent abstract designs. This has involved adding a new tolerance parameter to the *AND* operator to enable control over how abstract design spaces are intersected. In addition, we have extended GOLDBAR with a new *MERGE* operator to merge design spaces such that they encode novel designs based on their structural similarity. We demonstrate the utility of these new operators by applying them to case studies involving the combinatorial design of genetic circuits and the combinatorial design and assembly of biosynthetic gene clusters. We have implemented GOLDBAR in two open-source software tools (Constellation[1] and Knox[2]) and demonstrated the exchange of combinatorial design information between them using the SBOL 2 standard.[44] Finally, we have also developed an open-source software tool (GOLDBAR Generator[3]) for users to quickly generate GOLDBAR specifications based on known principles of genetic design. These tools are freely available to the public and can be integrated into existing design-build-test workflows alongside synthetic biology's major software offerings.

2. RESULTS

2.1. Case Studies. The following case studies focus on the application of GOLDBAR to three common scenarios in biological design specification: specifying a library of biological designs for testing, validating whether designs can be assembled from available parts, and inferring novel designs based on previous specifications. In particular, the first case study demonstrates the application of the *AND* operator to refine and validate a library of genetic circuit designs. The second use case then demonstrates the application of the *AND* operator to verify whether polycistronic TUs can be assembled for nitrogen fixation gene cluster designs. Finally, the third use case shows the application of the *MERGE* operator to infer novel biosynthetic gene cluster designs.

Before we describe these case studies in detail, we first define the following terms to motivate a more informal description of their contents (see also Figure 1)

2.1.1. Design Space. A design space is a possibly infinite set of DNA designs, each one consisting of a sequence of DNA parts. Design spaces have multiple possible representations (see Section B in Supporting Information), but they are represented as directed graphs in the use cases that follow. Figure 1C shows an example of a simple design space that encodes several different TU designs.

2.1.2. Level of Abstraction. Parts, designs, and design spaces can be described as concrete or abstract. As explained in the last set of definitions, the level of abstraction of a design space has implications for whether the *AND* operator and *MERGE* operator can be applied to the design space.

A **concrete part** has an ID that uniquely identifies it within some namespace (such as a part database or the World Wide Web), a DNA sequence, and one or more roles (see Figure 1A).

A **concrete design** contains only concrete parts (see Figure 1B).

A **concrete design space** encodes only concrete designs (see Figure 1C).

An **abstract part** has one or more roles but no ID and no sequence (see Figure 1D).

An **abstract design** contains at least one abstract part (see Figure 1E,1G).

An **abstract design space** encodes at least one abstract design (see Figure 1F,1H).

2.1.3. Types of Design Spaces. There are different types of design spaces that are useful for different applications.

A **construct space** is a concrete or abstract design space that encodes designs adhering to a known construct architecture, such as one or more transcription units (TUs) or operons (see Figure 1C). Construct spaces are typically used to specify libraries of related designs that can be selected and built for testing.

An **architecture space** is an abstract design space that encodes all possible designs adhering to a known construct architecture (see Figure 1F). An architecture space is typically used with the *AND* operator to refine a construct space by removing all designs that do not adhere to the architecture. Doing so also validates whether at least one design encoded by the construct space adheres to the architecture.

A **rule space** is a concrete or abstract design space that typically encodes many designs adhering to a small number of rules (see Figure 1H). These designs may or may not adhere to a known construct architecture. A rule space is typically used with the *AND* operator to refine a construct space by removing designs that do not follow the rule. Doing so also validates whether at least one design encoded by the construct space follows the rule.

*2.1.4. AND and **MERGE** Operators.* The *AND* operator now has a tolerance parameter that can range from 0 to 2. This parameter controls the criteria for matching edges of design space graphs, with values greater than 0 permitting matches based on part roles in addition to part IDs (for a detailed description, see Section 4). The levels of abstraction for the inputs and outputs of the *MERGE* operator, along with each tolerance of the *AND* operator are provided in Table 1.

Table 1. Input and Output Level of Abstraction (*AND* and *MERGE* Operators)

	AND_0	AND_1	AND_2	*MERGE*
inputs	concrete	concrete or abstract	concrete or abstract	concrete
output	concrete	concrete	concrete or abstract	concrete

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911

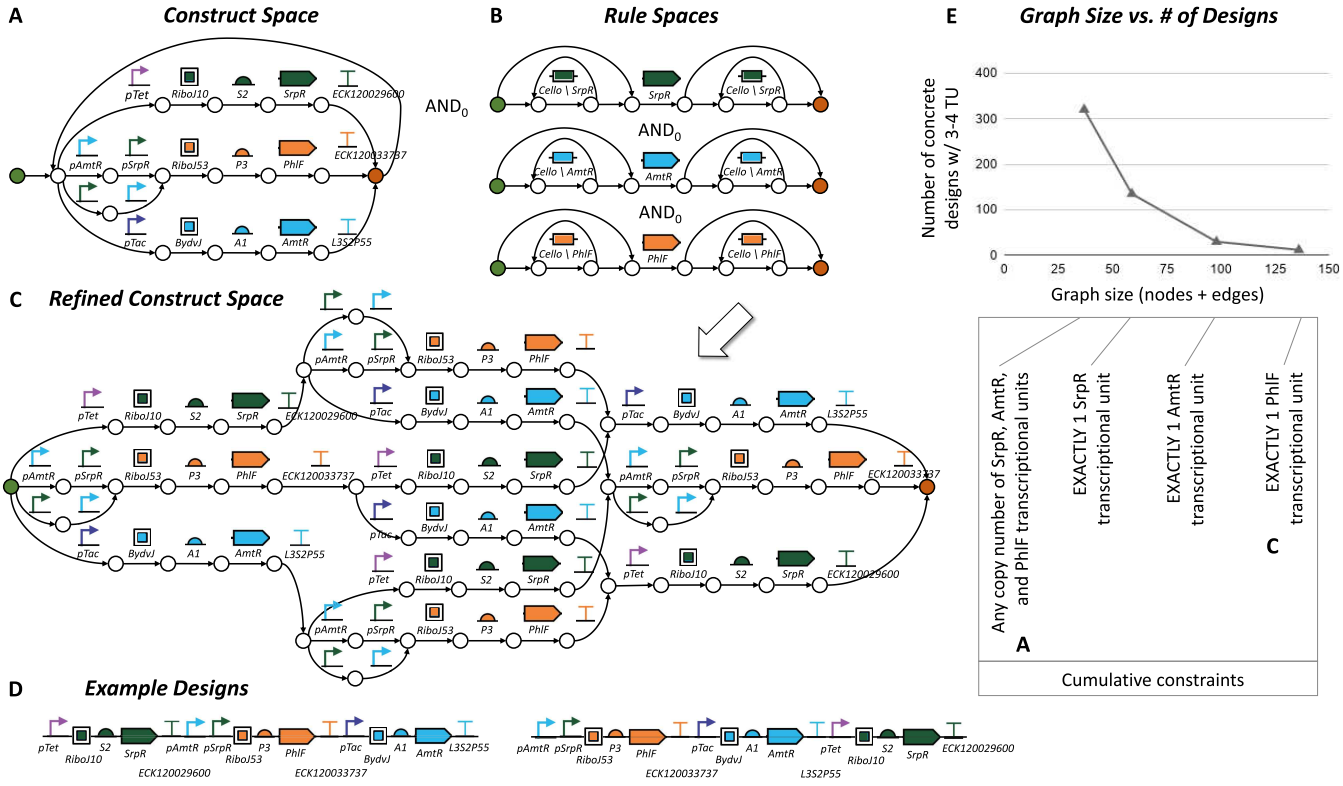


Figure 2. Application of the AND_0 operator to (A) the construct space for an AND circuit and (B) rule spaces for the CDS copy number. Each colored glyph represents a set of concrete parts with a role from the Sequence Ontology[48] as specified by the SBOL Visual 2 standard.[45] The exceptions are the glyphs labeled "Cello" followed by a slash and a part ID. Each one of these glyphs represents the set of all parts in the Cello library, except the part following the slash. (C) Result of applying the AND_0 operator to these design spaces is a construct space for the AND circuit in which each design contains exactly one copy of each TU. (D) Examples of AND circuit designs encoded by the final construct space. Each colored glyph in these designs represents a single concrete part rather than a set. (E) Line graph showing the relationship between graph size and the number of concrete designs encoded in the graph, as cumulative constraints are introduced via the application of the *AND* operator. The number of concrete designs containing 3−4 TUs is reported instead of the total number of concrete designs in order to better distinguish between graphs encoding infinitely many designs.

The ***AND*$_0$** operator takes two concrete design spaces as input and outputs a new concrete design space that encodes the intersection of their designs. The AND_0 operator is typically used to compare two construct spaces or to refine or validate a construct space against a rule space.

