Empirical Software Engineering (2021) 26: 44
https://doi.org/10.1007/s10664-021-09940-0

An empirical investigation of organic software product lines

Mikaela Cashman[1,2] · Justin Firestone[3] · Myra B. Cohen[1] · Thammasak Thianniwet[4] · Wei Niu[5]

Accepted: 15 January 2021 / Published online: 25 March 2021
© The Author(s) 2021

Abstract
Software product line engineering is a best practice for managing reuse in families of software systems that is increasingly being applied to novel and emerging domains. In this work we investigate the use of software product line engineering in one of these new domains, 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 DNA parts. In this paper we perform an investigation of domain engineering that leverages an open-source repository of more than 45,000 reusable DNA parts. We show the feasibility of these new types of product line models by identifying features and related artifacts in up to 93.5% of products, and that there is indeed both commonality and variability. We then construct feature models for four commonly engineered functions leading to product lines ranging from 10 to 7.5×10^{20} products. In a case study we demonstrate how we can use the feature models to help guide new experimentation in aspects of application engineering. Finally, in an empirical study we demonstrate the effectiveness and efficiency of automated reverse engineering on both complete and incomplete sets of products. In the process of these studies, we highlight key challenges and uncovered limitations of existing SPL techniques and tools which provide a roadmap for making SPL engineering applicable to new and emerging domains.

Keywords Software product lines · Synthetic biology · Reverse engineering · BioBricks

1 Introduction

Software product line (SPL) engineering has become a best practice for modeling, building, and managing families of software systems. It is epitomized by the use of common and

The first co-author is employed at Oak Ridge National Laboratory (a government-funded organization).

Communicated by: Laurence Duchien, Thomas Thüm and Paul Grünbacher

This article belongs to the Topical Collection: *Configurable Systems*

✉ Mikaela Cashman
mcashman.isu@gmail.com

Extended author information available on the last page of the article.

variable building blocks that can be combined in different ways as guided by a well-specified model (the feature model) (Clements and Northrop 2002). SPL engineering stems from the need to ensure efficient reuse and to improve system quality. It was originally restricted to commercial software development, where SPLs' assets could be well managed and planned. However, in recent years SPL engineering practices have moved into the broader software engineering community, and these are being applied to many types of non-commercial systems such as the Linux Kernel (Lotufo et al. 2010) and to traditional (highly-configurable) software systems such as `gcc` and `Firefox` (Garvin et al. 2013).

At the same time, the software development community has been gravitating towards open-source repositories such as GitHub, a marketplace where developers can find libraries and other reusable components. It is natural, therefore, that the SPL community has begun applying the idea of SPL engineering directly to open-source systems by defining open-source product lines (Sincero et al. 2007), and to other emerging domains such as cyber-physical systems (Cleland-Huang et al. 2018) and the Internet of Things (IoT) (Ayala et al. 2015; Cetina et al. 2009; Tzeremes and Gomaa 2018). Recently, Montalvillo and Díaz have proposed techniques to aid SPL practices within GitHub (Montalvillo and Díaz 2015).

In this paper, we ask whether SPL engineering can be applied to yet another emerging domain, that of synthetic biology. Synthetic biology, the practice of engineering living organisms by modifying their DNA, has advanced quickly over the last 30 years (Cameron et al. 2014). It is being used for sensing heavy metals for pollution mitigation (Bereza-Malcolm et al. 2014), development of synthetic biofuels (Whitaker et al. 2015), engineering cells to communicate and produce bodily tissues (Rossello and Kohn 2010), emerging medical applications (Kis et al. 2015; Weber and Fussenegger 2012), and basic computational purposes (Daniel et al. 2013). 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 (iGEM Competition 2018). 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) (iGEM Registry 2018) 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 private, 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. Next they 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 our initial study on this topic (Cashman et al. 2019), we explored the idea of using software 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 and that we have what we call *organic software product lines* (OSPLs). In this paper we extend this work and evaluate this claim in more depth and rigor. We have added two new engineered functions to our study and incorporate repetition and statistical analysis. In our original paper we provided only a single instance of reverse engineered feature model. In this paper, we separate reverse engineering into its own empirical study and apply this to all of our models. We also collect statistics over 340 runs to evaluate the quality of the results. We have also identified several interesting directions for SPL engineering techniques that may be applicable to other emerging SPL domains as well. This includes the need for continued attention and better support of (1) SPL evolution; (2) duplicate features; (3) scalability of reverse engineering; and (4) interactive feature modeling from incomplete sets of products. Last we have expanded the background and related work.

The contributions of this work are:

1. A mapping of software product line engineering to the domain of synthetic biology resulting in organic software product lines;
2. Empirical evidence demonstrating the potential reuse and existence of both commonality and variability in the BioBricks repository;
3. A study with four manually and automatically reverse engineered feature models, showing that we can build feature models which have potential to help synthetic biologists; and
4. A discussion on key challenges and limitations of existing techniques for organic software product lines and its implications for the software product line engineering community.

In the next section we present background on synthetic biology and a motivating example. Section 3 presents background on software product lines. We then propose the notion of an organic software product line (OSPL) in Section 4. We provide a roadmap for our evaluation in Section 5. We follow this with a feasibility study (Section 6), a case study demonstrating OSPLs in practice (Section 7), and an empirical study of automated reverse engineering of OSPLs (Section 8). We discuss the implications and future of OSPLs in Section 9. We end with conclusions and future work.

2 Living Organisms as Programs

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" (Anderson et al. 2012). This definition highlights the use of *design*. It begins with an abstracted model, which is then implemented by inserting strands of DNA (encoded with specific functionality) 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). It follows that we can view synthetic biology as a programming discipline.

This analogy of *programming biology* is not a new concept (Bornholt et al. 2016; Cai et al. 2010; Elowitz and Leibler 2000; Gardner et al. 2000; Nielsen et al. 2016; SBOL 2019; Tavella et al. 2018; Weber et al. 2007). There is even a programming language called the Synthetic Biology Open Language (SBOL) which defines a common way to represent biological designs (SBOL 2019). Researchers have used synthetic biology to create a context-free grammar using BioBricks (Cai et al. 2010), automated design of genetic circuits with NOT/NOR gates (Nielsen et al. 2016), and bacterial networks to use DNA for data storage (Bornholt et al. 2016; Tavella et al. 2018). There are also several examples of organic programs inspired by classic computer science constructs such as a genetic oscillator (Elowitz and Leibler 2000), a genetic toggle switch (Gardner et al. 2000), and a time-delay circuit (Weber et al. 2007).

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 (iGEM Registry 2018) (BioBrick repository 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” (iGEM Competition 2018). 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 can respond to the signal by emitting a 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 their knowledge of synthetic biology, then this approach is ad hoc. As in software development, they often would expect to use existing modules and only customize parts that are specific to their needs. If they want to reuse existing modules, then this analogy is similar to someone searching GitHub for code that performs a particular function. Suppose the user searches the BioBrick repository for “signalling” (with two “l” characters). 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 hits link to parts actually involved in cell-to-cell signaling. Some are links to 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 BioBrick repository.

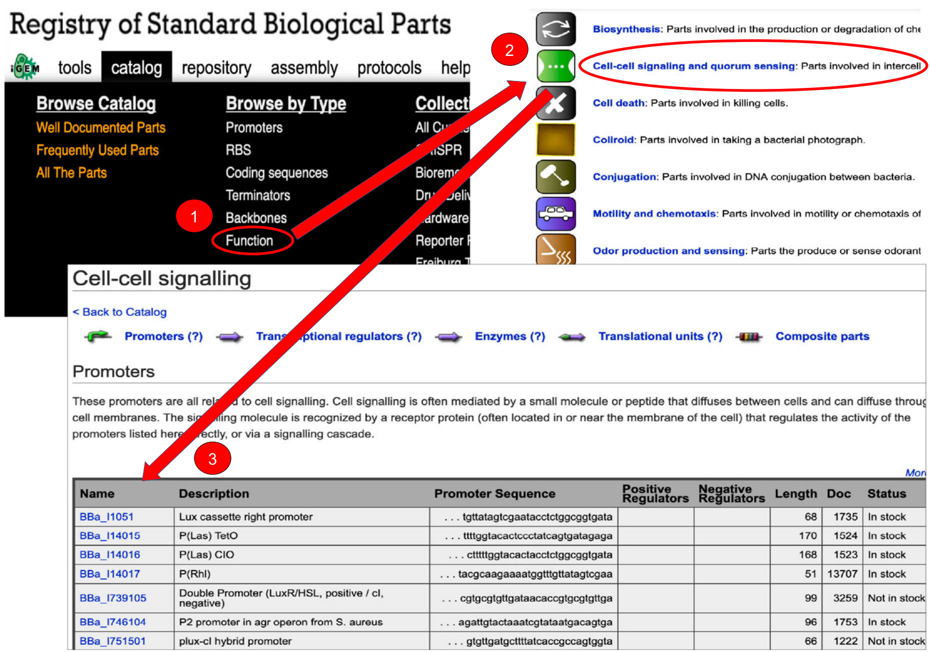


Fig. 1 Browse by function → Cell-to-cell signaling

There are 10 functions listed including “Cell-to-cell signaling and quorum sensing.” Parts are sorted (Fig. 1—step #3) into various lower-level categories on this page. Many 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 the 2017 iGEM team from Arizona State University (ASU) (Arizona State University 2017). 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 Fig. 2 (a subset of a feature model from our empirical study)? From this model the user immediately can see the architecture of their system. First, they learn that any cell-to-cell signaling system has three basic parts: a sender, a receiver, and a reporter. They see the reporter is also optional (you may have a system that recognizes a signal and does not respond). 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.

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

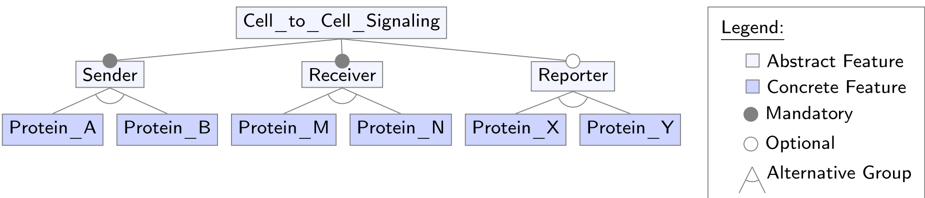


Fig. 2 Example feature model for a cell-to-cell signaling system

small example, but it demonstrates how software 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 issue of *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 leverage the power of SPLs. We explore these opportunities in more depth in our subsequent studies.

