DNA as Features: Organic Software Product Lines

Mikaela Cashman
Computer Science
Iowa State University
Ames, IA, USA
mcashman@iastate.edu

Justin Firestone
Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE, USA
jfiresto@cse.unl.edu

Myra B. Cohen
Computer Science
Iowa State University
Ames, IA, USA
mcohen@iastate.edu

Thammasak Thianniwet
School of Information Technology
Suranaree University of Technology
Nakhon Ratchasima, Thailand
thammasak@sut.ac.th

Wei Niu
Chem. & Biomolecular Engineering
University of Nebraska-Lincoln
Lincoln, NE, USA
wniu2@unl.edu

ABSTRACT

Software product line engineering is a best practice for managing reuse in families of software systems. In this work, we explore the use of product line engineering in the emerging programming domain of synthetic biology. In synthetic biology, living organisms are programmed to perform new functions or improve existing functions. These programs are designed and constructed using small building blocks made out of DNA. We conjecture that there are families of products that consist of common and variable DNA parts, and we can leverage product line engineering to help synthetic biologists build, evolve, and reuse these programs. As a first step towards this goal, we perform a domain engineering case study that leverages an open-source repository of more than 45,000 reusable DNA parts. We are able to identify features and their related artifacts, all of which can be composed to make different programs. We demonstrate that we can successfully build feature models representing families for two commonly engineered functions. We then analyze an existing synthetic biology case study and demonstrate how product line engineering can be beneficial in this domain.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; • **Applied computing → Systems biology**.

KEYWORDS

Software Product Lines, BioBricks, Synthetic Biology

ACM Reference Format:
Mikaela Cashman, Justin Firestone, Myra B. Cohen, Thammasak Thianniwet, and Wei Niu. 2019. DNA as Features: Organic Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3336294.3336298

SPLC '19, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-7138-4/19/09…$15.00
https://doi.org/10.1145/3336294.3336298

1 INTRODUCTION

Today's software development practices are dominated by reusability, with practitioners sharing plug-and-play modules that can be applied in a multitude of different applications. As part of this movement, open-source repositories such as GitHub have emerged as marketplaces where developers find libraries and other reusable components. At the same time, software product line (SPL) engineering has become a best practice for modeling and managing families of software systems. The SPL community has turned to open-source systems and used the concepts of commonality and variability to define open-source product lines [31]. Recently, Montalvillo and Díaz have proposed techniques to aid SPL practices within GitHub [25]. It seems natural to look at other emerging open-source marketplaces to understand whether SPL development can provide benefits to those communities as well.

Synthetic biology, the practice of engineering living organisms by modifying their DNA, has advanced rapidly over the last 30 years [8]. It is being used for sensing heavy metals for pollution mitigation [5], development of synthetic biofuels [40], engineering cells to communicate and produce bodily tissues [29], emerging medical applications [22, 38], and basic computational purposes [12]. Synthetic biologists design new functionality, encode this in DNA strands, and insert the new DNA *part* into a living organism such as the common K-12 strain of the bacteria *Escherichia coli (E. coli)*. As the organism reproduces, it replicates the new DNA along with its native code and builds proteins that perform the encoded functionality. In essence, the biochemist is *programming* the organism to behave in a new way. Hence we call these *organic programs*.

As DNA strands have become easy to engineer by simply purchasing a desired sequence, the field of synthetic biology has rapidly grown. For instance, each year 300+ teams of students (high school through graduate) compete in the International Genetically Engineered Machine (iGEM) Competition. Teams build genetically engineered systems to solve real-world problems [20]. Students are required to submit the engineered parts along with their designs and experimental results back into an open-source collection of DNA parts called *BioBricks*. This BioBrick repository (called The Registry of Standard Parts) [21] contains over 45,000 DNA parts and can be viewed as a Git repository for DNA. In this paper we utilize this as our exemplar system, but we note that companies

and other institutions are likely building their own commercial instances of this type of repository.

DNA parts are often advertised as "LEGO" pieces that can be combined in many ways to form new genetic devices. However, these LEGO pieces come with no "Building Instructions." An engineer begins a project by developing a blueprint for the organic program they want to build. They will have a general plan for the type of features they want their system to have (such as a part that produces a fluorescent protein or expresses a gene in the presence of a particular chemical). To bring this creation to fruition they must find the corresponding parts in a repository or in the literature. They then build the associated working organic system following an architecture that merges the features together. However, this architecture can require significant domain knowledge to develop. Rather than expecting engineers to create architectures from scratch, we hypothesize that SPL engineering can be used.

In this work we explore the use of product line engineering with the goal of helping developers in synthetic biology. We conjecture that there are families of products that consist of common and variable DNA parts, just as we see in other open-source repositories. We call these *organic software product lines*. We conduct a case study to evaluate this claim and lay the foundations for merging software product line engineering and synthetic biology.

The contributions of this work are:

(1) A mapping of SPL engineering to the domain of synthetic biology resulting in organic software product lines;
(2) A case study demonstrating the potential reuse and existence of both commonality and variability in the BioBricks repository and showing that we can build feature models that have potential to help synthetic biologists.

In the next section we present background on synthetic biology and a motivating example. We then propose the notion of an organic SPL (Section 3). We follow this with our case study (Section 4), results (Section 5), and discussion (Section 6). We present related work in Section 7, and end with conclusions and future work.

2 BACKGROUND

Synthetic biology has been defined as a process "to design new, or modify existing, organisms to produce biological systems with new or enhanced functionality according to quantifiable design criteria" [1]. Part of this definition emphasizes design and quantification. It follows that we can view synthetic biology as a programming discipline. It begins with a model, which is then implemented into strands of DNA that are inserted into a living cell. We can view the organism as the compiler that takes the DNA and translates it to machine level code, creating proteins that the organism uses to perform different functions. Just as machine code is written in 1s and 0s, biology is written in the four DNA bases adenine (A), thymine (T), cytosine (C), and guanine (G).

This analogy of *programming biology* is not a new concept. There is even a programming language called the Synthetic Biology Open Language (SBOL) which defines a common way to represent biological designs [30]. Researchers have used synthetic biology to create a context-free grammar using BioBricks [7], automated design of genetic circuits with NOT/NOR gates [26], and bacterial networks to use DNA for data storage [6, 32]. There are also several examples of organic programs inspired by classic computer science constructs such as a genetic oscillator [13], a genetic toggle switch [17], and a time-delay circuit [39].

At its core, synthetic biology breaks down a biological process into smaller functions, each of which can be represented by a DNA *part*. These parts can be put together in various combinations to make new functions. The largest open-source repository of DNA parts is called the Registry of Standard Parts [21] (Registry for short). Parts have been contributed to this registry through the iGEM Competition. Although participants from iGEM are the most frequently documented users of the repository, anyone can use it to find appropriate parts for their designs.

iGEM describes itself as a competition where teams "design, build, test, and measure a system of their own design using interchangeable biological parts and standard molecular biology techniques." Each year about 6,000 students participate, designing projects which often address various regional problems such as pollution mitigation. Teams are judged by community experts and can be awarded a medal (bronze, silver, and gold) corresponding to the impact and contributions of their project. A gold medal team must achieve several goals such as modeling their project, demonstrating their work through experimentation, collaborating with other teams, addressing safety concerns, improving pre-existing parts or projects, and contributing new parts.