The ***AND*$_1$** operator takes two concrete or abstract design spaces as input and outputs a new concrete design space that encodes the intersection of their designs. The AND_1 operator is typically used to refine or validate a construct space against a rule space or to compose two construct spaces.

The ***AND*$_2$** operator takes two concrete or abstract design spaces as input and outputs a new concrete or abstract design space that encodes the intersection of their designs. The AND_2 operator is typically used to refine or validate a construct space with an architecture space or to compose two construct spaces. The AND_2 operator can also be used to compare two architecture spaces or validate a rule space against an architecture space.

The ***MERGE*** operator takes two concrete design spaces as input and outputs a new concrete design space that encodes a merger of their designs. The *MERGE* operator is typically used to infer novel designs from two structurally similar construct spaces.

*2.1.5. Applying**AND** to Refine/Validate Genetic Circuits.* Our first case study demonstrates the application of GOLDBAR's *AND* operator to a design-for-test workflow similar to that applied to the output of genetic circuit design tools such as Cello.[46] These tools typically assign the genetic logic gates needed to realize a target logic function but then require that their gate assignments be mapped to libraries of genetic circuit designs with different architectures (i.e., TU ordering and orientation) for compilation to linear DNA sequences for testing.[47] Designs for different circuit architectures often need to be generated since layout-dependent effects are often not captured by the models used by circuit design tools for gate assignment; hence, these effects must be determined empirically.

The purpose of this case study is to show how the *AND* operator is sufficiently general to be applied to all three main steps in the genetic circuit design-for-test workflow: design composition, refinement, and validation. Each of these steps makes use of the *AND* operator with a different level of tolerance. We will describe these steps in order of increasing tolerance of the *AND* operator used rather than the order in which these steps appear in the workflow. Thus, we will begin with design refinement using the AND_0 operator before continuing with design validation using the AND_1 operator. For design composition using the AND_2 operator, see Section A in the Supporting Information.

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911

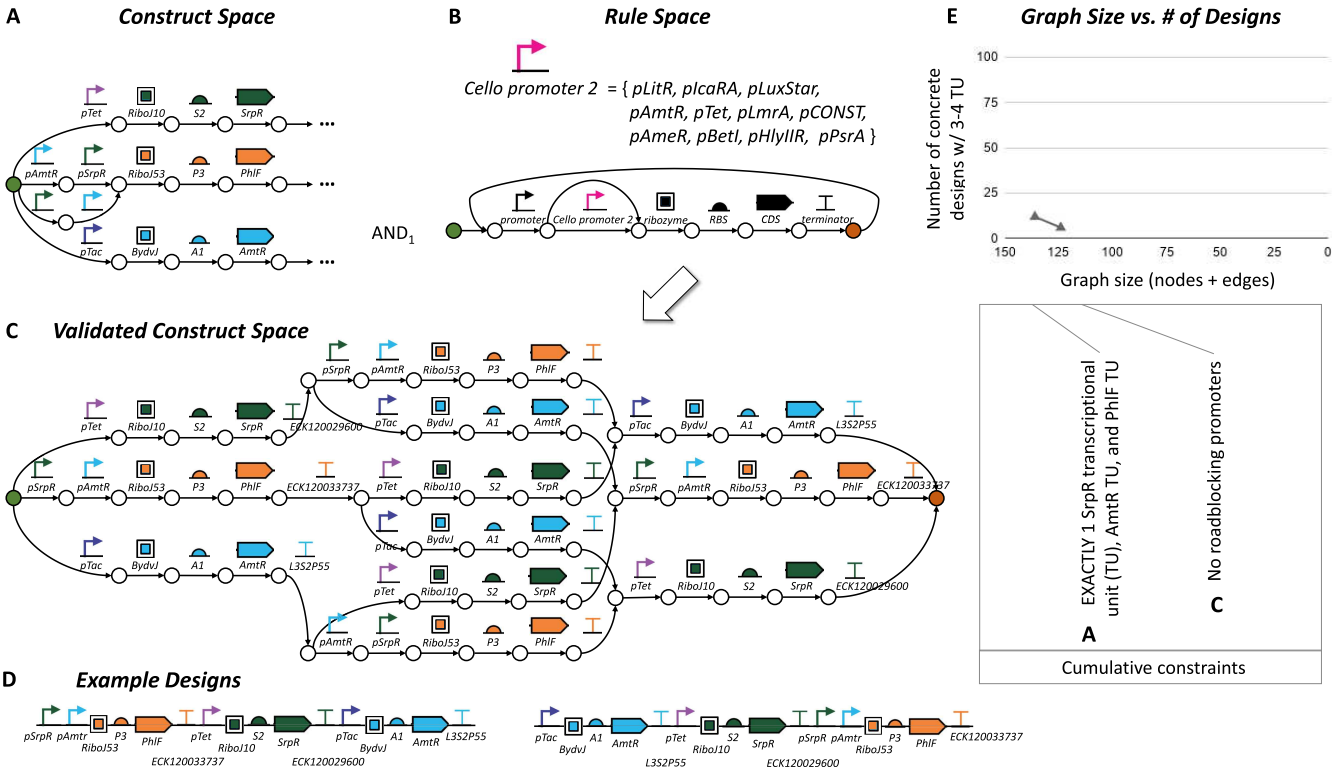


Figure 3. Application of the AND_1 operator to the (A) refined construct space for an AND circuit and (B) a rule space prohibiting promoter roadblocking. Each colored glyph represents a set of concrete parts with a role from the Sequence Ontology[48] as specified by the SBOL Visual 2 standard.[45] Each black glyph represents a set of a single abstract part. The pink promoter glyph labeled "Cello promoter 2" represents a set of concrete promoters that are identified to its right. (C) Result of applying the AND_1 operator to these design spaces is a construct space for the AND circuit in which designs positioning pSrpR as the second promoter in the PhlF TU have been omitted. (D) Examples of AND circuit designs encoded by the final construct space. Each colored glyph in these designs represents a single concrete part rather than a set. (E) Line graph showing the relationship between graph size and the number of concrete designs containing 3−4 TUs encoded in the graph, as cumulative constraints are introduced via the application of the *AND* operator. Since both the input and output construct spaces for the AND circuit in this example are finite, the number of concrete designs containing 3−4 TUs is equivalent to the total number of concrete designs encoded in the graph.

Figure 2 illustrates the refinement of the construct space for a Cello-designed AND circuit by applying the AND_0 operator to this space and a set of rule spaces to constrain the CDS copy number. (Note that the AND logic of the circuit has nothing to do with our use of the *AND* operator). The goal of this refinement is to produce a smaller construct space that is more feasible for testing but still captures interesting structural variation. The input circuit construct space is concrete and encodes every possible ordering and copy number for the TUs that make up the AND circuit. Each rule space is also concrete and fixes the copy number of a specific CDS in the AND circuit to exactly one (exactly one copy of *SrpR*, one copy of *AmtR*, and one copy of *PhlF*). Because all input design spaces are concrete, we applied the AND_0 operator. The output of this AND_0 operation is a concrete construct space that encodes only AND circuit designs containing exactly one copy of each TU in any order. Libraries of designs like this construct space are commonly used to determine the effect of TU order on design performance.

This example can be considered a refinement of the original construct space for the AND circuit since the number of encoded designs decreases from infinitely many to just $12 = 1 \times (2 \times 1 + 1 \times 2) + 2 \times (1 \times 1 + 1 \times 1) + 1 \times (1 \times 2 + 2 \times 1)$. Designs that have more TUs than the AND circuit, however, may have many more possible structural layouts (generally scaling as $N!$, where N is the number of TUs). Since the size of the graph encoding every possible ordering of single copy TUs also scales factorially, additional refinement via the *AND* operator is required for larger designs to disallow certain TU orderings and thus produce a computationally tractable construct space. This requirement is not inherently a disadvantage of the GOLDBAR framework since additional refinement is typically also desirable from the perspective of designing experiments. In particular, it is often not feasible or necessary to build and test all possible TU orderings in order to learn about the relationship between a design's structural layout and its performance. As we will discuss later on, while the expressive power of GOLDBAR could be increased to more compactly specify this type of design space, it would also become much more difficult to formally compare design spaces in general.