3 Software Product Lines

Software product line (SPL) engineering is a best practice for modeling, building, and managing reuse in families of software systems (Clements and Northrop 2002; Benavides et al. 2010; Pohl et al. 2005; Thüm et al. 2014a). The practice stems from the need to ensure efficient reuse and to improve system quality in large software systems that have both variable and common components. It is epitomized by the use of common and variable building blocks (called *features*) that can be combined in different ways to create individual *products*. An essential component of an SPL is the *feature model* which is a representation of the entire product space (Kang et al. 1990; Clements and Northrop 2002; Benavides et al. 2010). SPLs can be represented using a set of first order logic constraints, from which all possible products can be enumerated. The feature model provides a visual representation of this space in a compact and human readable view. Feature models represent individual features as nodes in a tree with dependencies (constraints) shown between these nodes. Some features will be mandatory to all products, some will be optional, and others are alternative or OR groupings. Constraints that cannot be shown graphically, are encoded as *cross-tree constraints*. Underlying the feature model is a logical representation of all constraints of the system. Common feature modeling tools exist such as FeatureIDE (Thüm et al. 2014b). Feature models are used to guide product creation, product maintenance and can also be used to support software testing activities (Benavides et al. 2010; Thüm et al. 2014b).

As an example consider a company that develops a line of cellular phones and the accompanying feature model in Fig. 3. The software product (cell phone) is depicted at the root of a tree. This software product line has a variety of features such as the operating system, storage components, media, and a network protocol. Each of these features have different options available to them, for example the operating system could be Android or iOS. Since

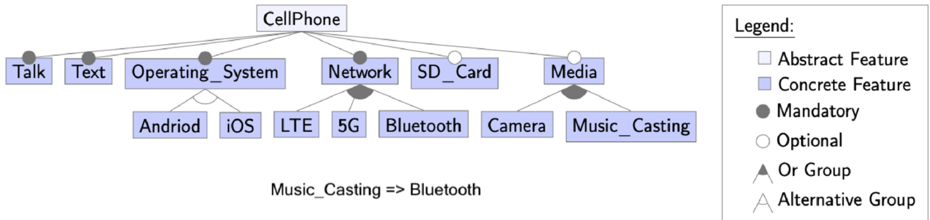


Fig. 3 An example feature model for a product line of cell phones

our cell phone can not have dual operating systems, these features are part of an *alternative* group. Some of these features are optional (`SD_Card`), and some are mandatory (`network protocol`). We can also observe that the network feature has three options—`LTE`, `5G`, and `Bluetooth`—in an *OR* group meaning we can have one or several of these feature. There is also a cross-tree constraint in this model where `Music_Casting` requires the `Bluetooth` feature to be included. Each cell phone in their line of products will be a combination of these features.

3.1 Automated Reverse Engineering

While software product line engineering relies heavily on the use of feature models, these are often not kept up to date as a system evolves, or don't exist at all when designing a software product line from existing products (Nadi et al. 2014). Therefore researchers have built techniques to extract constraints from source code (Nadi et al. 2014, 2015; Kenner et al. 2010), or to synthesize feature models from intermediate languages (She et al. 2011; Andersen et al. 2012). There have also been some recent approaches that forgo synthesis, and reverse engineer feature models directly from a set of products using evolutionary algorithms (Lopez-Herrejon et al. 2012, 2015) and multi-objective algorithms (Assunção et al. 2017; Thianniwet 2016). In our own recent work, we have built the Software Product Line Reverse Engineering Optimization framework, SPLRevO (Thianniwet and Cohen 2015). SPLRevO uses a genetic algorithm to search for a feature model that closely describes an existing set of products. It is based on an earlier framework by Lopez-Herrejon et al. (2012) and Lopez-Herrejon et al. (2015), ETHOM, An Evolutionary Algorithm for Optimized Feature Models. SPLRevO provides an improved fitness function which optimizes the number of products matched to the existing software system, and supports more complex cross-tree constraints in its representation than ETHOM. These methods can extract feature information from different starting points such as through the use of an existing set of products (Lopez-Herrejon et al. 2012, 2015; Thianniwet 2016; Thianniwet and Cohen 2015; Assunção et al. 2017), from plugins (Acher et al. 2011), or from source code (Nadi et al. 2014; Kenner et al. 2010; Thianniwet 2016; Assunção et al. 2017). The fitness functions for reverse engineering vary between tools, but are based on the correctness of representing the final set of products for the software product line by some measure that compares how well the feature model represents only the valid products (those which are defined by the feature model and its constraints). We present a single fitness function used for our study in Section 8.

Another trend has been to focus on large, open-source product lines (Lotufo et al. 2010; Montalvillo and Díaz 2015; Sincero et al. 2007) and the concept of a software ecosystem, where the community modifies and customizes product lines using a common platform and

look (Plakidas et al. 2016). It is possible to reverse engineer feature models from large, open-source projects such as the Linux, Debian, and FreeBSD kernels (Galindo et al. 2010; She et al. 2011). By extracting feature dependencies and descriptions from the code bases and documentation, engineers can determine feature groups, mandatory features, constraints, and invalid configurations based on packages which cannot be installed (dead features).

3.2 Software Product Lines in Other Domains

Software product line engineering was originally restricted to commercial software development, where SPLs' assets could be well managed and planned. However, in recent years SPL engineering practices have moved into the broader software engineering community, and these are being applied to many types of non-commercial highly-configurable software systems such as the Linux Kernel, `gcc`, and `Firefox` (Swanson et al. 2014; Lotufo et al. 2010).

Further still, product line engineering has recently been applied in many emerging domains including smart homes (Cetina et al. 2009), drone systems (Cleland-Huang et al. 2018), nanodevices (Lutz et al. 2012), and Internet of Things (IoT) devices (Ayala et al. 2015; Tzeremes and Gomaa 2018; Quinton et al. 2012). There is also a push towards open-source product lines (Sincero et al. 2007; Lotufo et al. 2010; Montalvillo and Díaz 2015) and the concept of a software ecosystem, where the community modifies and customizes product lines using a common platform and look (Plakidas et al. 2016). Lutz et al. (2012) studied a family of DNA nanodevices. They look at DNA and study chemical reaction networks (CRNs), rather than a living synthetically engineered organism as we do in this work. Their line of organic programming leverages CRNs, which are sets of concurrent equations that can be compiled into single strands of DNA (Winfree 1995). This work is complementary to ours.

3.3 Other 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 (Benavides et al. 2010; Thüm et al. 2014a). While, it is possible to represent variability and perform configuration without the use of a software product line, (e.g. see (Hubaux et al. 2012)), in this work we chose software product line engineering as our representation, since be believe the existence of a visual feature model is an important construct for our intended users. As discussed in (Firestone and Cohen 2018), the synthetic biologist may find applying traditional software engineering techniques challenging.

Ours is not the first analysis of the BioBricks repository. Valverde et al. (2016) examined the relationships within the repository from a network perspective to gain an understanding of the software complexity, and they also consider it to be a software ecosystem. Our work has been inspired by all of the related work to 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 as a way to guide synthetic biologists throughout the design-built-test cycle.

4 Organic Software Product Lines (OSPLs)

In this section we present the notion of an organic software product line which merges synthetic biology and software product line engineering. We note that it was not too long

ago that the software product line community asked whether 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 (Lotufo et al. 2010; Sincero et al. 2007). 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 (2002) state, the output of domain engineering should contain (1) a product line scope, (2) a set of core assets, and (3) a production plan. 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 on domain engineering, however we do touch upon application engineering in our third research question.

4.1 Assets

To begin, we need to identify what constitutes a core asset for this domain. Assets in traditional software 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.

First, the synthetic biology open language—SBOL—(or a similar representation) is used to define the functionality of a snippet of DNA code. This serves as an important design 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 Fig. 4. It is composed of four basic DNA parts, each represented with a unique glyph: the promoter, ribosome binding site, coding sequence, and terminator.

The DNA sequence is also a 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 (Firestone and Cohen 2018) and higher level system architecture such as GenoCAD (Cai et al. 2010).

4.2 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. They also define the common and

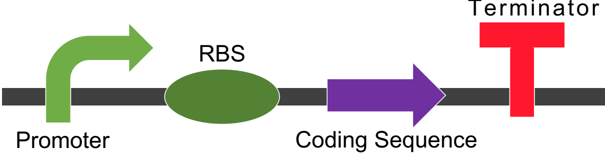


Fig. 4 SBOL model of a transcription unit. This composite part is composed of four basic DNA parts: The promoter (also called a regulator), ribosome binding site (RBS), coding sequence, and terminator

variable features and their relationships. An example of commonality is the transcription unit (Fig. 4). 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. We define a feature as a DNA part as it is the primitive element (basic unit) in synthetic biology.

4.3 Application Engineering

In this context, application engineering involves combining the expected parts using standard DNA cloning techniques for insertion into a living organism (Quan and Tian 2009). The synthetic DNA sequences are typically built into a circular strand of DNA called a plasmid. This plasmid is inserted into the living host where its sequence will integrate with the organism's core DNA and begin transcription and translation which mimics compilation of code. 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.