2.1 Motivating Example

We next present a motivating example derived from our case study demonstrating the potential of SPL engineering in this domain.

Cell-to-cell signaling is a common function of synthetic biology. It represents a key communication mechanism for cellular organisms. A *sender* organism communicates with a *receiver* organism which responds to the signal by emitting some chemical *reporter*. Suppose an engineer wants to build a cell-to-cell signaling system from scratch. If the engineer has no additional resources other than an online repository and his or her knowledge of synthetic biology, then this analogy is similar to someone searching GitHub for code that performs a particular function. The user can search the Registry for "signalling" and they will be redirected to a single page with a list of parts. It consists of eight senders, 11 receivers, and 464 "other" parts. If, however, the user searches with U.S. English convention for "signaling" the query returns 789 hits, each with its own page to investigate. It is important to note that not all of these page hits link to parts actually involved in cell-to-cell signaling. Some are pages that simply have the word "signaling" in them. This demonstrates the difficulty of any free-text search.

An alternate strategy is to search based on function using the field "Browse parts and devices by function." Figure 1, steps #1 and #2, show this in the Registry. There are 10 functions listed including "Cell-to-cell signaling and quorum sensing." Parts are sorted (Figure 1 - step #3) into various lower-level categories on this page. Most are basic parts that include: 39 promoters, 13 transcriptional regulators, 12 enzymes, and 21 translational units. There is also a separate list of 138 composite (or aggregate) parts.

Another strategy would be to search for previous projects that built a cell-to-cell signaling system. For example, one might locate

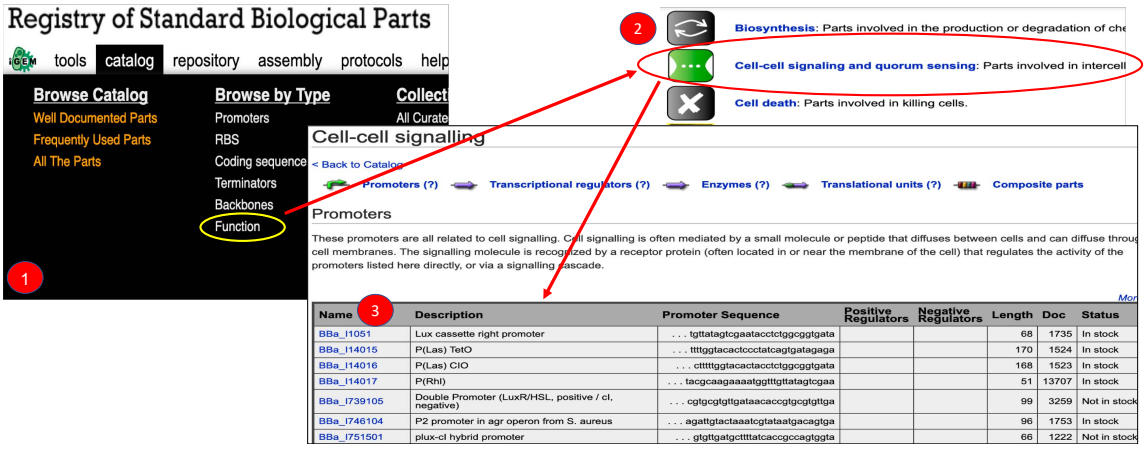

Figure 1: Browse by function → Cell-to-cell signaling

the 2017 iGEM team from Arizona State University (ASU) [2]. Looking on this team's web page would lead the user to find models for 30 composite parts for cell-to-cell signaling. While this is an improvement over the prior approaches and provides a roadmap to build the system (along with results of the study), it is limited to the 30 products that the ASU team chose to use in their experiments. As we will show, there are many more ways to build a cell-to-cell signaling system.

What if, instead of starting from scratch, the synthetic biologist begins this process with the feature model shown in Figure 2 (a subset of a feature model in our study)? From this model the user immediately can see the architecture of their system. First, they learn that any cell-to-cell signaling system requires three basic parts: a sender, a receiver, and a reporter. Instead of having to look at hundreds of possible parts, the user can also see there are only two possible parts for each of the three components. The user also notices a constraint in this model, so they will also not waste time testing combinations of features which are incompatible.

If users wanted to test the effectiveness of various receivers in this system, they could slice this model to get a specific set of products. They could also design experiments to test products that have not yet been analyzed in the laboratory. Once they complete their experiments, users could add their results back to the Registry as annotations. This is a small example, but it demonstrates how product line engineering could help users construct valid cell-to-cell signaling programs.

As further motivation, if we return to the ASU team's experiments, one of their goals was to investigate the crosstalk, or the interactions between various parts. To test this, they designed experiments with multiple combinations of senders and receivers. Without realizing it, they defined a family of products for cell-to-cell signaling and evaluated the individual products. If they were working with a feature model they would have been able to: 1) efficiently sample the product space; 2) know how much of the space they explored; and 3) add constraints when they found crosstalk between parts. They could then annotate the feature models and create assets to describe their findings which could be used by another team working on a similar project. In essence, they could

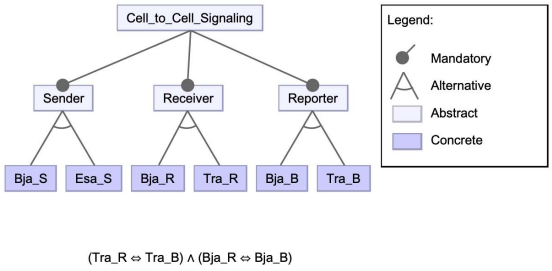

Figure 2: Example feature model for cell-to-cell signaling.

leverage the power of SPLs. It is this motivation that leads us to propose organic software product lines.

3 ORGANIC SOFTWARE PRODUCT LINES

Not long ago, the SPL community asked if open-source applications such as the Linux kernel should be considered product lines given that they are not managed and developed in the traditional manner [23, 31]. This has led to a broader view of SPLs. We ask the same question now of organic programs. Is there a mapping between traditional software product line engineering and synthetic biology that allows for managed development and reuse?

As Clements and Northrop state, the output of domain engineering should contain (1) a product line scope, (2) a set of core assets, and (3) a production plan [10]. This feeds into application engineering, which uses the production plan and scope to build and test individual products. Our focus in this work is primarily domain engineering, however we touch upon application engineering in the context of organic programs in the case study.

Assets. To begin we need to identify what constitutes a core asset for this domain. Assets in traditional product lines can include a software architecture, reusable software components, performance models, test plans and test cases, as well as other design documents. In organic programs, we see similar elements, discussed next. SBOL (or a similar representation) is used to define the functionality of a snippet of DNA code. This serves as an important design

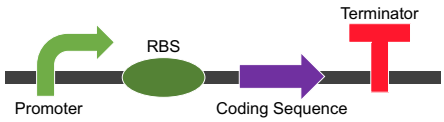

Figure 3: SBOL model of a transcription unit. This composite part is composed of four basic DNA parts: The promoter (also called a regulator), ribosome binding site (RBS), coding sequence, and terminator.

document for individual features. SBOL models can be composed and aggregated leading to composite models. The SBOL model for the simplest, stand-alone functional biological unit (called a *transcriptional unit*) can be seen in Figure 3. It is composed of four basic DNA parts: the promoter, ribosome binding site, coding sequence, and terminator.