Next, Figure 3 demonstrates validation of the previously refined construct space for the AND circuit by using the AND_1 operator and a single rule space that disallows specific tandem promoter orderings. Similar to the previous refinement step, the goal of validation is to refine the construct space and remove designs that do not follow the rule encoded by the rule space. The difference is that, whereas the previous refinement step primarily omitted designs in order to obtain a finite, tractable design space for testing, the validation step seeks to

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911

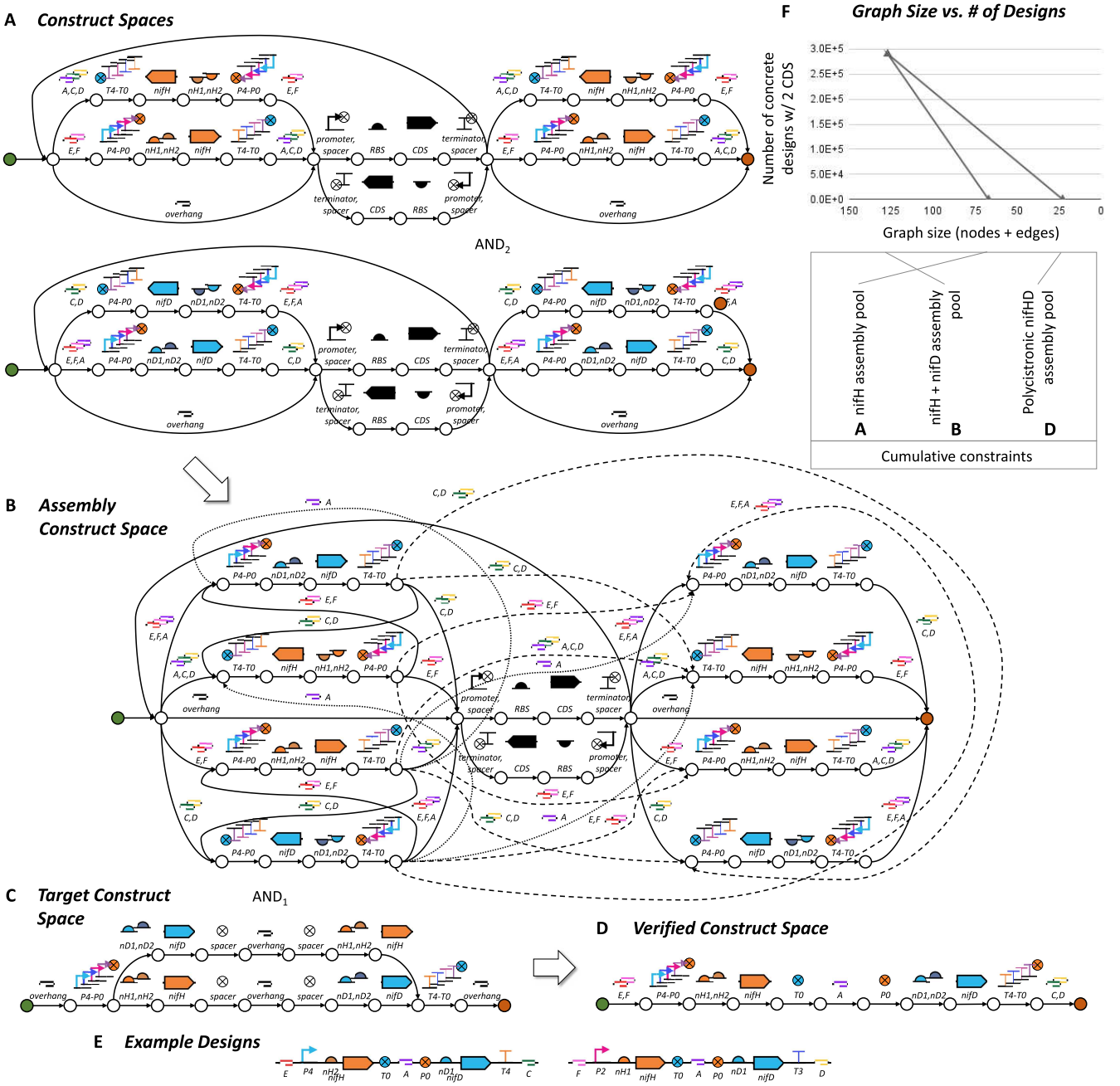


Figure 4. Application of the AND_2 operator to (A) the construct spaces for multiplexed pools of TU variants for *nifH* and *nifD*. (B) Result of applying the AND_2 operator to these construct spaces is to obtain the space of possible construct assemblies. (C) Target construct space for polycistronic TU variants of *nifHD* and *nifDH*. (D) Result of applying the AND_1 operator to the assembly space and target construct space to verify that the latter can be assembled from the former. (E) Examples of polycistronic TU variants for *nifHD* encoded by the final construct space. (F) Line graph showing the relationship between graph size and the number of concrete designs encoded in the graph, as cumulative constraints are introduced.

remove designs that are likely to fail based on knowledge gained via prior experimentation. In this example, the goal is to omit AND circuit designs that contain tandem promoter orderings known to cause "roadblocking," a phenomenon in which transcriptional repression of a TU's second promoter inhibits transcription from its first promoter.[42]

The rule space used in this example is partially abstract and allows one or more TUs that each contain a promoter, ribozyme, RBS, CDS, and terminator. If there is a second promoter in a TU, however, then it must be selected from a finite set of promoters that have been experimentally shown to not exhibit significant roadblocking. Note that this rule space could be concrete instead of abstract and specify exactly which parts are allowed in a TU (in which case the AND_0 operator could be used), but one benefit of being abstract is that this rule space can be reused to query or validate against future design spaces containing parts that did not exist when this rule space was specified.

The result of applying the AND_1 operator to the roadblocking rule space and the construct space for the fixed

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911

Inferring Novel Gene Clusters with MERGE

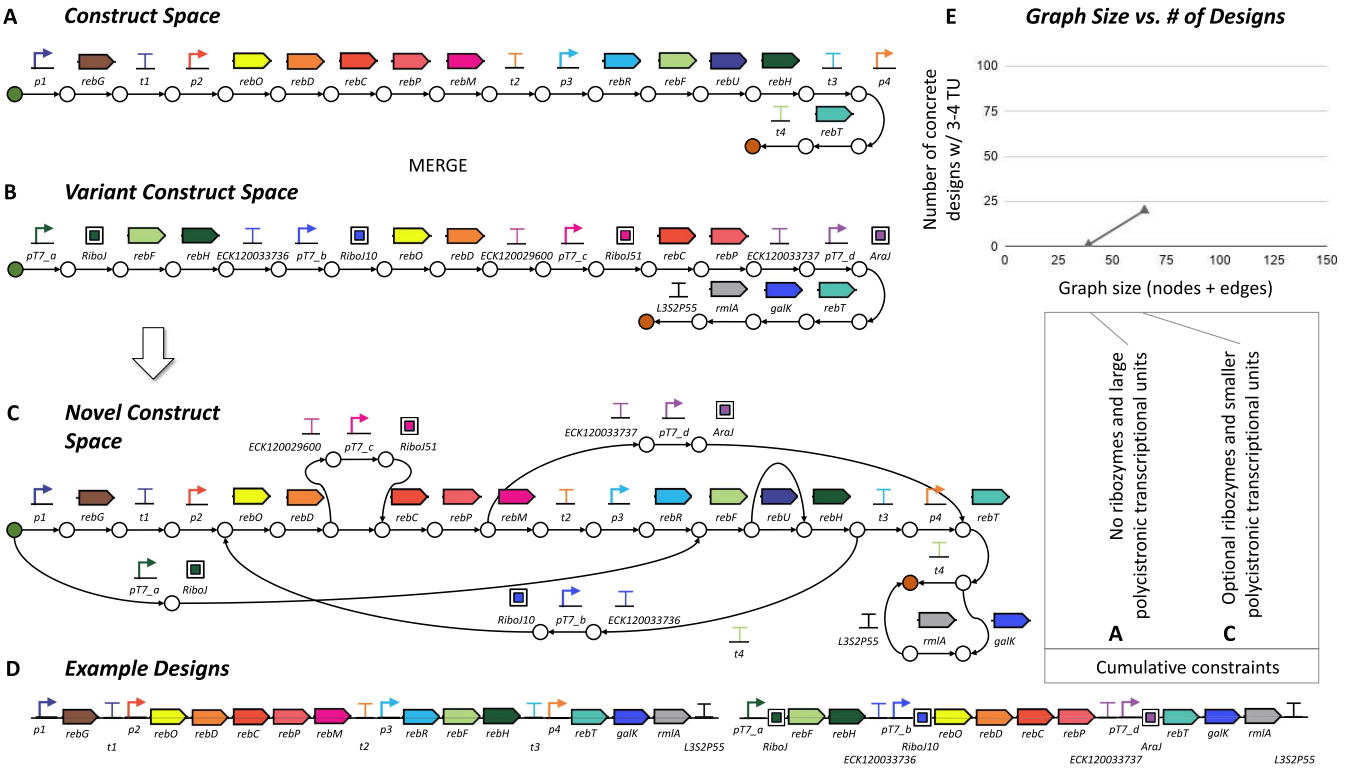


Figure 5. Application of the *MERGE* operator to (A) the construct space for a rebeccamycin gene cluster with natural operons and (B) the construct space for a rebeccamycin gene cluster with synthetic operons. (C) Result of applying the *MERGE* operator to these design spaces is a construct space that encodes designs for a rebeccamycin gene cluster with operons that exhibit both natural and synthetic characteristics. (D) Examples of gene cluster designs encoded by the final construct space. (E) Line graph showing the relationship between graph size and the number of concrete designs encoded in the graph, as cumulative constraints are introduced via the application of the *MERGE* operator.

copy number AND circuit produces a new construct space with just 6 designs instead of 12. This is because the pSrpR promoter in PhlF TU is roadblocking. By fixing the order of these promoters to prevent pSrpR from occurring in the second position, the application of AND_1 halves the total number of designs encoded by the construct space. Thus, the designer can be confident that their tested AND circuits will not include layouts that are more likely to exhibit poor performance due to known promoter roadblocking.