5 A Roadmap for Evaluating Organic Software Product Lines

In the following sections we evaluate our ability to apply these concepts to an existing DNA repository and a set of commonly engineered biological functions. We split this into three separate studies of increasing rigor. Figure 5 shows the layout of our study. The first study (Section 6) is a feasibility study asking about the potential for reuse and the existence of both commonality and variability in this repository—a necessary conjecture for the definition of OSPLs. We then build four feature models from this repository, each for a different engineered function. Our second study (Section 7) presents a case study on an existing synthetic biology project. We ask how common SPL analyses can provide benefit to end users for typical tasks such as reasoning about the size of the product line and which products to test. We then obtain informal feedback from several synthetic biologists on these results. Our last study (Section 8) is an empirical evaluation of the effectiveness and efficiency of automated reverse engineering. In order for OSPLs to be adopted in practice, automated reverse engineering is likely to be a key tool for aiding synthetic biologists who are not

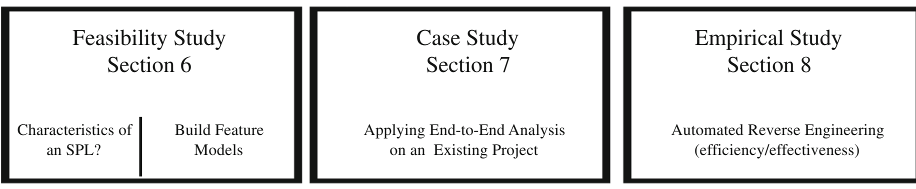


Fig. 5 Roadmap of our evaluation. We present three studies of increasing rigor. The first is a feasibility study with 2 research questions. The second is a case study on an existing project. The last is an empirical study of automated reverse engineering

necessarily experts at feature modeling. Supplemental data for each of our studies along with the reverse engineering tool used in our last study can be found on our website.[1]

6 Feasibility Study

Our first study evaluates the feasibility of organic software products lines (OSPLs) as described in Section 4. In this feasibility study we ask two questions:

- RQ1: What characteristics of a DNA repository are consistent with that of a software product line?
- RQ2: What are the characteristics of feature models built from a DNA repository?

6.1 RQ1: What Characteristics of a DNA Repository are Consistent with that of a Software Product Line?

We evaluate the required elements of a software product line—assets, and existence of both commonality and variability—as defined in Section 4 for organic software product lines. To demonstrate the feasibility of OSPLs these elements must exist in a DNA repository.

6.1.1 Methodology

We use as our subject DNA repository the *Registry of Standard Biological Parts* (the BioBrick repository) hosted by the international genetically engineered machine (iGEM) competition as described in Section 4 (iGEM Registry 2018). 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). Recall in Section 4 we define a *feature* as a DNA part as it is the primitive element (basic unit) in synthetic biology. In the BioBrick repository these are referred to as *BioBrick* parts. A *product* is the compilation of multiple basic BioBricks that together perform a cohesive function.

To obtain 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 (iGEM API 2018). Each entry contains information such as the part_id, part_name, part_type, uses, and creation_date. The *part_id* is a unique alpha-numerical tag for each part (e.g. *BBa_J23106*). The *part_name* is chosen by the user uploading the part. The *part_type* defines the main functionality of the part such as regulatory, composite, coding, ribosome binding site, and scar site. *Uses* defines how many times a part has been requested by a community user. This can tell us how useful a part is to the community.

To obtain additional assets we perform a manual evaluation of 200 randomly chosen composite parts. Each part's web page in the BioBrick repository can contain several additional assets including the SBOL model, the *ruler* model (an alternative to SBOL), the raw DNA sequence, single/double strand sequence, part description in plain text, and results of experimentation from iGEM teams. We randomly sample 200 composite parts from the repository, and two authors independently identify whether they contained each of these assets. We use two authors to reduce bias of subjectivity in determining whether an asset is present (*e.g.*, how much of a textual description constitutes an asset). The two authors next

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

compare their responses, and any discrepancies are resolved between them, or in discussion with a third author. Out of 1,400 decisions, the two authors initially agreed on 1,340 (95.7%) of the decisions. The criteria of determination for each asset was as follows:

- **SBOL model:** The “Subparts” section contains symbols that have a direct translation into symbols found in the SBOL 2.0 set of glyphs.
- **Ruler model:** The “Ruler” section contains at least one subpart name.
- **DNA sequence:** The “Get part sequence” section returns a non-empty result.
- **Single/Double Strand (aka SS/DS):** The “SS” or “DS” section contains a DNA sequence.
- **Textual description:** The page provides some textual information about the part. The information must be more than could be determined by the name of the part alone.
- **Experimental results:** The page provides any experimental results regarding the use of the part or a direct link to a results page.

6.1.2 Threats to Validity

We discuss several threats to validity of this study and our mitigations. First, with respect to external validity, there is only one DNA repository used in this study, which means our results may not generalize to all open source DNA systems. However, we chose the largest open source repository. A second threat to external validity is that we chose only 200 parts to manually analyze for assets and this is only a small percentage of the complete set of parts and may not generalize to the full database. However, we performed the part selection randomly, and therefore assume this is a representative sample.

With respect to internal validity, we used the BioBrick API to pull data from the repository and acknowledge there could be mistakes in our queries. However, we have manually validated these, and have provided both the database and the scripts on our external website for others to re-validate. Two authors of this study made the determinations for the presence of assets which could lead to bias. However, the analysis was first performed independently and compared. 95.7% of the decisions matched. A third author was involved when an easy resolution on consensus was not initially reached.

6.1.3 Results

Assets We evaluate the existence and use of the core assets (the code) and additional assets. Starting with the core assets, 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.

To demonstrate the usage of the core assets we look at 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 these results. 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. Of this group, some parts are used more than 100 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.

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

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 (Zhu et al. 2014).

To evaluate the existence of additional assets we study the following assets described in Section 6.1.1: SBOL model, ruler model, DNA sequence, single/double strand (SS/DS), textual description, and experimental results. All of these assets may be useful to a user interested in how a part can fit into their construct. Table 3 displays the results. 79.5% of parts included the SBOL format and 89.5% contain the ruler model. Most of them included the raw DNA sequence (93.5%) and 91.0% have SS or DS representation (we note that if a part has either the SS or DS, they had the other representation as well so we chose to group them together). 89.5% had a basic textual description, and only 20.5% of parts included any additional experimental results.

A characteristic often associated with software product lines in practice is their degree of evolution. To ask if the same characteristic applies to OPSLs we analyze the parts added to the repository over time. Each year on average 2,996 parts are added, and the cumulative number of parts since the start of the repository in 2003 to 2018 can be seen in Fig. 6. We can see the increase in parts follows a linear trend. This demonstrates that the repository is still actively used and new features are continuing to be added. Figure 7 displays the number of each type of part added over time. We can see that the majority of parts (averaged into one group of "Others") are added at a small and stable level. The outlier part types are coding,

Table 2 BioBrick use counts based on number of 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 200 random composite parts

Asset	SBOL model	Ruler model	DNA sequence	SS/ DS	Textual description	Experimental results
% parts	79.5%	89.5%	93.5%	91.0%	89.5%	20.5%

composite, and regulatory parts which increase in abundance over time. Since new parts are added over time, it is reasonable to assume resulting models built from this repository will also evolve. We further discuss this how this may lead to the need for tool support for maintenance of evolving feature models in Section 9.

Variability To examine variability we focus on the transcription unit, the most basic function (see Fig. 4). There are 4,165 promoters, 769 ribosome binding sites (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 Section 7 we examine a real family of products that has 14 common features.

6.1.4 Summary of RQ1

We identified and characterized several characteristics required to build a software product line in the BioBrick repository. The repository contains 47,934 core assets and is increasing

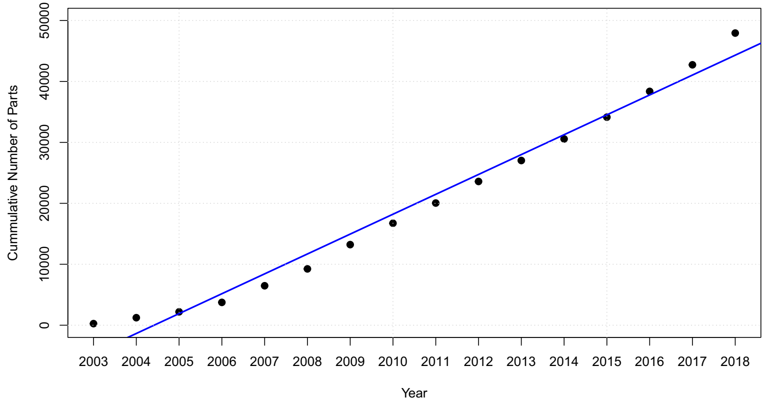


Fig. 6 Cumulative number of parts over time fitted with a linear trend line (R^2 of 0.9813)

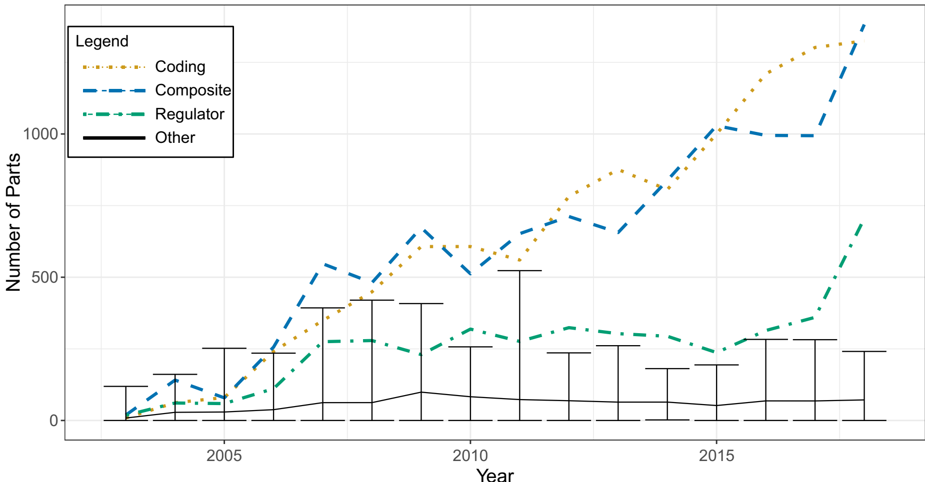


Fig. 7 Number of parts by type added to the BioBrick repository each year. All other part types were averaged into the "Other" category. The error bars represent the minimum and maximum for each year in all other part type categories

at a linear rate with an average of 2,996 parts being added each year. There is a mixture of actively used parts and specialized parts. There is variability in the ways to build the basic building block of any synthetic system (1.7×10^{13} ways to create a transcriptional unit), and there is commonality between products of the same biological function (*e.g.*, sender, receiver, and reporter in cell-to-cell signaling). We identify six additional assets that are present in as many as 93.5% of products and as few as 20.5% of products.