The DNA sequence is the reusable software component. Like code it is not tangible, but must be implemented as a program and compiled to a machine level (or byte-code level) representation. DNA can be synthesized into a physical strand which can be inserted into a compiler (the living organism) for translation to machine level code (via the biological processes of transcription and translation of DNA via RNA into proteins). Other assets such as test cases and test plans can be constructed which define either laboratory experiments or virtual simulations. Both lead to evaluation of the program's expected, versus observed, functionality. Additional assets in the form of design documents and documentation can be provided such as safety cases [16] and higher level system architecture (*e.g.*, GenoCAD [7]).

Domain Engineering. During domain engineering the engineer defines the product line scope by choosing a family of behavior such as a type of molecular communication. He or she also defines the common and variable features and their relationships. An example of commonality is the transcription unit (Figure 3). The specific choices for promoters and binding sites defines the variability. When inserted into their host organisms at specific binding sites, the sets of DNA sequences define unique sets of *products*. Last, a production plan can be created in combination with a feature model and constraints. The feature model and constraints show how the DNA parts, as *features*, form a family of products, along with experimental notes on expected environmental conditions or other assumptions that are required for the program to run correctly.

Application Engineering. Application engineering involves combining the expected parts using standard DNA cloning techniques for insertion into a living organism [28]. Just as with traditional product lines, it is up to the engineer to adhere to constraints and only compose products defined in the feature model, otherwise unexpected behavior may occur. As in traditional software, some constraints may be hard-coded into the program, while some may represent a domain expectation instead.

Thüm *et al.* [36] in their survey of product lines discuss different implementations of product lines and analyses that can be performed at both the family and variability level. We believe organic product lines are just another type of implementation and we can utilize common analyses and techniques from SPL engineering in this domain. We study that next.

4 CASE STUDY

We conducted a case study to evaluate the feasibility of using product line engineering in synthetic biology. Supplemental data for this study can be found on our website.[1] We ask the following research questions:

RQ1: Does a DNA repository have the characteristics of a software product line?

RQ2: Can we build feature models representing families of products from an existing DNA Repository?

RQ3: Can SPL engineering provide useful analyses for developers of organic programs?

4.1 Subject Repository

We use as our subject repository the *Registry of Standard Biological Parts* (herein referred to as the BioBrick repository) as described in Section 2. We choose this subject as it is the largest open-source DNA repository and continues to grow (on average 2,995 parts are added each year).

For this work we define a *feature* simply as a BioBrick part. A *product* is the compilation of multiple basic BioBricks that together perform a cohesive function.

4.2 Data Collection

To study the core assets of the system, we use the BioBrick API to pull data for all parts up through December of 2018, consisting of 47,934 entries [19]. Each entry contains information such as the part_id, part_name, part_type, uses, status, and creation_date. *Usage* in this case is defined by counting how many times a part has been requested by a community user. This can tell us how useful a part is to the community.

4.3 Study Objects

For RQ1 we use the BioBrick repository. For RQ2 we select two biological functions in which to build a feature model. The BioBrick repository has sorted parts into ten common biological functions: biosafety, biosynthesis, cell-to-cell signaling and quorum sensing, cell death, coliroid, conjugation, motility and chemotaxis, odor production and sensing, DNA recombinations, and viral vectors. We select the two functions with the greatest number of parts (biosafety and cell-to-cell signaling). Biosafety has several subcategories, we choose the subcategory with the most parts —kill switch. For RQ3 we selected a 2017 iGEM team from Arizona State University [2] whose project is a subcategory of cell-to-cell signaling.

4.3.1 Kill Switch. A kill switch is a safety mechanism which triggers cellular death, typically by engineering cells to produce proteins which destroy cellular membranes. Common triggers include exposure to specific chemicals, temperature ranges, pH levels, or frequencies of light. To build the kill switch feature model we manually reviewed all of the wiki pages from the 110 teams who earned a gold medal in the 2017 iGEM competition [16]. Only 14 mentioned some type of kill switch in their design. We use those pages. The exact set of teams is listed on our website.

[1] https://sites.google.com/view/splc-dnafeatures

Table 1: Part types for all BioBrick parts in the repository.

part_type	# parts	part_type	# parts
Coding	10,265	RBS	769
Composite	9,966	Primer	685
Regulatory	4,165	Plasmid	681
Intermediate	3,506	Project	656
Generator	2,425	Terminator	518
Reporter	2,310	Signalling	511
Device	2,277	Plasmid_Backbone	454
DNA	1,717	Tag	385
Other	1,419	Scar	121
Measurement	1,162	Inverter	117
RNA	976	Cell	75
Protein_Domain	917	T7	57
Translational_Unit	880	Conjugation	51
Temporary	866	Promoter	3

4.3.2 Cell-to-Cell Signaling. Cell-to-cell signaling is a key cellular communication mechanism by means of secreting and sensing small molecules or peptides/proteins between a sender and a receiver. To build a feature model for cell-to-cell signaling, we forward-engineered the parts listed under the cell-to-cell signaling category of the BioBrick repository. We employed basic knowledge of the structure of a cell-to-cell signaling category and sorted parts by their features.

There are 39 promoters, 13 transcriptional regulators, 10 biosynthesis enzymes, 2 degradation enzymes, 21 transitional units, and 138 composite parts in this category. Since we want to map systems down to the lowest level of feature we ignore the composite parts.

5 RESULTS

In this section we answer each of our research questions in turn. In RQ1 we quantify assets from the BioBrick repository and explore both commonality and variability between products. We focus on domain engineering in RQ2 and RQ3. We move to application engineering in RQ3.

5.1 RQ1 Does the BioBrick repository have the characteristics of a software product line?

In Section 3 we defined organic software product lines in terms of SPL engineering concepts. We start with the core assets, the code. There are 47,934 BioBrick parts at the time of this publication. Table 1 shows the counts of parts by function. The largest category, *coding*, has over 10,000 parts. These are sequences that encode specific proteins. The second most frequent category (9,966) is *composite part*. A composite part is composed of two or more basic parts (*i.e.*, an aggregate class or function). All of the top ten categories have more than 1,000 parts. We can consider these reusable assets for building products.

We next analyzed the *use count* for each part. The use count specifies how many times a request for the part was made by an external user. This is similar to a GitHub checkout. Table 2 displays this data. We can see that the majority of parts (about 71%) are never requested. Approximately 27% are used between one and ten times. Then we see a small percentage of parts (under 2%) that are used more than 11 times. This demonstrates the repository consists of many reusable core assets. The parts with high use may show potential commonality between projects, and the parts with lower use may represent potential variability. We leave a complete analysis as future work. We see a similar phenomenon in traditional software repositories with a large abundance of code, but a comparatively small number of highly used modules [42].

Table 2: BioBrick use counts - # user requests.

# of Uses	# of Parts
0	34,091 (71.12%)
1-10	13,117 (27.36%)
11-50	602 (1.26%)
51-100	61 (0.13%)
101+	63 (0.13%)

Table 3: Assets present in 100 random composite parts.