This concludes our demonstration of applying the *AND* operator to specify the design of a genetic AND circuit. To summarize, we have shown that the AND_0 operator can be used to refine a concrete construct space against rule spaces to constrain TU copy number for subsequent building and testing and that the AND_1 operator can be used to validate a concrete construct space against a rule space to remove designs likely to exhibit known promoter roadblocking.

2.1.6. Applying ***AND*** *to Verify nif Gene Cluster Assembly.* Our second case study demonstrates the application of GOLDBAR's *AND* operator to help verify the combinatorial assembly of a partial nitrogen fixation gene cluster. Combinatorial assembly is an important tool for implementing construct libraries to test and optimize the function of genetic designs with complex functions such as nitrogen fixation.[2]

The purpose of this use case is to show how the *AND* operator can be used to verify whether a target construct space or library can be combinatorially assembled from available parts. In doing so, we illustrate how GOLDBAR can be used to formally connect the design and build steps of synthetic biology workflows. Connecting design to build within a formal framework like GOLDBAR is important because it enables us to quickly generalize existing rules for combinatorial design and assembly to new synthetic biology parts with new functions, and it helps us ensure consistency in moving from the design step to the build step. In the future, as frameworks like GOLDBAR are extended with quantitative weights for different combinations of parts, we will also be able to verify whether a target construct space can be assembled from part combinations that have been empirically associated with a successful assembly.

Figure 4A,B illustrates the results of applying the AND_2 operator to the input construct spaces for the genes *nifH* and *nifD*. These input construct spaces are partially abstract and represent multiplexed pools of parts[27] (specifically, TU variants) that can be assembled in different orders and orientations. The only requirement for their assembly is that they have matching overhangs. Generally speaking, all levels of the *AND* operator's tolerance parameter can be used to match overhangs in this way. In this example, however, we apply the AND_2 operator because it also intersects the abstract portions of the input construct spaces with each other, thus preserving the ability of the output assembly space to be further assembled with construct spaces encoding new part pools.

After assembling the construct spaces for the *nifH* and *nifD* part pools, we apply the AND_1 operator to intersect the resulting assembly space (Figure 4B) with a target construct

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911

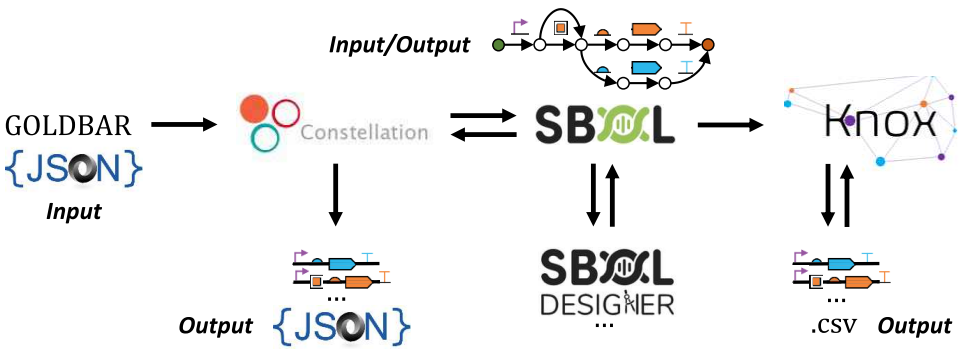


Figure 6. GOLDBAR software ecosystem. Constellation takes as input a GOLDBAR specification for a design space that is encoded by using JavaScript Object Notation (JSON). Alternatively, Constellation can take as input an SBOL specification for a design space. Constellation then produces as output a design space encoded using SBOL or a collection of designs encoded using JSON. The SBOL design spaces produced by Constellation can be imported by Knox and other SBOL-compatible tools that support combinatorial design, such as SBOL Designer.[52] Knox takes as input a SBOL specification for a design space and can output a collection of designs encoded using CSV. Knox can also take as input a SBOL or CSV specification for a collection of designs to convert to a design space.

space for possible polycistronic variants of *nifHD* (Figure 4C). This enables us to determine which polycistronic gene orderings can be assembled from available part pools. The reason that we apply the AND_1 operator is 2-fold. Number one, because all designs encoded by our input construct spaces are partially abstract, is that we must apply AND_1 or AND_2 in order to get any matches at all. Number two, by applying AND_1, we permit only concrete designs in the final output design space, which removes any designs that cannot be completely implemented via assembly from the available TU part pools.

In conclusion, while this case study focused on assembly via matching overhangs generated via Gibson or Golden Gate methods, GOLDBAR can be used to verify assembly via other methods, as well. Generally speaking, GOLDBAR should be applicable to any method in which assembly depends on matches between sequence overhangs or similar features, provided that these matches can be abstracted as matches between feature IDs or roles rather than feature nucleotide sequences.

2.1.7. Applying ***MERGE*** *to Infer Novel Biosynthetic Gene Clusters.* Our third case study demonstrates the application of GOLDBAR's *MERGE* operator to the construct spaces for a pair of related gene clusters for the biosynthesis of rebeccamycin, an antitumor antibiotic. The first of these construct spaces encodes a gene cluster based on natural operons found in *Saccharothrix aerocolonigenes*, while the second contains codon-optimized versions of the same genes organized into synthetic operons that are controlled with T7 promoters and insulated with ribozymes.[49,50]

The goal of merging the construct spaces for these gene clusters is to produce a new construct space that contains novel designs with motifs that exhibit characteristics of both clusters. In doing so, we can explore new architectures to produce the same target molecule (e.g., for optimizing biosynthesis) as well as architectures that can produce new molecules in the same chemical neighborhood. The *MERGE* operator accomplishes this task based on a fairly simple strategy: if two designs share a part, then whatever comes after that part in the first design can come after that part in the second design (and vice versa). Part of the power of the *MERGE* operator is that it applies this strategy to every design encoded by the input design spaces. In other words, the *MERGE* operator does not selectively apply this strategy due to bias or rules outside of those encoded by the input design spaces.

Figure 5 shows the result of the application of the *MERGE* operator to the construct spaces for the biosynthesis of rebeccamycin. Compared to these input construct spaces, the output construct space encodes infinitely many gene cluster designs instead of just two, and the designs it encodes exhibit greater diversity in terms of their architecture. Figure 5D shows two examples of gene cluster designs. The first design is for a gene cluster that is more similar to the input cluster with natural operons (the "natural" cluster) than the input cluster with synthetic operons (the "synthetic" cluster). In this case, the new gene cluster gained *rmlA* and *galK* (enzymes that help with the glycosylation of rebeccamycin in *Escherichia coli*) and lost *rebU* (a redundant rebeccamycin transporter). In other words, the *MERGE* operator effectively inferred some of the smallest changes needed to make the natural cluster more like the synthetic cluster. The second design is for a gene cluster that is more similar to the synthetic cluster since its operons are controlled with T7 promoters and insulated with ribozymes. In this case, however, the new gene cluster has a large *rebODCP* operon. That is, the *MERGE* operator has effectively inferred that splitting the *rebODCP* operon in two may be optional for synthetic operons.

Returning to the output construct space as a whole, the reason why this space encodes infinitely many designs is due to a property of the *MERGE* operator that emerges when input design spaces share at least two parts in different relative orders. For example, *rebO* comes before *rebF* in the natural cluster design, but it comes after *rebF* in the synthetic cluster design. Thus, the basic assumption of the *MERGE* operator produces a construct space that contains infinitely many gene cluster designs, each one with a different number of alternating copies of *rebO* and *rebF*. While this highly repetitive design motif can be desirable for some use cases (such as varying operator sites in synthetic promoters for CHO cell engineering[51]), for others, it can be removed via subsequent application of the *AND* operator and rule spaces to constrain part copy number. This use of the *AND* operator is the same as that in the previous use case of refining the design space for a genetic AND circuit in Section 2.1.5.