6.2 RQ2: What are the Characteristics of Feature Models Built from a DNA Repository?

While RQ1 evaluates the feasibility of the definition of OSPLs, RQ2 evaluates the implementation of OSPLs. We focus on domain engineering by constructing and evaluating four feature models for distinct functions in the BioBricks repository.

6.2.1 Methodology

We build feature models for three different biological functions: cell-to-cell signaling, kill switch, and viral vectors. The BioBrick repository sorts 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 three functions with the greatest number of parts (biosafety, cell-to-cell signaling, and viral vectors). Biosafety has several subcategories, and we choose the subcategory with the most parts —kill switch. The viral vector function can be split into two main categories: the gene of interest (GOI) and the capsid. We consider these two as separate models. We discuss the detail of the construction of each model next. Since each model required domain knowledge to construct we were restricted to a sample of three functions resulting in four models.

Cell-to-Cell Signaling Cell-to-cell signaling is a key cellular communication mechanism by means of secreting and sensing small molecules such as peptides or proteins between a sender and a receiver. The receiver will then issue some type of response such as fluorescing, expressing a gene, or creating some other chemical signal. For example, quorum sensing is a type of cell-to-cell signaling. When quorum sensing is applied to synthetic biology, two groups of organisms are generally engineered: 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) of signals from the sender. Quorum sensing is used by bacteria for antibiotic production, motility, and biofilm formation (Miller and Bassler 2001).

To build a feature model for cell-to-cell signaling, we forward-engineer the parts listed under the cell-to-cell signaling category of the BioBrick repository. We employ basic knowledge of the structure of a cell-to-cell signaling systems such as the transcription unit and the three main components (sender, receiver, reporter). We then sort the parts based on 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. We also discuss our design with a synthetic biology researcher (also an author of this paper), an iGEM team representative whose project was on a form of cell-to-cell signaling, and several graduate students familiar with the domain knowledge of cell-to-cell signaling (not authors of this paper).

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. A kill switch is usually designed to activate if the host cells escape their intended operating environment. There have been recent improvements for temperature-sensitive kill switches (Stirling et al. 2017) and pH-sensitive kill switches (Stirling et al. 2019). For a good survey of kill switches used in the iGEM competition, see Whitford et al. (2018).

To build the kill switch feature model we manually review all of the wiki pages from the 110 teams who earned a gold medal in the 2017 iGEM competition (Firestone and Cohen 2018). Fourteen teams mention some type of kill switch in their design. We went to these 14 team pages and reviewed their designs for a kill switch by noting the features of their designs (such as the trigger and method of cell death). The exact set of teams is listed on our supplementary website. Our model was also reviewed by a synthetic biology researcher (also an author of this paper).

Viral Vectors Viral vectors are an essential component of gene therapy in which the therapeutic gene of interest is inserted into a transportation vector (called a capsid) to be delivered into a cell (Naso et al. 2017). The gene of interest (GOI) can then replicate and implement its functionality. For example, it can produce insulin in the case of a diabetic host (Levine and Leibowitz 1999). *Adeno-associated virus* (AAV) is a recent popular choice for a transport vector due to its low pathogenicity and minimal recognition by the immune system (Aponte-Ubillus et al. 2018; Naso et al. 2017). AAV can be customized for the desired gene to be transported into the cell and is the subject of our feature models.

The set of viral vector parts in the BioBrick repository is a set of 103 parts that can be used to create a version of the Adeno-associated virus (AAV2). All but one of these parts (which we eliminate from our set of parts) was added to the repository by the 2010 iGEM team from Freiburg (Freiburg Bioware 2010). The Freiburg team created a “Virus Construction Kit” to allow other users to create an AAV2 virus. To build the feature model,

we use the list of parts in the BioBrick repository catalog page, and domain knowledge from the Freiburg iGEM team including their "Virus Construction Kit Manual." In review of the Freiburg team we identified two main components of a viral vector system, the gene of interest (GOI) vector and the capsid vector. The GOI vector represents the gene of interest that will be transported into the host. The capsid vector represents the container that the gene of interest will be transported in. We consider these two separate feature models. We also employ domain knowledge from a synthetic biology researcher (also an author of this paper) and a graduate student with domain knowledge of viral vectors (not an author of this paper).

6.2.2 Threats to Validity

With respect to external validity, we chose to model only four feature models which means our results may not generalize to all biological functions, however we select widely used and diverse functions to model. This helps to ensure our results are as broad as possible. While other functions may result in different models, we believe the basic principles will still apply. For example, the transcription unit will appear in some form in all organic feature models due to biological constraints.

With respect to internal validity, we acknowledge the authors of this paper manually constructed the features models and these have not been used to construct actual products. Hence, we cannot confirm that all of the constraints and features have been captured. However the models were built by two different authors and were discussed with several domain experts (both authors and non-authors of this paper) who agreed these were valid representations of intended functions. We also include the final models as artifacts on our websites for other researchers to evaluate and refine.

6.2.3 Results

We present all four of our reverse engineered models and discusses their characteristics.

Model(1) Cell-to-Cell Signaling Figure 8 shows the overall cell-to-cell signaling feature model we constructed. 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). Note that any numerals appearing below features indicate the number of obfuscated features which will appear if those nodes are expanded.

To compute the number of products in this model we used the dedicated SPL analysis tool FAMA (Benavides et al. 2007) (the model was too large for FeatureIDE to calculate the number of products). As seen in Table 4 the cell-to-cell signaling model has 160 features and 7.5×10^{20} total number of products. The sender and receiver each have 11,448,000 products. The reporter has 5,724,000 products. We describe the components of the feature model next.

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.

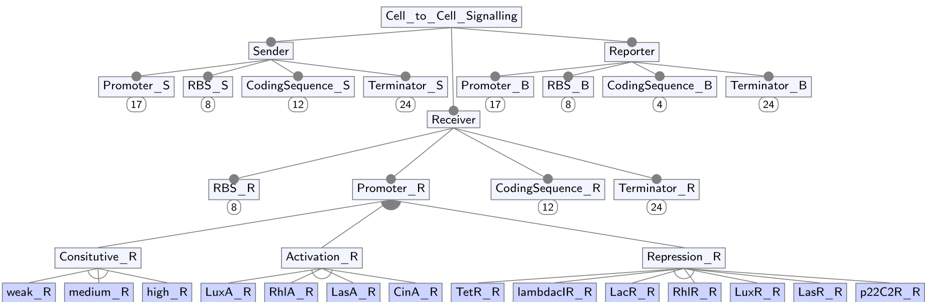


Fig. 8 Top levels of the cell-to-cell signaling feature model

We identified four proteins listed under cell-to-cell signaling promoters. We also 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 Fig. 8.

RBS The next high-level feature is the ribosome binding site (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 in the repository. 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 Fig. 9.

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

Table 4 Summary of feature models

Model	# of Features	# of Products
Cell-to-Cell Signaling	160	7.5×10^{20}
Kill Switch	23	882
Viral Vector—GOI	11	40
Viral Vector—Capsid	15	10

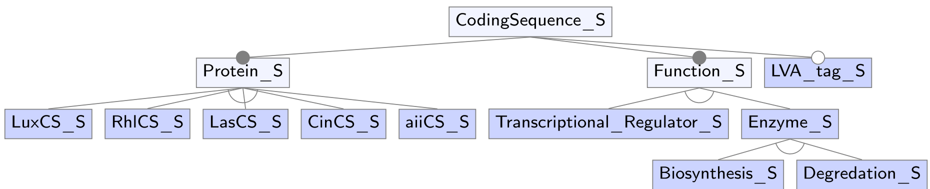


Fig. 9 Sub-feature model of a protein coding sequence

the forward direction, reverse direction, or 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.

Model(2) Kill Switch The constructed feature model is shown in Fig. 10. The *trigger type* (left side) is of particular interest 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. As seen in Table 4 this model has 23 features and represents 882 kill switches.

Models(3 & 4) Viral Vectors The components of building a viral vector are split into two models: the *gene of interest* which holds the gene to be inserted into the organism, and the *capsid* which encases the gene for transportation. We build these two feature models

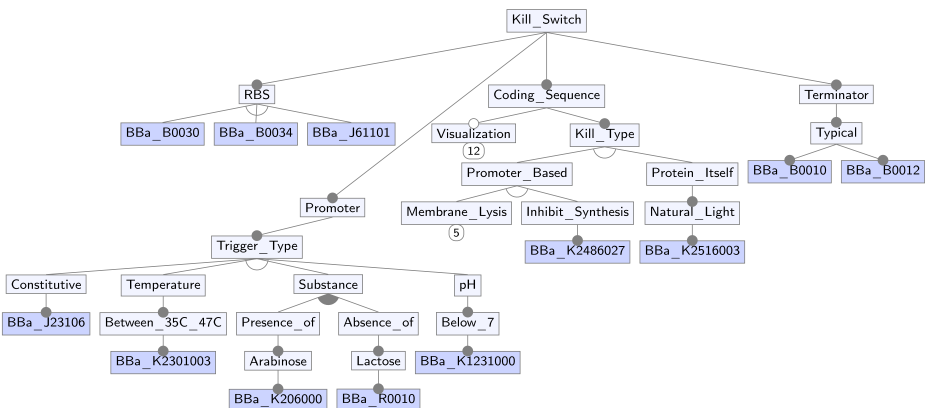


Fig. 10 A feature model for a kill switch. A numeral on a leaf represents how many nodes are in their subtrees (obfuscated)

separately. Figure 11 has the feature model for the gene of interest (GOI) and Fig. 12 has the feature model for the capsid.

The GOI model has 11 features and represents 40 products (Table 4). A product must contain the left and right ITRs, a promoter, and a gene of interest. There are two options for a promoter (pCMV and phTERT) and the gene of interest can either be florescence (two color options) or suicide (three options). There are two optional features (Beta-globin_intron and hGH_Terminator). These features were experimented on by the Freiburg team and were suggested to be used, but remain optional elements.

The capsid model represents 15 features and 10 products (Table 4). There are two promoters required for the REP proteins. The four Rep proteins and three Cap proteins are mandatory. The p5_TATAless promoter is also mandatory. The optional features are the HIS-tag and one of four linkers.