Asset	SBOL model	DNA sequence	Textual description	Experimental results
% parts	82%	95%	90%	24%

Each part's web page in the BioBrick repository can contain several additional assets including the SBOL model, the raw DNA sequence, a part description in plaintext, and results from experimentation from iGEM teams. All of these assets may be useful to a user interested in how a part can fit into their construct. We randomly sampled 100 composite parts from the repository and identified whether they had these assets. Table 3 shows the results. 82% of parts included the SBOL format. Most of them included the raw DNA sequence (95%) and a basic textual description (90%). Only 24% of parts included any additional experimental results.

We now focus on two aspects of SPLs that we would expect to see in practice:

Variability. To examine variability we focus on the transcription unit, the most basic function (see Figure 3). There are 4,165 promoters, 769 RBSs, 10,265 coding sequences, and 518 terminators. If we underestimate the possible product space by counting one of each part (a standard practice is to use two terminators which will increase the space by a large factor) we have on the order of 1.7×10^{13} (17 trillion) products representing transcription units.

There are also 9,966 parts labeled as composite in the repository (meaning they were built from basic parts and added back into the repository). Each represents one customized product built from the core components, again showing variability.

Commonality. Not every product is completely distinct from others. Products will share certain common features with other products in their biological functionality. There are ten functional categories listed in the BioBrick repository. Each of these categories can represent one set of products, and they will share common architectural elements. Two examples of this include: (1) a kill switch will always have a trigger and an effect; and (2) a cell-to-cell signaling system will always have a sender, receiver, and reporter. In RQ3 we examine a real family of products that has 15 common features.

5.2 RQ2 Can we build a feature model from the BioBrick Repository?

For this research question we built two different feature models for two different functions in the BioBricks repository.

5.2.1 Kill Switch. Based on manual review of 2017 iGEM teams who earned a gold medal, we built the feature model shown in Figure 4. The *trigger type* (left side) is the most interesting because synthetic biologists will want to engineer kill switches to activate only under certain conditions. The kill switches we reviewed can be triggered under several different conditions: temperature ranges; presence or absence of specific chemicals; low pH levels; or exposure to specific frequencies of natural light. Each of those trigger conditions ends in leaf nodes which are the BioBrick IDs correlated to specific DNA sequences. The promoters, RBSs, coding sequences, and terminators show which BioBricks can be used to complete a transcription unit for a kill switch. The *visualization* branch is optional. It provides visual evidence that the switch is working through production of fluorescent proteins. This model represents 882 valid (different) kill switches.

5.2.2 Cell-to-Cell Signaling. Figure 5 shows the overall cell-to-cell signaling feature model we constructed. Recall that a basic cell-to-cell signaling system has three different of transcriptional units (sender, receiver, reporter). The feature model follows a hierarchical model with variation points at the transcriptional unit level as a sub-feature model. Because the full cell-to-cell signaling model is too large to show here, we visually present only some of these sub-feature models and describe the rest in text (the complete model is on our supplementary website).

Promoter: One promoter is needed for the sender, one for the receiver, and one for the reporter. A promoter has three features: *constitutiveness*, *activation*, and *repression*. A promoter may have a different level of constitutive expression (meaning it expresses on its own, without being activated by any protein). We found parts that categorize this as *weak*, *medium*, or *strong*. A promoter can also be activated (increased expression) by a protein. We identified four proteins listed under cell-to-cell signaling promoters. We identified seven proteins that could be used for repression.

We represent the three features of a promoter (constitutive, activation, repression) with an OR relationship. Each of the parts below these features have an Alternative relationship. In our model one promoter alone has 159 possible configurations. The model for the receiver promoter is expanded and can be seen in Figure 5.

RBS: The next high-level feature is the RBS. This part will have the same variation in the sender, receiver, and reporter. Since the cell-to-cell signaling catalog does not include RBS parts, we looked at all RBS parts. They are sorted into different collections, so we use the community collection. The functional differences between them is in their protein expression level ("strength"). This feature in the SPL has eight possible configurations.

Coding Sequence (Protein): The next part is the coding sequence. We limit this model to only coding sequences which encode for creation of specific proteins. We have identified five proteins in the cell-to-cell signaling catalog. In addition to a protein, a *function* is required to be chosen, either *transcriptional regulation* or an *enzyme*. An enzyme can either be for *biosynthesis* or *degradation*. There is also an optional *LVA tag* which reduces the proteins' half-life. The coding sequence parts have 30 possible configurations. This model is shown in Figure 6.

Coding Sequence (Reporter): The other type of coding sequence we model represents the encoding of the reporter's signal. We identify four possible behavioral responses: green fluorescence, biofilm production, antibiotic production, and a kill switch.

Terminator: The final part is the terminator. There are no terminators listed in the cell-to-cell signaling catalog, so we look at all terminators in the repository. Terminators can be in the forward direction, reverse direction, or be bidirectional. We only consider forward directional terminators in this model. In the BioBrick catalog there are 24 terminators available. It is common to choose two terminators to ensure transcription stops, so in our model we allow choosing one or two (a {1,2} OR relation) terminators. The terminator parts allow 300 possible configurations. Since our model was too large for FeatureIDE to calculate the set of products we use FaMa [4]. The sender and receiver each have 11,448,000 products. The reporter has 5,724,000 products. The total number of products is 7.5×10^{20}.

5.3 RQ3: Can SPL engineering provide useful analyses for organic programs?

For this research question we ask questions related to application engineering. We use the 2017 iGEM team project from Arizona State University [2, 33] introduced in our motivating example. Because this team built a quorum sensing network, which is a type of cell-to-cell signaling, it helps us validate the results from RQ1 from an application engineering perspective.

In quorum sensing, there are two groups of organisms: the first group acts as the sender, and the second as the receiver. The receiver will exhibit a type of behavioral response at a certain concentration (a quorum) signaled by the presence of both these proteins. For example, the receiving organism may turn green when enough of the sending organism's signal is sensed in the system.

The choice of protein for the sender and the receiver plays an important role. Some combinations of proteins cause crosstalk for the receiver which can render the system inefficient or even useless. As stated on the ASU team's project web page: "Knowing the rates of induction also allows for greater precision when designing genetic circuits." The team chose ten different proteins for the sender (Aub, Bja, Bra, Cer, Esa, Las, Lux, Rhl, Rpa, Sin), three different proteins for the receiver (Las, Lux, Tra), and three different proteins for the reporter (Las, Lux, Tra).