2.2. GOLDBAR Software Ecosystem. We have developed two complementary open-source software tools that implement GOLDBAR and together enable the specification,

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911

Table 2. Constellation and Knox Metrics

	graph construction		design enumeration	
use case	nodes + edges	runtime (ms)	designs	runtime (ms)
genetic circuit (AND_0) (Constellation)	227	1142	12	6
genetic circuit (AND_1) (Constellation)	215	13,275	6	1
nif cluster (AND_2 + AND_1) (Constellation)	26	443,596	400	6
biosynthetic cluster (*MERGE*) (Constellation)	77	476	10^3	92
genetic circuit (AND_0) (Knox)	136	807	12	616
genetic circuit (AND_1) (Knox)	124	277	6	731
nif cluster (AND_2 + AND_1) (Knox)	23	406	400	143
biosynthetic cluster (*MERGE*) (Knox)	65	157	10^3	386

storage, visualization, manipulation, and sampling of combinatorial biological design spaces: Constellation[4] and Knox[5]. Here, we describe these tools and the results of benchmarking them with the use cases presented in Section 2.1.

2.2.1. Constellation. Constellation is a design specification tool that can ingest both GOLDBAR and SBOL to produce equivalent design space graphs and enumerate designs from them. It is available as a web application[6] and a standalone Node.js package[7]. An example of a Constellation workflow is as follows: first, a user provides their GOLDBAR specification, part categories encoded using JavaScript Object Notation (JSON), and additional parameters for design enumeration (such as the number of designs and depth of cycles). Constellation then automatically converts this input to a design space graph and enumerates a list of designs obtained by traversing the graph. Constellation also generates an SBOL representation of the design space that can be downloaded as an XML file using the web application. This SBOL XML file can then be uploaded to Knox for storage and subsequent manipulation of its encoded design space, reuploaded to Constellation for design enumeration, or uploaded to another SBOL-compatible tool that supports the combinatorial design, such as SBOL Designer[52] (see Figure 6).

The Constellation API exposes three public functions: *sbol*, *goldbar*, and *symbolic*. The first two functions, as their names suggest, take as input a SBOL document or GOLDBAR specification, respectively. Both functions return a design space graph and a list of designs enumerated from this graph, and the *goldbar* function also returns an SBOL document. The *symbolic* function, on the other hand, only takes as input a GOLDBAR specification and returns only a list of enumerated designs. In order to parse the GOLDBAR specifications, Constellation relies on imparse[8] library.

Table 2 reports metrics for design space graph construction and design enumeration when Constellation is applied to the use cases in Section 2.1. For graph construction, runtime is weakly correlated with the output graph size (nodes plus edges). This is because some of the genetic circuit use cases involve the application of multiple graph product operators, and in these cases, the intermediate graphs generated by Constellation can be larger than the final output graph. In addition, because Constellation does not store intermediate graphs, the GOLDBAR specifications for the use cases that are later in the genetic circuit workflow must include the specifications for the use cases that are earlier in the workflow, which can cause their graph construction runtimes to be larger. For example, the runtime for the genetic circuit (AND_1) use case is larger than the runtime for the genetic circuit (AND_0) use case despite having a similar output graph size. This is because the specification for the AND_1 use case includes the specification for the AND_0 use case.

For design enumeration, runtime is mostly correlated with the number of designs enumerated. This is why the runtime for the *MERGE* use case (10^3 designs enumerated) is 1 to 2 orders of magnitude larger than the runtimes for the genetic circuit (AND_0) and genetic circuit (AND_1) use cases (12 and 6 designs enumerated, respectively). However, in general, the runtime for design enumeration can also be affected by the structure of the design space graph and the graph traversal algorithm used.

2.2.2. Knox. Knox is a database for storing and manipulating biological design spaces. Knox is available as a full-stack desktop application built on the Java Spring framework and Neo4j graph database platform. A modular web interface that exercises Knox's API is also included in the same Git repository. Lastly, a Docker container with the Knox application and a fresh database is also available for easy installation. An example of a Knox workflow is as follows: a user first imports SBOL specifications for design spaces generated with Constellation or another tool for combinatorial design into the database. The user can then enumerate designs from these spaces or apply GOLDBAR operators to these spaces and enumerate designs from the results.

The Knox API exposes a variety of functions for manipulating design space graphs stored in its database including one function per GOLDBAR operator. Each one of the GOLDBAR operator functions takes as input the IDs for two or more design space graphs in the database. The *AND* operator function also requires a tolerance parameter (an integer between 0 and 2, inclusive) that controls how these input design space graphs are intersected. Knox then produces an output design space graph that is the result of applying the corresponding GOLDBAR operator to the input design space graphs and stores the results in the database. Lastly, the Knox API also exposes functions for importing SBOL and CSV, exporting JSON for a D3[9] graph visualization, and enumerating designs from graphs.

Table 2 reports metrics for design space graph construction and design enumeration when Knox is applied to the use cases in Section 2.1. Again, graph construction runtime weakly correlates with output graph size since some of the *AND* use cases involve the application of multiple graph product operators, and the intermediate graphs can be larger than the final output graphs. Since Knox stores these intermediate graphs, its graph construction runtimes are faster than those for Constellation. Another reason for Knox's increased speed over Constellation for graph construction is that Knox omits unnecessary edges that Constellation instead attempts to remove during a postprocessing step.

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911

For design enumeration, Knox's runtimes are, on the whole, 1 to 2 orders of magnitude larger than Constellation's runtimes. This is primarily due to the fact that Knox must reload design spaces from Neo4j prior to design enumeration, whereas Constellation enumerates designs from graphs directly after their construction while they are still in memory.

3. DISCUSSION

We believe that frameworks for biological design should ideally permit specifications that are representative of synthetic biology, compact in size, and formally comparable in their specified structures. With respect to all three of these characteristics, it is natural to ask questions about the expressive power of a given design framework. For example, does the framework permit the specification of some design motifs and patterns but not others? Do these constraints impact the size of specifications within the framework or the ability to compare their specified structures? In this section, we discuss the characteristics of GOLDBAR's expressive power and connect them to the addition of GOLDBAR's *MERGE* operator and the extension of its *AND* operator. We also discuss several ways in which GOLDBAR could enable grammar-based machine learning in synthetic biology.

In the case of GOLDBAR, it can readily be shown that its expressive power with respect to formal language theory is limited to that of a regular grammar.[39] In practice, this means that GOLDBAR can be used to specify biological motifs with copy numbers that are variable and unbounded (Figure 2), but it cannot be used to specify relationships between these copy numbers. More specifically, GOLDBAR cannot be used to specify variable, unbounded palindromes or patterns like A^nB^n, where the variable, unbounded copy number of motif *A* is equal to that of motif *B*. Examples of biological design motifs that match the second pattern are those involving paired DNA parts such as recombination sites or a TF CDS and its cognate promoter.

Practically speaking, this limitation on the expressive power of GOLDBAR has a greater impact on the compactness of its specifications than it does on its representation of synthetic biology. Many synthetic biological systems do not have multiple copies of paired parts since systems containing these motifs can be prone to unwanted homologous recombination or cross-talk in regulation. If a designer still wants to specify such a system in GOLDBAR, then they can apply the *OR* operator to a set of design spaces in which paired parts have various copy numbers, starting with a copy number of zero or one and ending with the largest feasible copy number. The primary disadvantage of specifying paired-part relationships in this way is that it can be less concise than specifying them in a more expressive framework (e.g., one with expressive power equal to or greater than that of context-free grammars).

While a context-free grammar would be capable of representing unbounded, paired DNA parts, it has also previously been noted by Cai et al.[40] that there exist natural biological motifs such as overlapping genes, introns, and splicing sites that would require an even more expressive indexed grammar to represent them. These types of motifs are much less common in synthetic biology; however, overlapping features make it even more difficult to model the relationship between the structure and function of a design. Still, these features present a boundary in terms of what can be represented by using GOLDBAR.

The primary advantage of restricting the expressive power of GOLDBAR to that of a regular grammar is that doing so enables one to formally compare any two specifications written in GOLDBAR and determine whether they specify the same set of designs.[53] This is built upon the known result that comparing two regular grammars to determine whether they encode the same language is a decidable problem[54,55] (e.g., in contrast to the undecidable general problem of comparing two context-free grammars). This result is also critical for our definition of GOLDBAR's *AND* and *MERGE* operators since they are predicated on exhaustively matching between the structures of their input design space graphs. If GOLDBAR had greater expressive power, then its specified design space graphs could potentially be infinite in size, thus precluding their complete intersection or merger in a finite amount of time.