6.2.4 Summary of RQ2

We successfully built four feature models for distinct biological functions with variable and common features. The diversity of the chosen models demonstrates the feasibility of feature models in practice for synthetic engineers, we believe our concepts will extend to other biological functions as well. Table 4 shows summaries of all four models. The total number of products range from 10 to 7.5×10^{20}. Three of these models come from two separate teams of synthetic biology researchers showing how we can apply organic software product line engineering in practice.

6.3 Study Summary

We have demonstrated the feasibility of organic software product lines by showing the existence of core and additional assets, and demonstrating both commonality and variability. We built four feature models for three distinct biological functions showing the feasibility of implementing OSPLs.

7 Case Study: Applying End-to-End Analysis for Developers of Organic Programs

Section 6 demonstrates the feasibility of organic software product lines. In this study we investigate what benefits organic software product line engineering can provide in practice. We evaluate whether OSPL engineering can help us reason about a system while performing different tasks of development which include (a) building and understanding the product space, (b) choosing products to build and test, and (c) communicating domain knowledge to

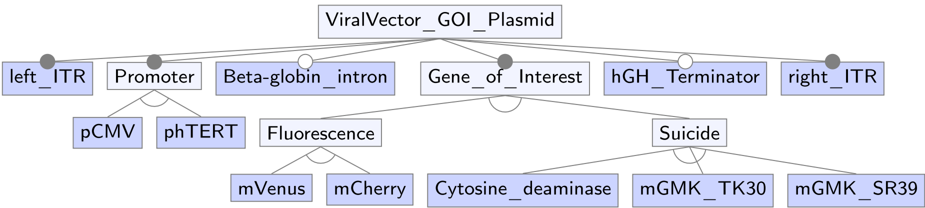


Fig. 11 Viral Vector Gene of Interest (GOI) feature model

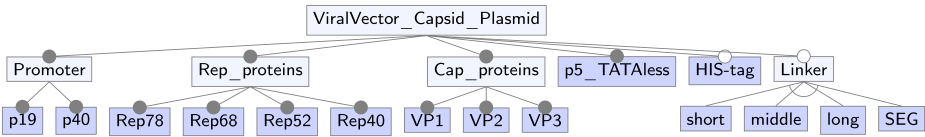


Fig. 12 Viral Vector Capsid feature model

the broader community. We choose to follow an existing research team's project in order to compare their experience with the use of OSPLs, to understand how SPL engineering could have aided their design and subsequent results.

7.1 Methodology

To evaluate the benefits of OSPLs we place ourselves in the role of a synthetic developer by studying the 2017 iGEM team from Arizona State University (ASU). We begin by referring to the team's project web page (Arizona State University 2017). Since their complete set of parts is not available in the repository we contacted the team and were granted permission to view the parts through a cloud-based informatics platform called Benchling. This lab page included the basic parts for each of their products. Their research has since been published and their Benchling link can be found in that article (Tekel et al. 2019).

Recall in quorum sensing, there are two groups of organisms: the first group acts as the sender, and the second as the receiver. The receiver produces one type of protein, and will exhibit a type of behavioral response at a certain concentration (a quorum) signaled by a protein sent by the sender. The two proteins will combine causing the receiver to respond (for example, the receiving organism may turn green).

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 (Arizona State University 2017):

> Knowing the rates of induction also allows for greater precision when designing genetic circuits.

The team's research goal was to experiment on how different combinations of these proteins interact which causes crosstalk. 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).

In this study we perform a *slice* using the slicing functionality in FeatureIDE (Thüm et al. 2014b). We also generate samples of feature model product spaces by using combinatorial interaction testing. For this we use CASA (Garvin et al. 2011).

7.1.1 Threats to Validity

In this case study we examined a single project, quorum sensing. It is possible that other projects may not benefit as much from OSPL engineering, creating a threat to external validity. However, given the wide range of biological functions, and our prior study indicating that many functions have OSPL characteristics, it is likely this work will generalize to other functions. We also used only a single sampling technique to demonstrate the possibility of

reducing the experimental testing space, CIT. However, this is a common technique used to find interactions (the goal of the ASU study).

With respect to internal validity, we acknowledge the authors of this paper built the feature models, however for each subject, we used extensive documentation from the original project and asked external domain experts (both synthetic biology scientists and graduate students with domain expertise in the biological function related to the model) for help validating the models. We used existing tools to perform the slicing and CIT sampling. It is possible that there are faults in these tools, however, we did some manual verification and we provide all of our artifacts on our external website.

7.2 Results

We first present our model for the Arizona State University's project, then present several examples of analysis that could be performed by a research team.

7.2.1 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 13 and 14 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 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 experimental model*.

If we compare this ASU model to the reverse engineered model presented in Section 6.2.3 (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 Fig. 15a. However the actual overlap can be seen in Fig. 15b. 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.

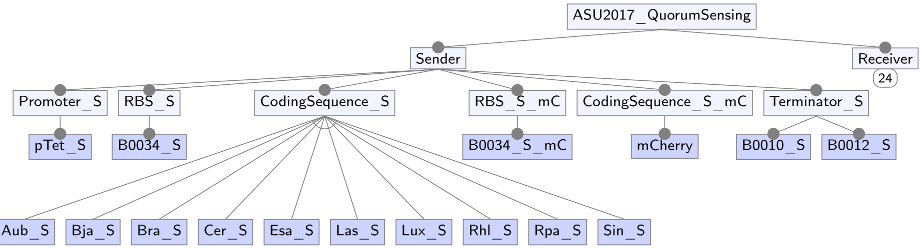


Fig. 13 The sender slice from a feature model for the 2017 Arizona State University's iGEM project (ASU Model)

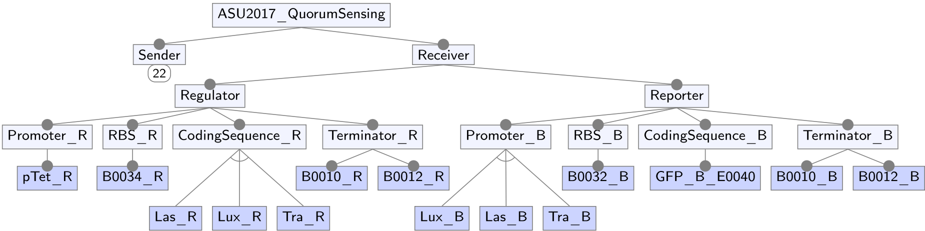


Fig. 14 The receiver slice from a feature model for the 2017 Arizona State University's iGEM project (ASU Model)

To understand why there is such a small overlap we examine the features each model considers. The ASU team's experimental focus is on the interactions of protein features for the sender, regulator, and reporter, so the points of variation were chosen on these three variables. We would expect 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. Ideally a complete set of products would be available, however this may not always be possible. For example, two engineers might have two different sets of domain knowledge resulting in two different (but both correct) models of the system. This demonstrates the need for collaborative modeling tools (Kuiter et al. 2019) to allow users to merge domain knowledge and reconcile multiple models. We further discuss this in Section 9.

7.2.2 Testing and Analysis

We next move to testing and analysis of our applications. Assume the ASU team uses 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 focuses only on the proteins, they could take a *slice* of the cell-to-cell signaling model. This would yield the model in Fig. 16 (called *protein slice*) which

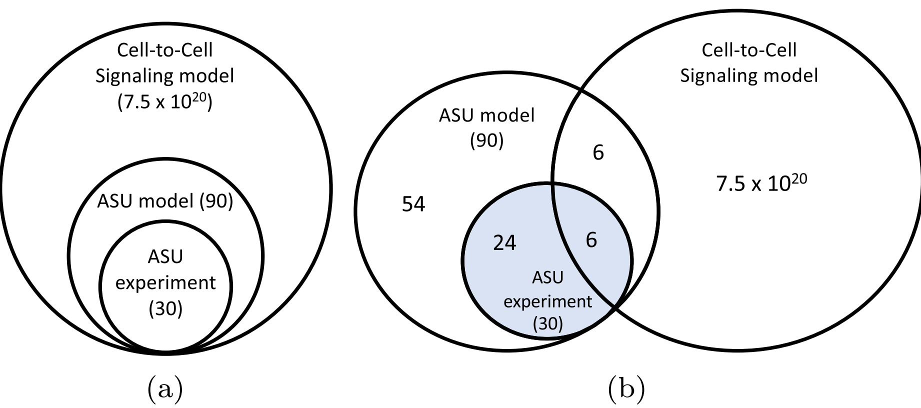


Fig. 15 The overlap of the feature models and ASU experimentation. The sum of each circle is in parentheses

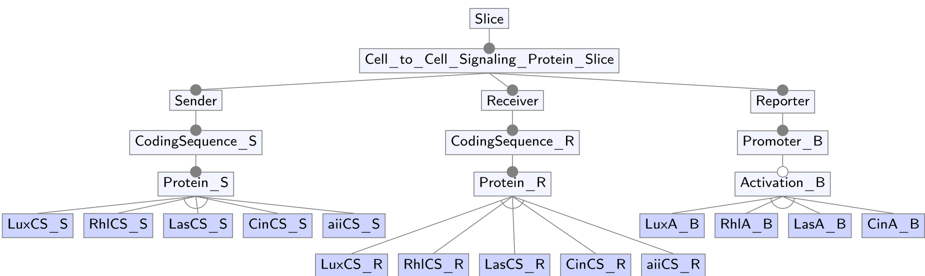


Fig. 16 A slice of the complete cell-to-cell signaling feature model focuses on protein combinations to mimic the ASU team's experiment (protein slice)

represents 100 products. This is significantly fewer than the total product space (7.5×10^{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 (Cohen et al. 1997). Using sampling allows us to test a larger space of combinations. We used CASA (Garvin et al. 2011) 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.

We also built a 2-way CIT sample for the cell-to-cell signaling model, covering all pairs of *all features*. This resulted in 715 tests, a significant (rounding to 100%) reduction of the entire configuration space (7.5×10^{20}). Each of these samples and their product overlaps can be seen in Fig. 17. All samples can be found on our associated website.