5.3.1 Providing a Broader View of the Product Space. Though perhaps unintentionally, the ASU team's project represents a feature model. We formalize it by manually reverse-engineering their project feature model from their web page. Figures 7 and 8 show the resulting sender and receiver models respectively (herein referred to together as the *ASU model*). This model contains the high-level features: sender and receiver. The receiver contains both a regulator and reporter. Each feature has four basic parts (promoter, RBS, coding sequence, terminator) and the sender has an extra feature to incorporate a red fluorescence. Many of the features are mandatory. The points of variability lie in the sender's coding sequence, the regulator's coding sequence, and the reporter's promoter. This model

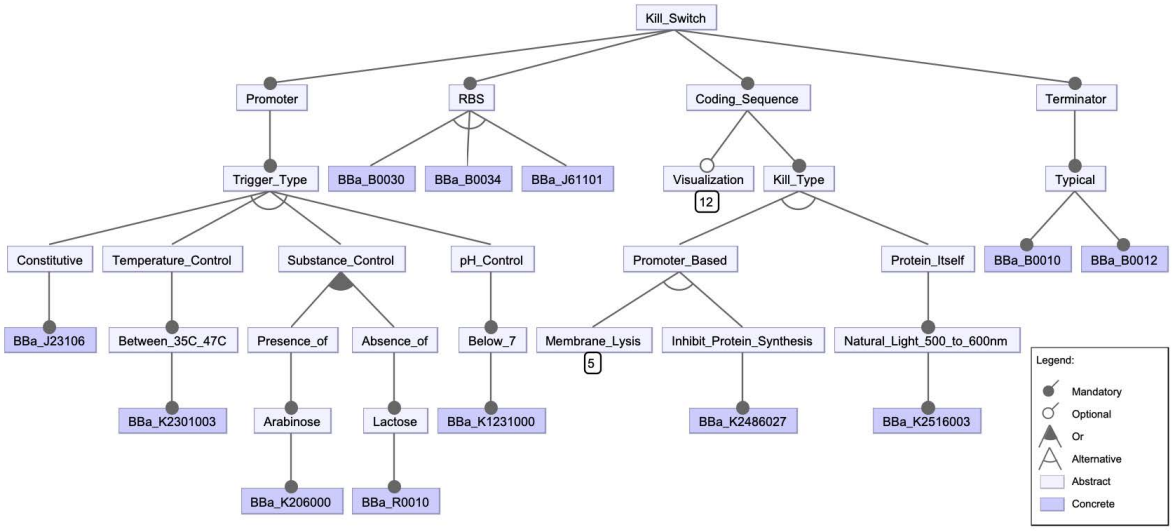

Figure 4: A feature model for a kill switch. An integer on a leaf represents how many nodes are in their subtrees (obfuscated).

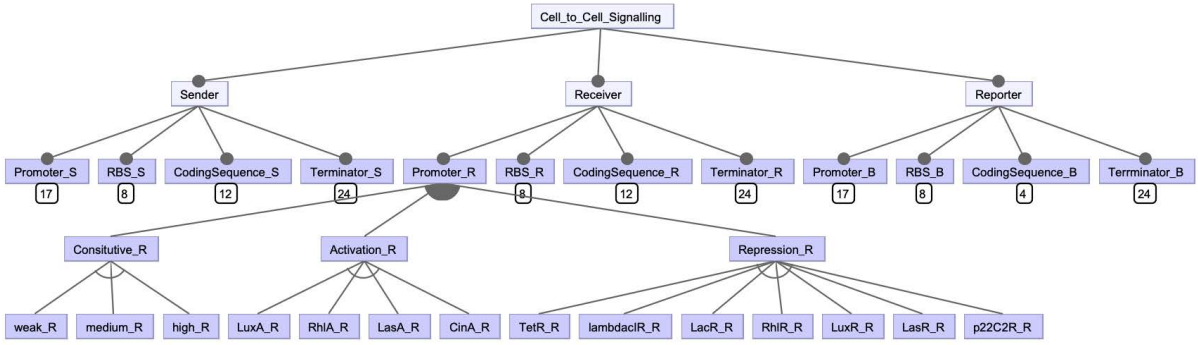

Figure 5: Top levels of the cell-to-cell signaling feature model.

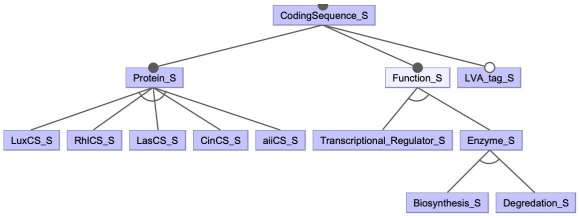

Figure 6: Sub-feature model of a protein coding sequence.

represents a total of 90 products. To conduct their experiments, the ASU team added a constraint between the regulator's coding sequence and the reporter's promoter (they are required to be the same). Thus their experiment tested 30 of these products in the laboratory. We call this the *ASU experiment model*.

If we compare this ASU model to the reverse engineered model presented in Section 5.2.2 (we call this the *cell-to-cell signaling model*), the ASU model is not a direct subset of the cell-to-cell signaling model. In practice it should be. We would have expected the products in the models to overlap, as seen in Figure 9a. However the actual overlap can be seen in Figure 9b. We can see that only 12 products overlap between the ASU model and the cell-to-cell signaling model. There are an additional 78 products that the cell-to-cell signaling model misses.

To understand why there is such a small overlap we examine the features each model considers. The ASU team's experimental focus was on the interactions of protein features for the sender, regulator, and reporter, so the points of variation were chosen on these three variables. We expected a complete set of proteins to be documented on the cell-to-cell signaling page, but they are not comprehensively listed in the BioBrick repository.

Both models are valid representations of quorum sensing systems, however they come from different resources and their products differ. This highlights a key problem with the current method of engineering a model - there is a lack of complete information available. In an ideal world we would show a complete list of all proteins, possibly through a central, open-source feature model.

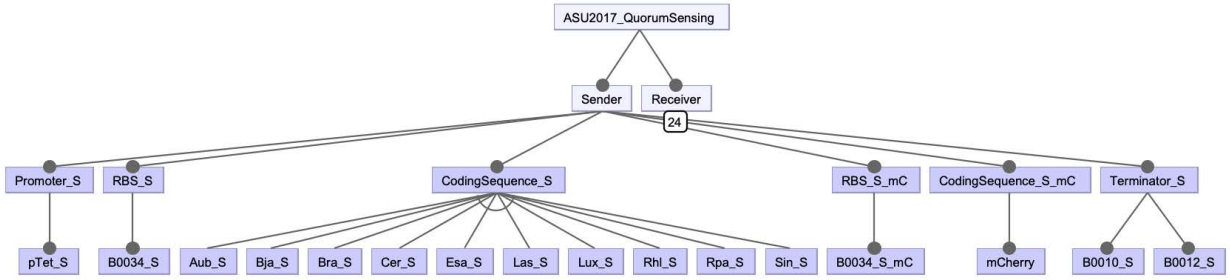

Figure 7: The sender slice from a feature model for the 2017 Arizona State University's iGEM project (ASU Model).

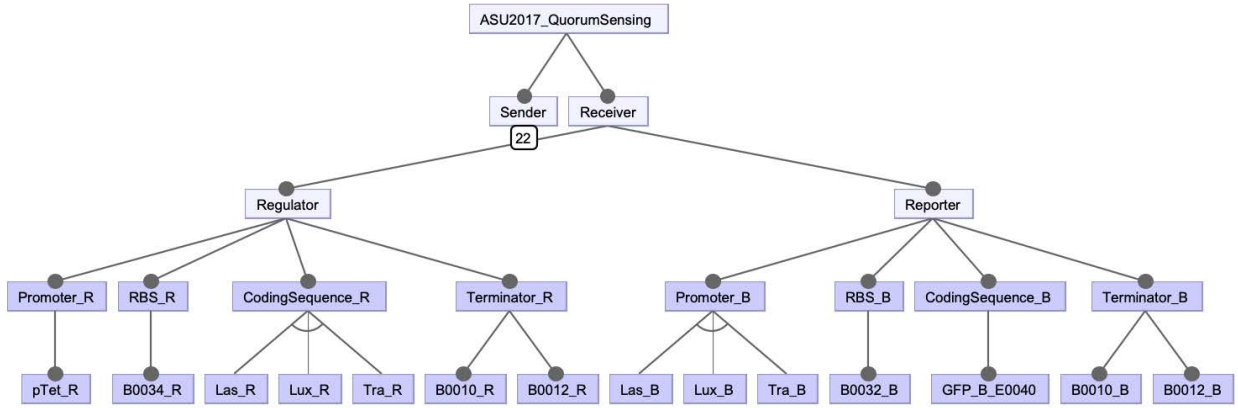

Figure 8: The receiver slice from a feature model for the 2017 Arizona State University's iGEM project (ASU Model).