Ultimately, our decision to limit the expressive power of GOLDBAR is a trade-off between its representation of synthetic biology and the compactness and comparability of its design spaces. We have chosen a degree of expressiveness that maximizes the comparability of design spaces specified with GOLDBAR because we see a need in synthetic biology for frameworks that enable new designs to be inferred in an interpretable manner based on the structure of existing designs rather than deduced from first principles. This need is rooted in the observation that synthetic biology still largely lacks accurate mechanistic models to predict differences in the function of designs based on differences in their structural layout. We believe that our addition of a *MERGE* operator to GOLDBAR helps to satisfy this need by providing a means to exhaustively compare two design spaces and infer new designs based on their shared motifs.

One more aspect that can affect a design framework's representation of synthetic biology is its degree of abstraction, which has more than one possible interpretation in the context of biological design. For instance, degree of abstraction can refer to how abstract is an individual element's representation of the design. In other words, a DNA part can have both a sequence and a functional role or can just have a functional role if its precise sequence has not yet been defined. GOLDBAR supports this notion of abstraction by permitting the specification of parts with roles but no ID or DNA sequence. Our extension of GOLDBAR's *AND* operator enables design spaces containing abstract parts to be intersected, which in turn supports the definition of general rule spaces that can be used to validate or query future designs containing parts that do not yet exist (see Figure 3). Perhaps more importantly, this extension also permits the representation of combinatorial DNA assembly and design using a single formal framework that supports their unified comparison and validation. In other words, GOLDBAR can now be used to answer questions such as "what is the set of assembly-compatible designs that are structurally valid (and vice versa)?" (see Figure 4) without requiring integration across multiple formalisms and software tools.

Second, the degree of abstraction can also refer to a framework's ability to compactly encode multiple design elements as a single element, which can in turn be abstracted in a similar manner. For example, a promoter followed by a RBS, CDS, and terminator can be encoded as a TU, and a series of TUs can be encoded as a gene cluster. This notion of abstraction can have the practical effect of simplifying specifications for human interpretation and increasing the

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911

efficiency of their comparison when they contain the same composite parts. GOLDBAR currently does not directly support this degree of abstraction, but it can be indirectly implemented by applying the *OR* operator to specifications of the same design space at different levels of abstraction. Of course, this implementation increases rather than decreases the size of specifications written in GOLDBAR.

Going forward, GOLDBAR (and, more generally, the combinatorial design specification for synthetic biology) could move in several new directions. These include increasing expressive power beyond that of context-free grammars to enable the representation of structural relationships between paired and overlapping DNA parts. For GOLDBAR, this increase in expressive power must be balanced against the ability to define algorithmic (as opposed to heuristic) operators for intersecting and merging design spaces.

Another direction for GOLDBAR is the addition of numeric weights to the edges of its design space graphs. This extension would enable the application of stochastic grammar induction techniques to learn common biological design motifs, such as rules for genetic circuit design. A stochastic grammar induced in this way could be used to generate new designs or predict how likely a new design adheres to its encoded rules (and thus potentially exhibits similar functional properties to its encoded designs). These types of grammar-based models would also be interpretable, unlike black-box models based on neural networks. This is an important consideration for use cases in which genetic design via generative models may be subject to regulation, as interpretable models provide the best possible description of how a generated design was derived from a model's input. In addition, interpretable models can be more easily inspected for insight into the biological basis for high-performing designs. For example, merging design space graphs for genetic circuits that have been weighted according to a performance metric and inspecting the result can provide insight into which biological design motifs are unique to high-performing circuits.

Alternatively, weighted design space graphs could be used to generate better input features for deep learning to predict the functional performance of biological designs from their structure. These graphs could also serve as the structural basis for hybrid models such as neural network grammars, which have exhibited gains in learning efficiency and accuracy over traditional neural networks in the natural language domain.[56]

In these ways, a design framework such as GOLDBAR could facilitate grammar-based machine learning in synthetic biology. Before this can happen, however, new tools and software libraries are needed to tackle practical problems in combinatorial design and engineering for synthetic biology. With this work, we have demonstrated why we think GOLDBAR is a suitable formal foundation for developing these tools in terms of its expressive power (for representing designs across a range of applications), its operators for intersecting and merging design spaces (for comparing design motifs and generating new ones), and its compatibility with existing community standards such as SBOL (for enabling interoperability).

4. METHODS

The GOLDBAR framework and its associated software libraries and tools rely on a mathematically rigorous foundation (Supporting Information). This includes formal definitions of the syntax and semantics of GOLDBAR specifications (including detailed definitions of the AND and MERGE operator variants), as well as design space graph construction rules that are accompanied by soundness and completeness theorems. In particular, for a GOLDBAR specification e subject to certain constraints and a design space graph D constructed from e using the construction rules, it is proven that (1) every design permitted by specification e appears as a path in design space graph D and (2) every path in design space graph D is a design that is permitted by specification e. It is further demonstrated that these theorems do not apply when a specification e contains instances of the AND_1, AND_2, and *MERGE* operators, thus providing some guidance to users of the GOLDBAR framework and associated tools on how they should understand the capabilities and limitations of these operators for exploring new design spaces (e.g., using these operators on semantically equivalent but syntactically distinct specifications may lead to distinct design spaces).

■ ASSOCIATED CONTENT

Supporting Information

The Supporting Information is available free of charge at https://pubs.acs.org/doi/10.1021/acssynbio.4c00296.

An additional example of applying the AND_2 operator to compose the starting construct space for the AND circuit featured in Section 2.1.5; formal definitions of the syntax and semantics of GOLDBAR, as well as proofs concerning the equivalence of its different representations; a brief description of the GOLDBAR Generator software, which can be used to quickly create GOLDBAR specifications for design spaces that encode known principles of genetic design (PDF)

■ AUTHOR INFORMATION

Corresponding Author

Nicholas Roehner − *RTX BBN Technologies, Cambridge, Massachusetts 02138, United States;* orcid.org/0000-0003-4957-1552; Email: nicholas.roehner@rtx.com

Authors

James Roberts − *Biological Design Center, Boston University, Boston, Massachusetts 02215, United States; Department of Biomedical Engineering, Boston University, Boston, Massachusetts 02215, United States;* orcid.org/0000-0003-3462-5183

Andrei Lapets − *Reity, Boston, Massachusetts 02215, United States*

Dany Gould − *Hariri Institute for Computing, Boston University, Boston, Massachusetts 02215, United States*

Vidya Akavoor − *Hariri Institute for Computing, Boston University, Boston, Massachusetts 02215, United States*

Lucy Qin − *Hariri Institute for Computing, Boston University, Boston, Massachusetts 02215, United States*

D. Benjamin Gordon − *The Foundry, Cambridge, Massachusetts 02142, United States; Broad Institute of MIT and Harvard, Cambridge, Massachusetts 02142, United States; Synthetic Biology Center, Department of Biological Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, United States*

Christopher Voigt − *The Foundry, Cambridge, Massachusetts 02142, United States; Broad Institute of MIT and Harvard,*

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911

Cambridge, Massachusetts 02142, United States; Synthetic Biology Center, Department of Biological Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, United States; orcid.org/0000-0003-0844-4776

Douglas Densmore − *Biological Design Center, Boston University, Boston, Massachusetts 02215, United States; Department of Electrical and Computer Engineering, Boston University, Boston, Massachusetts 02215, United States;* orcid.org/0000-0002-7666-6808

Complete contact information is available at: https://pubs.acs.org/10.1021/acssynbio.4c00296

Author Contributions

N.R. designed the GOLDBAR framework and developed the Knox software. J.R. developed the GOLDBAR Generator software. A.L. led the development of the Constellation software and wrote formal definitions for the syntax and semantics of GOLDBAR. D.G., V.A., and L.Q. developed the Constellation software. D.B.G., C.V., and D.D. led the project. N.R., J.R., A.L., and D.D. wrote the manuscript.

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

We thank Swapnil Bhatia for developing the original design space formalism on which GOLDBAR is based. The authors are also grateful to Glenn Xavier, Shirene Cao, and Frederick Jansen from Boston University, and Evan Bowman, Erika Schwartz, and Melissa Garcia for their software development contributions. The development of Knox and Constellation was supported by DARPA Living Foundries award HR0011-15-C-0084 and the NIH Training Program in Quantitative Biology & Physiology (QBP) 1T32GM145455-01. 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 Defense Advanced Research Projects Agency (DARPA), the Department of Defense, the National Institutes of Health (NIH), or the United States Government. This document does not contain technology or technical data controlled under either the U.S. International Traffic in Arms Regulations or the U.S. Export Administration Regulations.