Using CIT would provide ASU a more systematic method of approaching the interaction space, leading to a more broad sample. This could reduce possible bias when developing experimental designs.

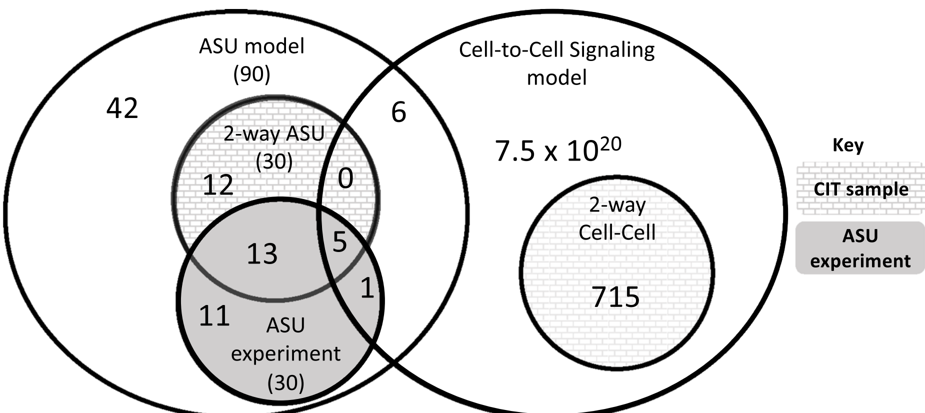


Fig. 17 Product overlap between each of the feature models and possible samples

7.2.3 Domain Expert Feedback

Though we do not perform a user study in this work, we informally requested feedback from several domain experts including a synthetic biology researcher and iGEM team advisor (also an author of this paper), two iGEM teams (the same teams used as inspiration to build the quorum sensors model and the viral vector models), and two graduate students working in synthetic biology (not authors of this paper).

Comments included: “it would be very helpful for new scientists or beginning iGEM teams to understand how different components can be combined to create a working product”, and “very helpful for students on their first day” to learn the required domain knowledge, but also later on when the teams need to locate specific parts to user. One iGEM team representative said they “could see the models being useful for future teams that are interested in assembling a biosensor circuit” (regarding the quorum sensor model), but that the current models are limited to specific interests. They suggested the use would increase if we broadened the use to “other biosensor transcription factors (receivers) or synthases (senders) of other molecules”. It was also mentioned incorporating additional part information “such as promoter strength, inducibility, etc, then it would be even more powerful as a tool”. Other feedback also echoed this by discussing other compatibility issues that could arise when combining multiple plasmids or under different organism hosts. We discuss this extension in the form of annotations in our future work. We also received feedback that the iGEM BioBrick repository is difficult to navigate due to the lack of categorization and standardization, thus it was commented that “any tool that could help iGEM teams better visualize or design their parts would greatly benefit the community”. We plan to conduct a user study as future work, but this informal feedback demonstrates the potential benefit to synthetic biology research teams in practice.

7.2.4 Case Study Summary

We have demonstrated application engineering by placing ourselves in the role of a synthetic biology research team and designing a feature model representative of their project. We found that software product line methodologies can aid researchers designing a new project domain space or utilizing existing models benefiting from existing domain knowledge. In comparing the ASU experimental model to the full cell-to-cell signaling model from our feasibility study, we see there is discrepancy in domain knowledge between the entire repository and the iGEM team. This signals the potential benefits of SPL engineering. We further highlight this potential by demonstrating the use of slicing and sampling to provide a systematic way to more broadly sample the product space. Finally we provide informal feedback from a variety of domain experts and synthetic biology researchers describing their perceived benefits of the models.

8 Empirical Study—Effectiveness and Efficiency of Automated Reverse Engineering

We now present an empirical evaluation of the effectiveness and efficiency of automated reverse engineering in the domain of organic software product lines. Having an automated way to construct the feature model artifact based on a given set of products, provides the last piece of the puzzle for synthetic biologists. Reverse engineering does not require biologists to learn fundamentals of modeling and designing an SPL from the top down, while still

allowing them to use (and potentially refine) this important artifact. In order for reverse engineering to be practical and beneficial, it must be both correct (effective) and efficient. Thus we ask the following research question:

- RQ1: What is the effectiveness and efficiency of automated reverse engineering?

8.1 Methodology

We utilize an existing reverse engineering tool, SPLRevO developed by Thianniwet and Cohen (2015, 2016), although other similar tools could 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 represent the software product line. SPLRevO uses a genetic algorithm to automatically build a feature model that represents all products. The fitness function (described below) aims to maximize the coverage of the set of desired products while minimizing any undesired (additional) products using a penalty.

8.1.1 Objects of Study

We reverse engineer four models from the prior studies: *Arizona State University experimental (ASU)*, *kill switch*, *viral vector gene of interest (GOI)*, and *viral vector capsid*. By choosing diverse subjects we reduce threats to selection bias. Since we have a known oracle for the first two subjects we can compare results to manually constructed models. The latter two subjects allow us to investigate how effective automated reverse engineering starting from an incomplete model. The current prototype of SPLRevO we use for this experiment can handle up to 27 features when reverse engineering from products.[2] 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. In order to satisfy computational resources, we removed the six features under the visualization branch. We chose this branch because it is optional with respect to the functionality of a kill switch. That reduced the model to 12 features and 105 products to use as inputs to SPLRevO. We run all of our experiments 100 times except for our largest model (ASU) for which we were only able to complete 40 runs due to its long runtimes.

8.1.2 Independent Variable

We perform two types of reverse engineering: (1) from a set of complete products and (2) from an incomplete set of products. We use this differentiation as the independent variable. Reverse engineering from a complete set of products can be performed during system maintenance, or when there is a domain expert who can generate a set of products representing the entire product line to use as a predefined oracle. If we can enumerate all products, the fitness function is optimal when the feature model matches the original set of products exactly. An example is the set of 90 quorum sensors from ASU. However, in practice, we often do not have the full set of products and/or it may be too large to enumerate. In this scenario, we can use an incomplete set of products. We do not know the optimal fitness, but

[2]There are similar limitations with other SPL reverse engineering tools

approximate it by trying to get as close to the original set of products as possible. This type of reverse engineering can be used to approximate a good starting point for a feature model.

To create a complete set of products (the known oracle) we generate the set of products for the ASU and kill switch directly from the feature models in our feasibility study (Section 6.1.3). We generate the incomplete set of products for the viral vector models from the following methodology. We began by obtaining all 102 parts in the BioBrick repository under the category of viral vectors (iGEM Viral Vectors 2018). We then partitioned the parts into parts associated with the gene of interest (GOI) vector (20 parts) and the capsid vector (82 parts). We use the *ruler* model (an alternative to SBOL and one of our assets in Section 6.1.1) to identify the main components of each part to use as the features of the products. The catalog list contains both products and basic parts, we discuss implications of this in Section 8.2.1. We eliminated basic parts since these cannot be considered products, and used the remaining parts as our input product list to SPLRevO. For the GOI set of products we removed any part that had three or fewer basic parts, leaving us with 11 unique products. For the capsid vector we removed all parts that did not include a type of promoter and at least one other part, resulting in 18 unique products. The full set of products including the basic parts can be found on our supplementary website.

8.1.3 Dependent Variables

To evaluate effectiveness of the reverse engineered models we utilize the following metrics: precision, recall, validity, and F-measure. We refer to the products used as input to SPLRevO as the $in_products$ and the products represented by the reverse engineered model as the $out_products$. The precision is the ratio of matched products to all output products (1). *Precision* is a real value ranging from 0 to 100 where 100.0% precision occurs when the engineered model has no additional products. The complementary metric *recall* is the ratio of all matched products to all input products (2). Recall is a real value ranging from 0 to 100 where 100% recall occurs when the model covers all input products. The F-measure is the weighted harmonic mean of precision and recall (3). Intuitively, we need the model that covers the most input products but also has less additional products. Thianniwet and Cohen (2015) show that the validity fitness function in SPLRevO performs the best to reverse engineering feature models.

$$Precision = \frac{in_products \cap out_products}{out_products} \tag{1}$$

$$Recall = \frac{in_products \cap out_products}{in_products} \tag{2}$$

$$F\text{-}Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3}$$

$$Validity = \frac{\log_2(in_products \cap out_products + 1) - \alpha \cdot \log_2(out_products - (in_products \cap out_products) + 1)}{\log_2(out_products + 1)} \tag{4}$$

The fitness function increases when the input products are valid, and decreases when products not in the input set are considered valid by the feature model. 100% validity represents the optimal model (covers all input products, no more and no less). The formal definition of validity can be seen in (4). We use the default value for the weight of penalty at 10% ($\alpha = 0.1$) to slightly penalizing additional products while gives higher priority to the number of matched products.

To evaluate the efficiency we utilize the runtime of SPLRevO. This evaluation assesses the practicality of automated reverse engineering in practice. To compute runtime we use `long java.lang.management.ThreadMXBean. getCurrentThreadCpuTime()` to capture the total CPU time for the current thread in nanoseconds (beginning to end).

8.1.4 Experimental Setup

We ran 200 generations and 100 runs in SPLRevO for the kill switch and viral vector models. Due to the large number of features in the ASU model we use 400 generations and 40 runs. We find the validity plateaued at these settings (full data can be found on our supplementary website). All experiments were run on a high-speedy cluster on a single node constrained to 2GHz with 4 CPUs and 32GB RAM each (HCC 2020). We choose one reverse engineered model from each subject to visually display in the results. We choose the model that obtained the highest validity, in the case of a tie we choose the model with the least number of cross tree constraints. In the case of a further tie, we randomly select one of the models.

8.1.5 Threats to Validity

With respect to external validity we chose only four models to reverse engineer, however, these are representative from our other studies. We also used only a single reverse engineering tool, which is limited to 27 features. We acknowledge that we could have biased the results in our choice of feature merging, however, this was only necessary for one of the models. We also acknowledge that other tools could have been used, however, SPLRevO has been shown to be competitive with existing tools and the other tools have similar limitations with respect to scalability. We provide both our tool and resulting models on our external website.