(a) Expected overlap

(b) Actual overlap

Figure 9: The overlap of the feature models and ASU experimentation. The sum of each circle is in parentheses.

5.3.2 Testing and Analysis. We next move to testing and analysis of our applications. Assume the ASU team used the cell-to-cell signaling model to drive their experiments instead of working from scratch. We demonstrate how they could have used this feature model to help with testing and analysis.

Since the ASU team focused only on the proteins, they could have *sliced* the cell-to-cell signaling model. This would yield the model in Figure 10 (called *protein slice*) which represents 100 products. This is significantly fewer than the total product space (7.50169×10^{20}), but more than what the ASU team eventually tested (30).

We could also employ common sampling methods such as combinatorial interaction testing (CIT) which samples broadly across a set of features [11]. Using sampling allows us to test a larger space of combinations. We used CASA [18] to build a 2-way CIT sample of the ASU model. In this scenario an interaction is between two proteins. We can cover all pairs of proteins in the complete model using 30 tests. Note that ASU also tested 30 products, but in order to scope the project they applied their own constraint that does not test for possible interactions between the regulator and reporter proteins. Using CIT will more broadly sample the interaction space of all three proteins.

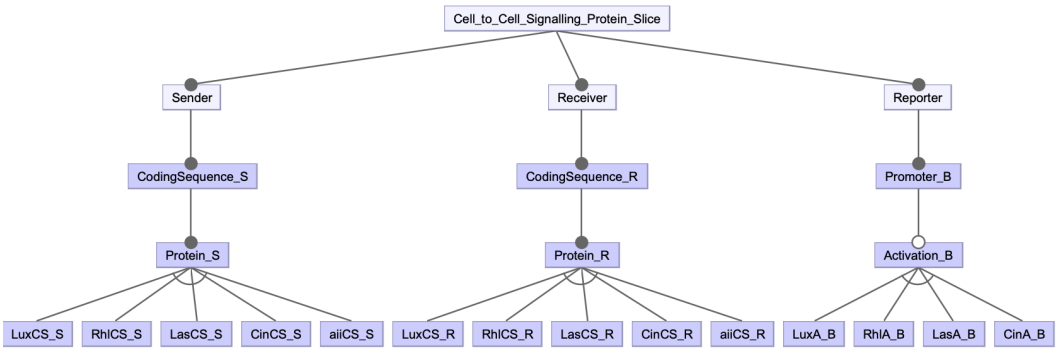

Figure 10: A slice of the complete cell-to-cell signaling feature model focuses on protein combinations to mimic the ASU team's experiment (protein slice).

We also built a 2-way CIT sample for the cell-to-cell signaling model, covering all pairs of *all features*. This has 715 tests. This is a significant reduction of the entire configuration space, however it may be too large in practice. Each of these samples and their product overlaps can be seen in Figure 11. All samples can be found on our associated website.

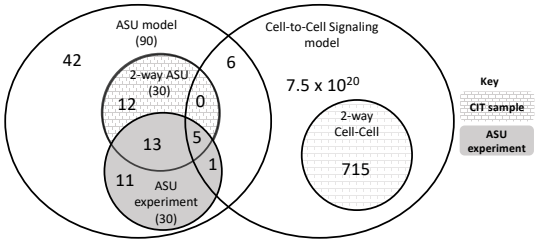

Figure 11: Product overlap between each of the feature models and possible samples.

5.3.3 Leveraging Existing Reverse Engineering Tools. Given that synthetic biology is an end-user software engineering domain, developers of organic programs may not be well versed in software development methods. Therefore, building a feature model from scratch may be a roadblock. We have observed this in other recent studies that have applied software engineering concepts to synthetic biology [16]. Instead, we want to understand whether it would be possible to obtain an initial feature model using only a set of products (which is how the ASU team built their experiments). This model could then be used to define CIT sampling, etc.

We utilized an existing reverse engineering tool, SPLRevO [34, 35]. We note that other similar tools could also be used. This tool accepts either (1) a set of constraints based on domain knowledge describing the compatibility of the DNA parts, or (2) a set of products which can be the known working composite components. The tool then uses a genetic algorithm to automatically build a feature model that represents all products. The fitness function (validity) aims to maximize the set of existing products while minimizing any additional products using a penalty. Because we do not have a set of constraints for this model we used a set of products as inputs. We believe that a domain-specific language for users that describes the product line and which can generate constraints is an interesting avenue for future work.

The current prototype of SPLRevO can handle up to 27 features when reverse engineering from products. Therefore, we reduced the ASU model from 30 to 27 features by selecting two of the common features (B0010 and B0012) and merging them into one feature (B0015) for the sender, regulator, and reporter (e.g. B0010_S+B0012_S→B0015_S, B0010_R+B0012_R→B0015_R, etc.). Since there are 15 features in the ASU model that are common to all products, we could have chosen any of those features to combine or remove while still representing the same 30 products. SPLRevO returned the feature model seen in Figure 12. It was able to provide us with a model that closely resembles the hand-built model and has 100% validity (it represents exactly the same number of products). Though the models represent the same products, their physical structure is different. The SPLRevO model grouped the sender's protein coding sequences together like the ASU model (Group1). Group2 represents the proteins for the regulator and reporter. Instead of adding a cross-tree constraint like the ASU team did for their experiment, the SPLRevO model uses mandatory relations for these under Group2. The rest of the features are all mandatory.

During this process we realized that 100% validity, while a good result for this study, might be relaxed in a real scenario. If the user provides a set of products, but they are not complete, relaxation could be used to allow them to interactively improve their model. We may also want to add additional features (from the BioBricks repository) and show the potentially larger model. Ultimately, if we can represent BioBricks features using a constraint language, there is an opportunity for direct reverse-engineering and greater scalability. SPLRevO has been evaluated on reverse-engineering a feature model up to 100 features when using input constraints [35].