■ ADDITIONAL NOTES

[1]https://github.com/CIDARLAB/constellation-js
[2]https://github.com/CIDARLAB/knox
[3]https://github.com/CIDARLAB/goldbar-generator
[4]https://github.com/CIDARLAB/constellation-js
[5]https://github.com/CIDARLAB/knox
[6]www.constellationcad.org
[7]https://www.npmjs.com/package/constellation-js
[8]http://imparse.org/
[9]https://d3js.org/

■ REFERENCES

(1) Du, J.; Yuan, Y.; Tong, S.; Lian, J.; Zhao, H. Customized optimization of metabolic pathways by combinatorial transcriptional engineering. *Nucleic Acids Res.* **2012**, *40* (2), No. e142.

(2) Smanski, M. J.; Bhatia, S.; Zhao, D.; Park, Y. J.; Woodruff, L. B. A.; Giannoukos, G.; Ciulla, D.; Busby, M.; Calderon, J.; Nicol, R.; Gordon, D. B.; Densmore, D.; Voigt, C. A. Functional optimization of gene clusters by combinatorial design and assembly. *Nat. Biotechnol.* **2014**, *32*, 1241−1249.

(3) Jones, J. A.; Vernacchio, V. R.; Lachance, D. M.; Lebovich, M.; Fu, L.; Shirke, A. N.; Schultz, V. L.; Cress, B.; Linhardt, R. J.; Koffas, M. A. G. epathoptimize: A combinatorial approach for transcriptional balancing of metabolic pathways. *Sci. Rep.* **2015**, *5*, No. 11301.

(4) Freestone, T. S.; Zhao, H. Combinatorial pathway engineering for optimized production of the anti-malarial FR900098. *Biotechnol. Bioeng.* **2016**, *113* (2), 384−392.

(5) Coussement, P.; Bauwens, D.; Maertens, J.; Mey, M. D. Direct combinatorial pathway optimization. *ACS Synth. Biol.* **2017**, *6* (2), 224−232.

(6) Jeschek, M.; Gerngross, D.; Panke, S. Combinatorial pathway optimization for streamlined metabolic engineering. *Curr. Opin. Biotechnol.* **2017**, *47*, 142−151.

(7) Carbonell, P.; Jervis, A. J.; Robinson, C. J.; Yan, C.; Dunstan, M.; Swainston, N.; Hollywood, K. A.; Vinaixa, M.; Currin, A.; Rattray, N. J. W.; Taylor, S.; Spiess, R.; Sung, R.; Williams, A. R.; Fellows, D.; Stanford, N. J.; Mulherin, P.; Feuvre, R. L.; Barran, P.; Goodacre, R.; Turner, N. J.; Goble, C.; Chen, G. G.; Kell, D. B.; Micklefield, J.; Breitling, R.; Takano, E.; Faulon, J.-L.; Scrutton, N. S. An automated design-build-test-learn pipeline for enhanced microbial production of fine chemicals. *Commun. Biol.* **2018**, *1*, No. 66.

(8) Naseri, G.; Behrend, J.; Rieper, L.; Mueller-Roeber, B. Compass for rapid combinatorial optimization of biochemical pathways based on artificial transcription factors. *Nat. Commun.* **2019**, *10*, No. 2615.

(9) Taylor, G. M.; Heap, J. T. Design and Implementation of Multi-Protein Expression Constructs and Combinatorial Libraries Using Start-Stop Assembly. In *DNA Cloning and Assembly of Methods in Molecular Biology*; Chandran, S.; George, K., Eds.; Humana: New York, NY, 2020; Vol. *2205*.

(10) Pfleger, B. F.; Pitera, D. J.; Smolke, C. D.; Keasling, J. D. Combinatorial engineering of intergenic regions in operons tunes expression of multiple genes. *Nat. Biotechnol.* **2006**, *24*, 1027−1032.

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

(12) Chen, Y.-J.; Liu, P.; Nielsen, A. A. K.; Brophy, J. A. N.; Clancy, K.; Peterson, T.; Voigt, C. A. Characterization of 582 natural and synthetic terminators and quantification of their design constraints. *Nat. Methods* **2013**, *10*, 659−664.

(13) Curran, K. A.; Morse, N. J.; Markham, K. A.; Wagman, A. M.; Gupta, A.; Alper, H. S. Short synthetic terminators for improved heterologous gene expression in yeast. *ACS Synth. Biol.* **2015**, *4*, 824−832.

(14) Redden, H.; Hal, S. A. The development and characterization of synthetic minimal yeast promoters. *Nat. Commun.* **2015**, *6*, No. 7810.

(15) Guo, Y.; Dong, J.; Zhou, T.; Auxillos, J.; Li, T.; Zhang, W.; Wang, L.; Shen, Y.; Luo, Y.; Zheng, Y.; Lin, J.; Chen, G.-Q.; Wu, Q.; Cai, Y.; Dai, J. YeastFab: the design and construction of standard biological parts for metabolic engineering in Saccharomyces cerevisiae. *Nucleic Acids Res.* **2015**, *43* (13), No. e88.

(16) Lee, M. E.; DeLoache, W. C.; Cervantes, B.; Dueber, J. E. A highly characterized yeast toolkit for modular, multipart assembly. *ACS Synth. Biol.* **2015**, *4* (10), 975−986.

(17) Moore, S. J.; Lai, H.-E.; Kelwick, R. J. R.; Chee, S. M.; Bell, D. J.; Polizzi, K. M.; Freemont, P. S. EcoFlex: A multifunctional MoClo kit for *E. coli* synthetic biology. *ACS Synth. Biol.* **2016**, *5* (10), 1059−1069.

(18) Iverson, S. V.; Haddock, T. L.; Beal, J.; Densmore, D. M. CIDAR MoClo: Improved MoClo assembly standard and new *E. coli* part library enable rapid combinatorial design for synthetic and traditional biology. *ACS Synth. Biol.* **2016**, *5* (10), 99−103.

(19) Zong, Y.; Zhang, H. M.; Lyu, C.; Ji, X.; Hou, J.; Guo, X.; Ouyang, Q.; Lou, C. Insulated transcriptional elements enable precise design of genetic circuits. *Nat. Commun.* **2017**, *8*, No. 52.

(20) Naseri, G.; Balazadeh, S.; Machens, F.; Kamranfar, I.; Messerschmidt, K.; Mueller-Roeber, B. Plant-derived transcription