With respect to internal validity, we acknowledge that the reverse engineering tool may have faults, however, we manually validated the models and provide the resulting models on our website. SPLRevO is a stochastic tool, both with respect to the effectiveness and efficiency. To reduce this threat, we ran it multiple times on each model and report both the averages and standard deviations.

With respect to construct validity, we could have chosen different metrics. However, we use commonly used metrics such as F-measure, recall, and precision which have been used in other papers on reverse engineering. Our complexity metrics are also commonly used elsewhere. We clearly describe the metrics chosen, citing related work in which those metrics were proposed or used.

8.2 Results

We next present our results for the two methods of reverse engineering.

Table 5 contains a summary of results. The models with a complete set of products are on top, followed by the incomplete set of products. For each model we show the number of starting products (In Prod.), the number of features, followed by the reverse engineered number of products, the fitness (validity) and runtime in minutes. We show the average and standard deviation for the last three columns. The average validity for the complete set of products is higher than the incomplete set of products. The ASU model has a validity of 94.37% with a standard deviation of 5.34 and the kill switch has a validity of 98.16% with a standard deviation of 3.50. In contrast the GOI and capsid models' validity ranges

Table 5 Effectiveness and efficiency metrics. Average and standard deviations over all runs of reverse engineering using SPLRevO on all four subject models. The top two subjects were reverse engineered from a complete set of products. The bottom two subjects were reverse engineered from an incomplete set of products

Model	Runs	In Prod.	Features	Products		Validity %		Runtime (min)	
				AVG	SD	AVG	SD	AVG	SD
Complete set of products									
ASU	40	30	26	49.58	24.86	94.37	5.34	17.15 h	3.46 h
KillSw.	100	105	16	114.48	36.57	98.16	3.50	18.53	2.70
Incomplete set of products									
GOI	100	11	11	19.31	11.04	90.44	2.86	1.06	0.29
Capsid	100	22	18	113.18	51.47	83.34	3.18	15.84	3.89

from only 83.34–90.44%. If we look at the number of products, ASU and the kill switch have an average number of products close to the input set although there is a large standard deviation. This large deviation in the number of products is not unexpected since one change in a feature will propagate to changes in many products. In practice, if resources allow, an engineer may run their reverse engineering tool multiple times and choose the best result. The ASU model took the longest at an average of 17.15 hours. This model took longer due to having more features and thus a larger search space, but achieved an average of 94.37% validity within 400 generations. The kill switch three models took on average 18.53 min with a standard deviation of 2.70 min. The runtime is correlated more with the number of features (exponentially with an R^2 of 0.9765) than with the independent variable. The capsid model ran faster than the kill switch, but also has fewer features.

Boxplots for all four effectiveness metrics can be seen in Fig. 18. For all four metrics the complete product reverse engineering is higher than the incomplete product reverse engineering although there is a lot of variance for precision and recall. Across the board, the validity is high and we see little variation. This demonstrates that we can achieve high coverage of the input products when reverse engineering. The kill switch has the highest median precision while the ASU model median precision drops below 75%. A low precision means the tool is generating excessive products (false positives). Examining the data more closely we see that 17 of the 40 runs do achieve 100% precision, while only the first (GOI) incomplete model achieves this precision at least once. With recall we see that all four models have perfect recall in at least some of the run, but the recall for the subjects reverse engineered from complete products is higher on average.

Out of the 17 ASU models that have 100% validity, 10 have the fewest number of cross-tree constraints (1), a randomly chosen representative can be seen in Fig. 19. It was able to provide us with a model that closely resembles the hand-built model and has 100% validity. 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.

For the kill switch model, 73 of the runs finished with 100% validity and 56 of those had zero cross-tree constraints. We randomly chose one to display in Fig. 20. If we

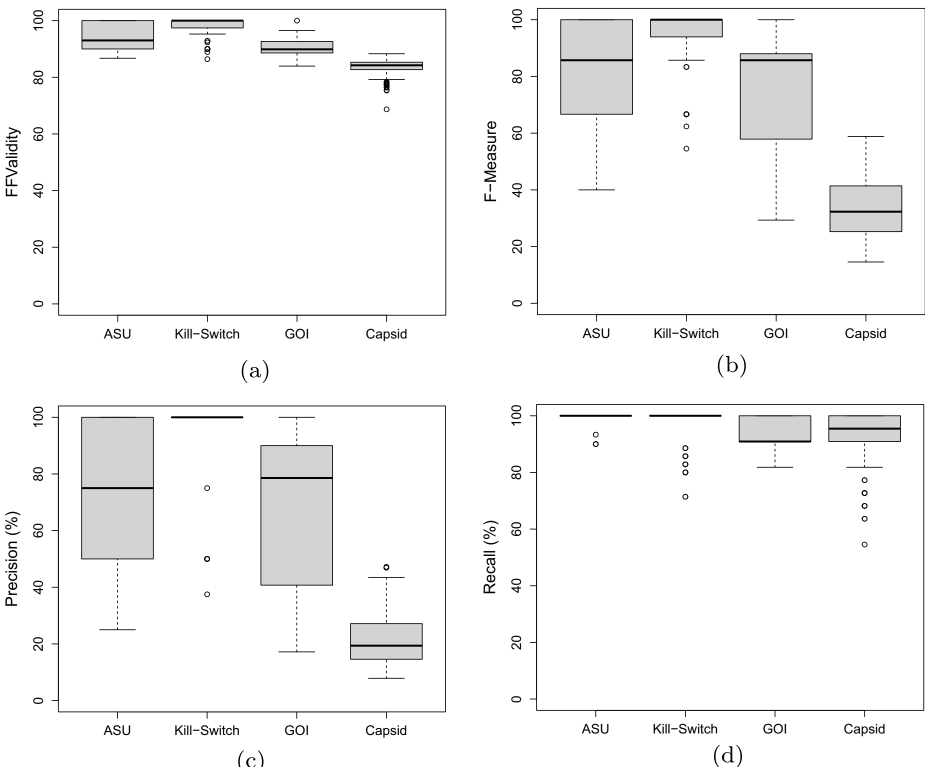


Fig. 18 Effectiveness metrics on all four reverse engineered models over all runs

compare this model to the manually reverse engineered model from Section 6.2, we notice that the parts are grouped in the same manner (promoters, RBS, coding sequence, terminator). However the domain knowledge of the abstract features are missing such as *Trigger Type* and *Visualization* as seen in Fig. 10.

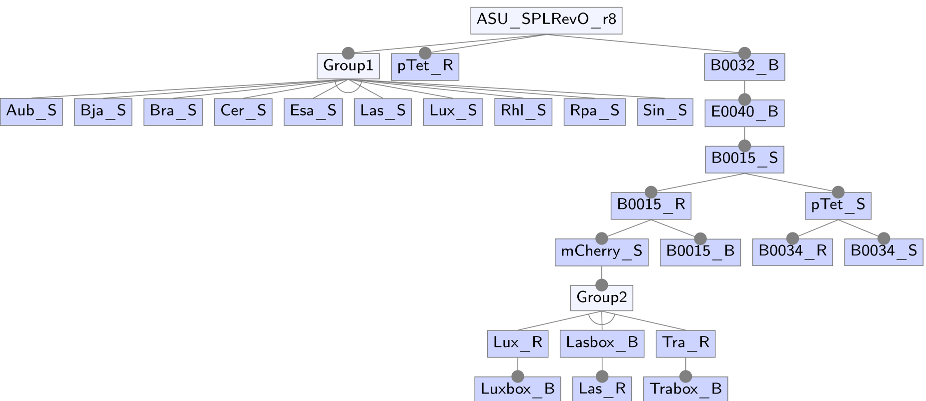


Fig. 19 ASU reverse-engineered feature model using SPLRevO. We note that the one cross-tree constraint created two false optional features so the displayed feature model is simplified

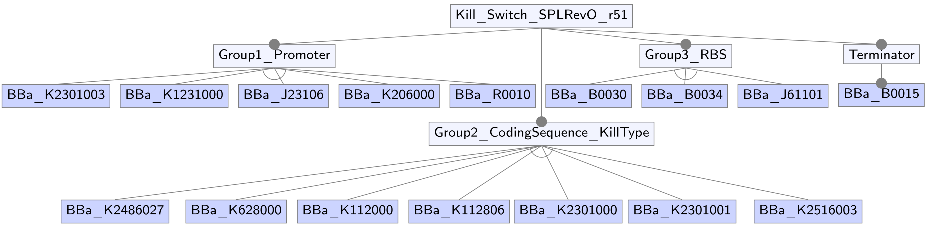


Fig. 20 Kill switch reverse-engineered feature model using SPLRevO

Out of the 100 runs of the GOI model, one has 100% validity representing 11 products which we present in Fig. 21. This automatically engineered model closely matches our manual model where Group2 represents the options for the promoter and Group3 represents the options for the gene of interest. The left and right ITRs are mandatory. The only difference between the models is that the reverse engineered model requires either the Beta-globin intron or the hGH terminator (or both). In contrary, the manual model allows neither to be selected.

The best capsid model out of 100 runs performed with 88.32% validity representing 60 products and can be seen in Fig. 22. Though this model is structurally difficult to interpret, we notice several attributes. Many features appear to be optional such as the VP proteins, HIS-tag, and the linkers. We can also see that if we have the Rep78 protein, the Rep68 and Rep40 are also required. These attributes align with the domain knowledge we learned in RQ2. The rest of the feature model however shows differences.

8.2.1 Further Analysis and Discussion

In order to learn more about the quality of the design, we examine the complexity of the engineered models. We note, that the reverse engineering tool does not explicitly use this metric as part of their search, but it is possible to reduce complexity using a multi-objective approach (Thianniwet 2016). One metric we utilize is the *height* of the feature model which is measured by the longest distance from the root to any leaf. A second metric used by Hemaspaandra and Schnoor (2011) is to count the number of cross-tree constraints. A third metric we use described by Thianniwet (2016) (simply called *complexity*) is the sum of the total number of nodes of the expression tree representation of the feature model (Benavides et al. 2006).