6 DISCUSSION

We found several interesting insights when working on this study. First, we found the domain engineering from an open source repository like this to be challenging. We did not obtain the same feature model for cell-to-cell signaling as ASU, and cannot be certain whether they had additional domain expertise to restrict their study

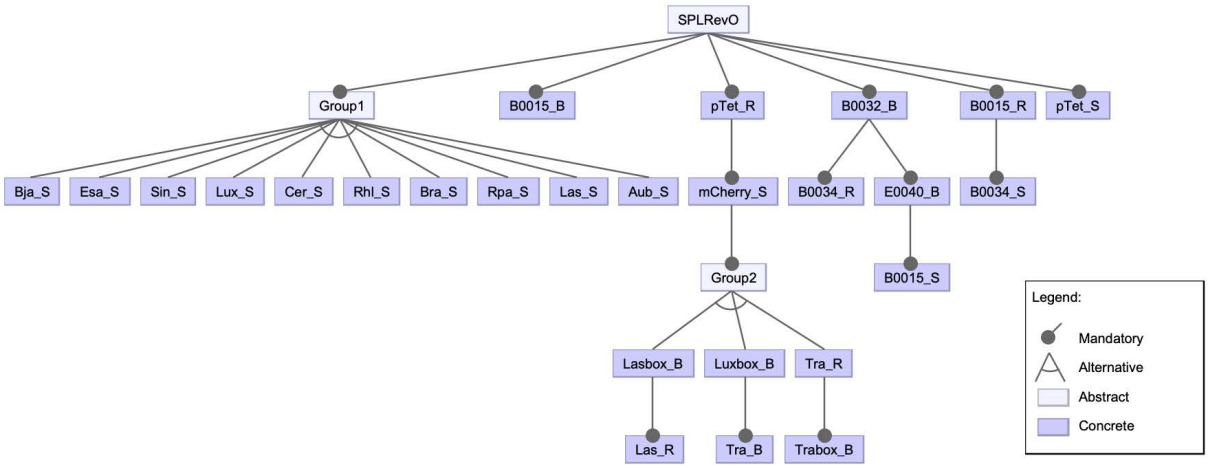

Figure 12: A reverse-engineered feature model using SPLRevO.

or if this was due to chance. However, we believe that an interactive environment that we have described would be useful. Recent work on other SPL development in open source repositories are looking at ways to provide branching constructs for more easy reuse and understandability [25]. Second, there is an opportunity to develop more techniques for domain experts (who may not understand feature modeling) to work with product line engineers to build models that are functionally useful. It would be useful to provide some domain specific tools that can be used to generate sets of constraints, rather than require the user to either build the full feature model or list a set of products by hand. Third, we believe there is an opportunity for users to provide qualitative/experimental information which can be returned to the product line and connected via trace links. These can be provided as assets in the BioBricks Repository. Last, we note that there is a lot of noise in this repository. For instance, some parts do not have a DNA sequence which means it is empty code. We did not attempt to account for all of that during our study, but leave that as future work.

7 RELATED WORK

This paper follows a long line of research on software product lines. We do not attempt to summarize all of that work here, but point readers to several good surveys on this topic [3, 36]. Product line engineering has been applied in many emerging domains recently including drones and nanodevices [9, 24]. There is also a push towards open-source product lines [23, 25, 31] and the concept of a software ecosystem, where the community modifies and customizes product lines using a common platform and look [27].

The closest work is that of Lutz et al. [24] who study a family of DNA nanodevices. While they also look at DNA, they study chemical reaction networks (CRNs), rather than a living synthetically engineered organism. Their line of organic programming leverages CRNs, which are sets of concurrent equations that can be compiled into single strands of DNA [41]. CRNs have been shown go be Turing complete [14]. CRNs are not necessarily part of living organisms; there is no transcription and translation of the code [15].

Ours is not the first analysis of the BioBricks repository. Valverde *et al.* examined the relationships within the repository from a network perspective to gain an understanding of the software complexity (they too treat this as a software ecosystem) [37].

This work differs from the existing work in that we demonstrate the use of domain engineering to build a family of synthetic biology products which can be analyzed and reasoned about using traditional SPL engineering techniques.

8 CONCLUSIONS AND FUTURE WORK

In this paper we have shown how the emerging programming field of synthetic biology can potentially benefit from software product line engineering. We first presented the notion of an organic software product line. We then used the largest open source DNA repository to analyze 1) whether there are assets which are reused and products that share common and variable elements, 2) whether we can build feature models to represent the products in this repository, and 3) how common SPL techniques can be used to benefit product line development in this domain. We found reusable assets, commonality, and variability in the repository. We were able to build feature models to represent several common functions. We then demonstrated how we might automatically reverse engineer a model and how this can help users test and reason about the product space more comprehensively.

In future work we plan to investigate building a domain specific language for generating SPL constraints, and evaluating this approach in practice with teams of synthetic biologists perhaps in the iGEM Competition. We also plan to investigate challenges that are shared with modern SPLs including scalability, dynamic feature models, and collaborative SPLs.

ACKNOWLEDGMENTS

We would like to thank the Haynes Lab at Emory University (formally Arizona State University) for sharing additional artifacts with us. This work is supported in part by NSF Grant CCF-1901543, National Institute of Justice Grant 2016-R2-CX-0023, and by NSF Grant CBET-1805528.

REFERENCES

[1] James Anderson, Natalja Strelkowa, Guy-Bart Stan, Thomas Douglas, Julian Savulescu, Mauricio Barahona, and Antonis Papachristodoulou. 2012. Engineering and ethical perspectives in synthetic biology. *EMBO reports* 13, 7 (2012), 584–590. https://dx.doi.org/10.1038/embor.2012.81

[2] Arizona State University 2017. ASU iGEM 2017: Engineering variable regulators for a quorum sensing toolbox. Retrieved June 13, 2019 from https://2017.igem.org/Team:Arizona_State

[3] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: a literature review. *Information Systems* 35, 6 (2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[4] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-cortés. 2007. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS)*. 129–134.

[5] Lara Tess Bereza-Malcolm, Gülay Mann, and Ashley Edwin Franks. 2014. Environmental sensing of heavy metals through whole cell microbial biosensors: a synthetic biology approach. *ACS Synthetic Biology* 4, 5 (2014), 535–546. https://doi.org/10.1021/sb500286r

[6] James Bornholt, Randolph Lopez, Douglas M. Carmean, Luis Ceze, Georg Seelig, and Karin Strauss. 2016. A DNA-based archival storage system. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 637–649. https://doi.acm.org/10.1145/2980024.2872397

[7] Yizhi Cai, Mandy L. Wilson, and Jean Peccoud. 2010. GenoCAD for iGEM: a grammatical approach to the design of standard-compliant constructs. *Nucleic Acids Research* 38, 8 (2010), 2637–2644. https://doi.org/10.1093/nar/gkq086

[8] D. Ewen Cameron, Caleb J. Bashor, and James J. Collins. 2014. A brief history of synthetic biology. *Nature Reviews Microbiology* 12, 5 (2014), 381–390. https://doi.org/10.1038/nrmicro3239

[9] Jane Cleland-Huang, Michael Vierhauser, and Sean Bayley. 2018. Dronology: An incubator for cyber-physical systems research. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE)*. 109–112. https://doi.acm.org/10.1145/3183399.3183408

[10] Paul Clements and Linda Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA.