factors for orthologous regulation of gene expression in the yeast Saccharomyces cerevisiae. *ACS Synth. Biol.* **2017**, *6* (9), 1742−1756.
(21) Machens, F.; Balazadeh, S.; Mueller-Roeber, B.; Messerschmidt, K. Synthetic promoters and transcription factors for heterologous protein expression in Saccharomyces cerevisiae. *Front. Bioeng. Biotechnol.* **2017**, *5*, No. 63.
(22) Dossani, Z. Y.; Apel, A. R.; Szmidt-Middleton, H.; Hillson, N. J.; Deutsch, S.; Keasling, J. D.; Mukhopadhyay, A. A combinatorial approach to synthetic transcription factor-promoter combinations for yeast strain engineering. *Yeast* **2018**, *35* (3), 273−280.
(23) Werner, S.; Engler, C.; Weber, E.; Gruetzner, R.; Marillonnet, S. Fast track assembly of multigene constructs using golden gate cloning and the MoClo system. *Bioengineered* **2012**, *3* (1), 38−43.
(24) Coussement, P.; Maertens, J.; Beauprez, J.; Van Bellegem, W.; De Mey, M. One step dna assembly for combinatorial metabolic engineering. *Metab. Eng.* **2014**, *23*, 70−77.
(25) Zhang, S.; Zhao, X.; Tao, Y.; Lou, C. A novel approach for metabolic pathway optimization: Oligo-linker mediated assembly (OLMA) method. *J. Biol. Eng.* **2015**, *9*, No. 23.
(26) Mitchell, L. A.; Chuang, J.; Agmon, N.; Khunsriraksakul, C.; Phillips, N. A.; Cai, Y.; Truong, D. M.; Veerakumar, A.; Wang, Y.; Mayorga, M.; Blomquist, P.; Sadda, P.; Trueheart, J.; Boeke, J. D. Versatile genetic assembly system (VEGAS) to assemble pathways for expression in S. cerevisiae. *Nucleic Acids Res.* **2015**, *43* (13), 6620−6630.
(27) Woodruff, L. B. A.; Gorochowski, T. E.; Roehner, N.; Mikkelsen, T. S.; Gordon, D. B.; Densmore, D.; Nicol, R.; Voigt, C. A. Registry in a tube: Multiplexed pools of retrievable parts for genetic design space exploration. *Nucleic Acids Res.* **2016**, *45* (3), 1553−1565.
(28) Yuan, Y.; Andersen, E.; Zhao, H. Flexible and versatile strategy for the construction of large biochemical pathways. *ACS Synth. Biol.* **2016**, *5* (1), 46−52.
(29) Jin, P.; Ding, W.; Du, G.; Chen, J.; Kang, Z. DATEL: a scarless and sequence-independent DNA assembly method using thermostable exonucleases and ligase. *ACS Synth. Biol.* **2016**, *5* (9), 1028−1032.
(30) Basitta, P.; Westrich, L.; Rösch, M.; Kulik, A.; Gust, B.; Apel, A. K. AGOS: a plug-and-play method for the assembly of artificial gene operons into functional biosynthetic gene clusters. *ACS Synth. Biol.* **2017**, *6* (5), 817−825.
(31) van Dolleweerd, C. J.; Kessans, S. A.; Van de Bittner, K. C.; Bustamante, L. Y.; Bundela, R.; Scott, B.; Nicholson, M. J.; Parker, L. Y. MIDAS: A modular dna assembly system for synthetic biology. *ACS Synth. Biol.* **2018**, *7* (4), 1018−1029.
(32) Wu, Y.; Zhu, R.-Y.; Mitchell, L. A.; Ma, L.; Liu, R.; Zhao, M.; Jia, B.; Xu, H.; Yang, Y.-X.; Li, Z.-M.; Li, Y.; Ma, X.; Liu, H.; Liu, D.; Xiao, W.-H.; Zhou, X.; Li, B.-Z.; Yuan, Y.-J.; Boeke, J. D. In vitro DNA SCRaMbLE. *Nat. Commun.* **2018**, *9*, No. 1935.
(33) Kang, Z.; Ding, W.; Jin, P.; Du, G.; Chen, J. DNA assembly with the DATEL method. In *Synthetic Biology: Methods and Protocols*; Braman, J. C., Ed.; Methods in Molecular Biology; Humana Press: New York, NY, 2018; Vol. *1772*.
(34) Taylor, G. M.; Mordaka, P. M.; Heap, J. T. Start-Stop Assembly: a functionally scarless DNA assembly system optimized for metabolic engineering. *Nucleic Acids Res.* **2019**, *47* (3), No. e17.
(35) Ma, X.; Liang, H.; Cui, X.; Liu, Y.; Lu, H.; Ning, W.; Poon, N. Y.; Ho, B.; Zhou, K. A standard for near-scarless plasmid construction using reusable DNA parts. *Nat. Commun.* **2019**, *10*, No. 3294.
(36) Liu, S.; Xiao, H.; Zhang, F.; Lu, Z.; Zhang, Y.; Deng, A.; Li, Z.; Yang; Wen, T. A seamless and iterative DNA assembly method named PS-Brick and its assisted metabolic engineering for threonine and 1-propanol production. *Biotechnol. Biofuels* **2019**, *12*, No. 180.
(37) Fero, M. J.; Craft, J. K.; Vu, T.; Hillson, N. J. Combinatorial-Hierarchical DNA Library Design Using the Teselagen DESIGN Module with j5. In *DNA Cloning and Assembly*; Chandran, S.; George, K., Eds.; Humana: New York, NY, 2020; Vol. *2205*.
(38) Bilitchenko, L.; Liu, A.; Cheung, S.; Weeding, E.; Xia, B.; Leguia, M.; Anderson, J. C.; Densmore, D. Eugene - a domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PLoS One* **2011**, *6* (4), No. e18882.
(39) Bhatia, S. P.; Smanski, M. J.; Voigt, C. A.; Densmore, D. M. Genetic design via combinatorial constraint specification. *ACS Synth. Biol.* **2017**, *6* (11), 2130−2135.
(40) Cai, Y.; Hartnett, B.; Gustafsson, C.; Peccoud, J. A syntactic model to design and verify synthetic genetic constructs derived from standard biological parts. *Bioinformatics* **2007**, *23* (20), 2760−2767.
(41) Decoene, T.; Paepe, B. D.; Maertens, J.; Coussement, P.; Peters, G.; Maeseneire, S. L. D.; Mey, M. D. Standardization in synthetic biology: An engineering discipline coming of age. *Crit. Rev. Biotechnol.* **2018**, *38* (5), 647−656.
(42) Brophy, J. A. N.; Voigt, C. A. Principles of genetic circuit design. *Nat. Methods* **2014**, *11* (5), 508−520.
(43) Roehner, N.; Bartley, B.; Beal, J.; McLaughlin, J.; Pocock, M.; Zhang, M.; Zundel, Z.; Myers, C. J. Specifying combinatorial designs with the Synthetic Biology Open L anguage (SBOL). *ACS Synth. Biol.* **2019**, *8* (7), 1519−1523.
(44) Roehner, N.; Beal, J.; Clancy, K.; Bartley, B.; Misirli, G.; Grünberg, R.; Oberortner, E.; Pocock, M.; Bissell, M.; Madsen, C.; Nguyen, T.; Zhang, M.; Zhang, Z.; Zundel, Z.; Densmore, D.; Gennari, J. H.; Wipat, A.; Sauro, H. M.; Myers, C. J. Sharing structure and function in biological design with SBOL 2.0. *ACS Synth. Biol.* **2016**, *5*, 498−506.
(45) Beal, J.; Nguyen, T.; Gorochowski, T. E.; Goni-Moreno, A.; Scott-Brown, J.; Madsen, C.; McLaughlin, J. A.; Aleritsch, B.; Bartley, B.; Bhakta, S.; Bissell, M.; Clancy, K.; Castillo Hair, S.; Luna, A.; Novere, N. L.; Palchick, Z.; Pocock, M.; Sauro, H.; Sexton, J. T.; Tabor, J. J.; Voigt, C. A.; Zundel, Z.; Myers, C.; Wipat, A. Communicating structure and function in synthetic biology diagrams. *ACS Synth. Biol.* **2019**, *8* (8), 1818−1825.
(46) Nielsen, A. A. K.; Der, B.; Shin, J.; Vaidyanathan, P.; Paralanov, V.; Strychalski, E. A.; Ross, D.; Densmore, D.; Voigt, C. A. Genetic circuit design automation. *Science* **2016**, *352* (6261), No. aac7341.
(47) Vaidyanathan, P.; Der, B. S.; Bhatia, S.; Roehner, N.; Silva, R.; Voigt, C. A.; Densmore, D. A framework for genetic logic synthesis. *Proc. IEEE* **2015**, *103* (11), 2196−2207.
(48) Eilbeck, K.; Eilbeck, S. E.; Mungall, C. J.; Yandell, M.; Stein, L.; Durbin, R.; Ashburner, M. The Sequence Ontology: A tool for the unification of genome annotations. *Genome Biol.* **2005**, *6*, No. R44.
(49) Sánchez, C.; Butovich, I. A.; Braña, A. F.; Rohr, J.; Mendéz, C.; Sálas, J. A. The biosynthetic gene cluster for the antitumor rebeccamycin: Characterization and generation of indolocarbazole derivatives. *Chem. Biol.* **2018**, *9*, 519−531.
(50) Casini, A.; Chang, F. Y.; Eluere, R.; et al. A pressure test to make 10 molecules in 90 days: External evaluation of methods to engineer biology. *J. Am. Chem. Soc.* **2018**, *140*, 4302−4316.
(51) Brown, A. J.; Sweeney, B.; Mainwaring, D. O.; James, D. C. Synthetic promoters for CHO cell engineering. *Biotechnol. Bioeng.* **2014**, *111*, 1638−1647.
(52) Zhang, M.; McLaughlin, J. A.; Wipat, A.; Myers, C. J. SBOLDesigner 2: An intuitive tool for structural genetic design. *ACS Synth. Biol.* **2017**, *6*, 1150−1160.
(53) Almeida, M.; Moreira, N.; Reis, R. In *Testing the Equivalence of Regular Languages*, Proceedings Eleventh International Workshop on Descriptional Complexity of Formal Systems, DCFS 2009, Magdeburg, Germany, July 6−9, 2009, volume 3 of EPTCS; Dassow, J.; Pighizzini, G.; Truthe, B., Eds.; 2009; pp 47−57.
(54) Hunt, H. B.; Rosenkrantz, D. J.; Szymanski, T. G. On the equivalence, containment, and covering problems for the regular and context-free languages. *J. Comput. Syst. Sci.* **1976**, *12* (2), 222−268.
(55) Hopcroft, J. E.; Motwani, R.; Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*, 3rd ed.; Addison-Wesley Longman Publishing Co., Inc.: USA, 2006.
(56) Dyer, C.; Kuncoro, A.; Ballesteros, M.; Smith, N. A. Recurrent neural network grammars **2016**, arXiv:1602.07776. arXiv:physics/0402096. https://arxiv.org/abs/1602.07776.

https://doi.org/10.1021/acssynbio.4c00296
ACS Synth. Biol. 2024, 13, 2899−2911