[11] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444. https://doi.org/10.1109/32.605761

[12] Ramiz Daniel, Jacob R. Rubens, Rahul Sarpeshkar, and Timothy K. Lu. 2013. Synthetic analog computation in living cells. *Nature* 497, 7451 (2013), 619–623. https://doi.org/10.1038/nature12148

[13] Michael B. Elowitz and Stanislas Leibler. 2000. A synthetic oscillatory network of transcriptional regulators. *Nature* 403 (2000), 335–338. Issue 6767. https://doi.org/10.1038/35002125

[14] François Fages, Guillaume Le Guludec, Olivier Bournez, and Amaury Pouly. 2017. Strong Turing completeness of continuous chemical reaction networks and compilation of mixed analog-digital programs. In *Proceedings of the 15th International Conference on Computational Methods in Systems Biology (CMSB)*. 108–127. https://doi.org/10.1007/978-3-319-67471-1_7

[15] Martin Feinberg. 1987. Chemical reaction network structure and the stability of complex isothermal reactors–I. The deficiency zero and deficiency one theorems. *Chemical Engineering Science* 42 (1987), 2229–2268. https://doi.org/10.1016/0009-2509(87)80099-4

[16] Justin Firestone and Myra B. Cohen. 2018. The assurance recipe: facilitating assurance patterns. In *Proceedings of the International Conference on Computer Safety, Reliability, and Security (SAFECOMP), ASSURE Workshop*. 22–30. https://doi.org/10.1007/978-3-319-99229-7_3

[17] Timothy S. Gardner, Charles R. Cantor, and James J. Collins. 2000. Construction of a genetic toggle switch in Escherichia coli. *Nature* 402 (2000), 339–342. Issue 6767. https://doi.org/10.1038/35002131

[18] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering* 16, 1 (2011), 61–102. https://doi.org/10.1007/s10664-010-9135-7

[19] iGEM API 2018. Registry of Standard Biological Parts API. iGEM Foundation. Retrieved June 13, 2019 from https://parts.igem.org/Registry_API

[20] iGEM Competition 2018. International Genetically Engineered Machine Competition. iGEM Foundation. Retrieved June 13, 2019 from https://igem.org

[21] iGEM Registry 2018. Registry of Standard Biological Parts. iGEM Foundation. Retrieved June 13, 2019 from https://parts.igem.org

[22] Zoltán Kis, Hugo Sant'Ana Pereira, Takayuki Homma, Ryan M. Pedrigi, and Rob Krams. 2015. Mammalian synthetic biology: emerging medical applications. *Journal of The Royal Society Interface* 12, 106 (2015), 1–18. https://doi.org/10.1098/rsif.2014.1000

[23] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux kernel variability model. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC)*. 136–150. https://doi.org/10.1007/978-3-642-15579-6_10

[24] Robyn R. Lutz, Jack H. Lutz, James I. Lathrop, Titus H. Klinge, Divita Mathur, Donald M. Stull, Taylor G. Bergquist, and Eric R. Henderson. 2012. Requirements analysis for a product family of DNA nanodevices. In *Proceedings of the 20th IEEE International Requirements Engineering Conference (RE)*. 211–220. https://doi.org/10.1109/RE.2012.6345806

[25] Leticia Montalvillo and Oscar Díaz. 2015. Tuning GitHub for SPL development: branching models & repository operations for product engineers. In *Proceedings of the 19th International Conference on Software Product Line (SPLC)*. 111–120. https://doi.acm.org/10.1145/2791060.2791083

[26] Alec A.K. Nielsen, Bryan S. Der, Jonghyeon Shin, Prashant Vaidyanathan, Vanya Paralanov, Elizabeth A. Strychalski, David Ross, Douglas Densmore, and Christopher A. Voigt. 2016. Genetic circuit design automation. *Science* 352, 6281 (2016). https://doi.org/110.1126/science.aac7341

[27] Konstantinos Plakidas, Srdjan Stevanetic, Daniel Schall, Tudor B. Ionescu, and Uwe Zdun. 2016. How do software ecosystems evolve? A quantitative assessment of the R ecosystem. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC)*. 89–98. https://doi.acm.org/10.1145/2934466.2934488

[28] Jiayuan Quan and Jingdong Tian. 2009. Circular polymerase extension cloning of complex gene libraries and pathways. *PloS one* 4, 7 (2009), 1–6. https://doi.org/10.1371/journal.pone.0006441

[29] Ricardo A. Rossello and David H. Kohn. 2010. Cell communication and tissue engineering. *Communicative & Integrative Biology* 3, 1 (2010), 53–56. https://doi.org/10.4161/cib.3.1.9863

[30] SBOL 2019. Synthetic Biology Open Language. SBOL Research Group. Retrieved June 13, 2019 from https://sbolstandard.org

[31] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2007. Is the Linux kernel a software product line?. In *Proceedings of the 2nd SPLC Workshop on Open Source Software and Product Lines*. 1–4.

[32] Federico Tavella, Alberto Giaretta, Triona Marie Dooley-Cullinane, Mauro Conti, Lee Coffey, and Sasitharan Balasubramaniam. 2018. DNA molecular storage system: Transferring digitally encoded information through bacterial nanonetworks. (2018). arXiv:1801.04774

[33] Stefan J. Tekel, Christina L. Smith, Briana Lopez, Amber Mani, Christopher Connot, Xylaan Livingstone, and Karmella Ann Haynes. 2019. Engineered orthogonal quorum sensing systems for synthetic gene regulation in Escherichia coli. *Frontiers in Bioengineering and Biotechnology* 7, 80 (2019). https://doi.org/10.1101/499681

[34] Thammasak Thianniwet and Myra. B. Cohen. 2015. SPLRevO: Optimizing complex feature models in search based reverse engineering of software product lines. In *Proceedings of the 1st North American Search Based Software Engineering Symposium (NasBASE)*. 1–16.

[35] Thammasak Thianniwet and Myra. B. Cohen. 2016. Scaling up the fitness function for reverse engineering feature models. In *Symposium on Search-Based Software Engineering (SSBSE)*. 128–142. https://doi.org/10.1007/978-3-319-47106-8_9

[36] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)* 47, 1, Article 6 (2014), 45 pages. https://doi.acm.org/10.1145/2580950

[37] Sergi Valverde, Manuel Porcar, Juli Peretó, and Ricard V. Solé. 2016. The software crisis of synthetic biology. *bioRxiv* (2016). https://doi.org/10.1101/041640

[38] Wilfried Weber and Martin Fussenegger. 2012. Emerging biomedical applications of synthetic biology. *Nature Reviews Genetics* 13, 1 (2012), 21–35. https://doi.org/10.1038/nrg3094

[39] Wilfried Weber, Jörg Stelling, Markus Rimann, Bettina Keller, Marie Daoud-El Baba, Cornelia C. Weber, Dominique Aubel, and Martin Fussenegger. 2007. A synthetic time-delay circuit in mammalian cells and mice. *Proceedings of the National Academy of Sciences of the United States of America* 104, 8 (2007), 2643–2648. https://doi.org/10.1073/pnas.0606398104

[40] William B. Whitaker, Nicholas R. Sandoval, Robert K. Bennett, Alan G. Fast, and Eleftherios T. Papoutsakis. 2015. Synthetic methylotrophy: engineering the production of biofuels and chemicals based on the biology of aerobic methanol utilization. *Current Opinion in Biotechnology* 33 (2015), 165–175. https://doi.org/10.1016/j.copbio.2015.01.007

[41] Erik Winfree. 1995. On the computational power of DNA annealing and ligation. In *DNA Based Computers*. https://resolver.caltech.edu/CaltechAUTHORS:20111024-133436564

[42] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2014. Patterns of folder use and project popularity: A case study of GitHub repositories. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, Article 30, 4 pages. https://doi.org/10.1145/2652524.2652564