The complexity (Fig. 23a) of the model reverse engineered from the complete products was highest with an average of 420.5 for the ASU model and 272.8 for the kill switch. The

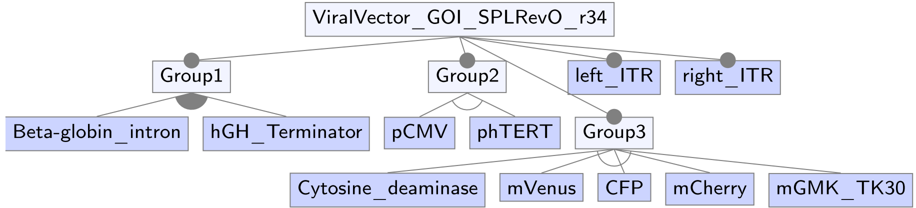


Fig. 21 GOI reverse engineered feature model using SPLRevO

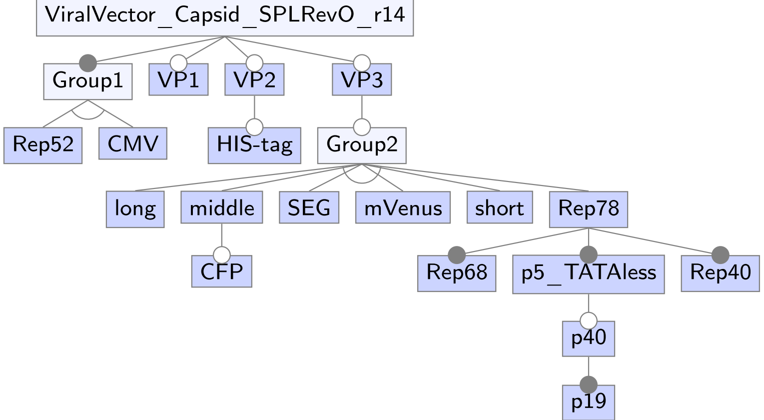


Fig. 22 Capsid reverse engineered feature model using SPLRevO

ASU model also had the highest height (Fig. 23b) of 5.8 where the kill switch had the shortest height (1.4). The complexity measurement of the models reverse engineered from the partial products was 136.5 for the GOI model and 196.1 for the capsid with average heights of 2.6 and 4.6 respectively. Since these models are more broad compared to reverse engineering from a known oracle, it follows they would be less complex. The height however is mixed between the four models not providing us with any real trend (Table 6).

Last, we performed a statistical analysis to determine if we can draw any stronger conclusions between the two groups. We used the Shaprio-Wilk test (Shapiro and Wilk 1965) in R (R Core Team 2013) and determined that the validity, runtime, and complexity do not consistently follow a normal distribution (with $p < 0.05$). Therefore we used the nonparametric Mann-Whitney U test to compare our subjects (Hollander et al. 2013). If we compare the set of models reverse engineered by complete products (ASU and kill switch) with the model reverse engineered by partial products (GOI and capsid) we find the validity, runtime, and complexity are all statistically different ($p < 10^{-16}$). However the subjects themselves are a confounding variable. In a pair-wise Mann-Whitney U test among all four subjects we find statistically differences in the validity, runtime, and complexity. Due to the number of subjects and diversity of these models, we can not isolate the difference between reverse

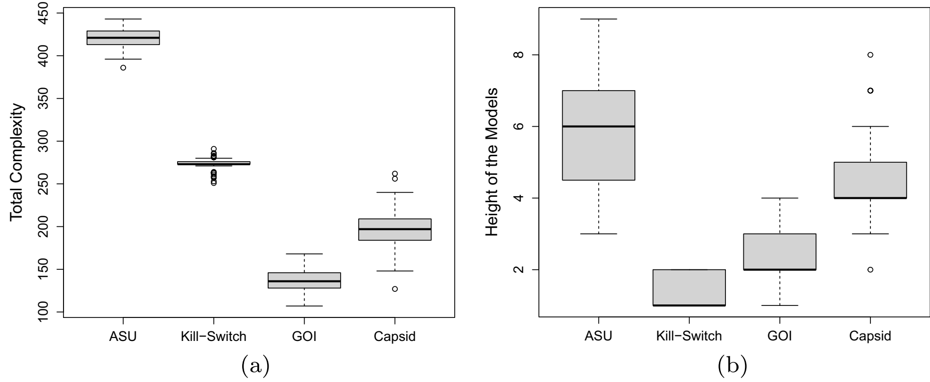


Fig. 23 Complexity metrics on all four reverse engineered models over all runs

Table 6 Complexity metrics. Average and standard deviations over all runs of reverse engineering using SPLRevO on all four subject models. The top two subjects were reverse engineered from a complete set of products. The bottom two subjects were reverse engineered from an incomplete set of products

Model	Complexity		Height		# CTC	
	AVG	SD	AVG	SD	AVG	SD
ASU	420.5	11.6	5.8	1.7	2.8	2.0
KillSwitch	272.8	6.3	1.4	0.5	0.6	1.0
GOI	136.5	12.8	2.6	0.9	3.6	2.3
Capsid	196.1	21.8	4.6	1.2	3.7	2.4

engineering from complete and partial products. We leave this type of a study as future work where we can build a larger sample set and a new experimental design (see Section 10).

8.2.2 Summary of Empirical Study

We performed reverse engineering for two use cases, in the presence of a complete set of products and in the case of having only a partial set of products. In the case of a known oracle 42.5% of runs achieved 100% validity in the ASU model and 72.7% for the kill switch. In the absence of domain knowledge we were able to reverse engineer one model with 100% validity (average of 90.44%) and the best capsid model had 88.32% validity (average of 83.34%). Three of the four models finished in less than 20 min with the exception of the ASU model which took on average 17.15 h due to its large size. This is a reasonable one-time cost to construct an initial model, though since we found the runtime to be exponentially correlated with the number of features scalability may become an issue as the number of features increases. We discuss further in Section 9. All four models were statistically different with respect to the respective metrics, therefore we were unable to draw strong conclusions based on our independent variables.

While the models starting from a complete set of products produced models with higher average validity, we believe there is still an opportunity to reverse engineer from a smaller set of products. This can be used early in the design phase when the user is unsure of the structure of the final software product line or when they simply do not want to start from scratch (as we have done). If the user provides a set of products, but they are not complete (such as the viral vector models), relaxation could be used to allow them to interactively improve their model perhaps with interactive reverse engineering tool support. We believe it would be an interesting line of future work to develop tools that can build such *incomplete feature models*.

9 Implications: the Future of OSPL Engineering

In our evaluation of organic software product lines (OSPLs), we discovered several challenging characteristics that we believe help to provide a roadmap for the continued extension of SPL engineering research. In the following discussion, we highlight the key challenges

that have emerged, and highlight specific areas of future SPL engineering research and may aid the future design of OPSLs specifically. We also discuss how this might extend to other emerging software product lines such as other open-source applications, IoT, robotics, and more.

9.1 Importance of Tools Supporting SPL Evolution

Most software systems have a cycle of evolution and maintenance. Software product lines, therefore, are dynamic and require modifications over time. This is especially evident in open-source product lines. For example, Montalvillo and Díaz (2015) present a feature model that is updated as commits are made. Likewise, we see in our feasibility study (Section 6) that the number of parts in the repository is increasing at a linear rate. This means new features are being added each year, calling for new feature models over time. Though we leave a formal evolutionary study as future work, we do observe several different transformations that can occur. In any evolving product line it may be necessary to have tools to modify the models and keep track of version history. In an open-source model, collaborative feature modeling tools will be needed to allow multiple users to contribute their domain knowledge and refine the models over time. As such some recent work has emerged to formalize evolutionary operations on SPLs and their feature models (Mitschke and Eichberg 2008; Nieke et al. 2016; Ananieva et al. 2019; Hinterreiter et al. 2019). For example temporal feature models (Hinterreiter et al. 2019; Nieke et al. 2016) develop a first-class notation for evolution in a feature model. However, tool support for evolution is still minimal (Marques et al. 2019).

There has been a recent increase in tool development such as by Kuiter et al. (2019), where users can view and modify feature models dynamically. In the domain of OSPL, we see the importance of both collaborative feature modeling and evolution being included in tool development.

9.2 Importance of Constructs that Support Duplicate Features

We observed two types of duplication in parts across these models. In one situation we have the same part appear multiple times in the model (and can appear multiple times within one configuration). For example, in the ASU model seen in Fig. 14, the same proteins (Las, Lux, and Tra) can be used in both the regulator and the reporter. In another situation we see complete branches of the model duplicated. For example, in the same ASU model we see the branch for the Terminator_R and Terminator_B map to identical DNA parts.

In this study we choose to handle it by creating a new feature for each duplicate (cloning), and appending an identifier. For example in the case of duplicated proteins (e.g. Las) in the regulator and reporter we had the features Las_R and Las_B. Though this does work, we lose the information that both these features are functionally identical. We see the need for future work in alternative model abstractions to allows designs that may have duplicated features or other architectural elements without losing potentially valuable information.

It is possible that *multi product lines* could be beneficial (Damiani et al. 2019; Holl et al. 2012; Trujillo-Tzanahua et al. 2018) to help with this representation. A multi product line breaks down a large-scale system into a set of several self-contained and independent product lines. Concepts such as multi-level feature trees making use of mappings to a reference model (Reiser and Weber 2006) and cloned features may also apply to OSPLs.

9.3 Need for More Scalable Reverse Engineering

We see a need for increased scalability of reverse engineering tools. We had to reduce both the ASU and the kill switch model to be able to fit the scope of SPLRevO. Thianniwet and Cohen (2016) demonstrated that if we can use a set of constraints instead of products, reverse engineering can increase in scalability. Even their work however, was limited to approximately 100 features. In general, we need more scalable reverse engineering approaches. We also need ways to easily obtain sets of constraints, without building a feature model. We suggest the use of a domain specific language to help the domain experts interface with product line engineering. We believe all of these topics are interesting avenues for future work and will impact both organic and traditional software product lines.

9.4 Incomplete Feature Models

We explored the idea of using an incomplete set of products as inputs to reverse engineering. In the case of the viral vector models we used the products directly from the catalog. However, some of these parts may be only parts of complete products (such as only one half of a transcriptional unit). There was a notable decrease in the validity of the feature models that were reverse engineered with these incomplete products. Having incomplete products intuitively leads to greater variability than in reality. Thus we may not want to optimize for 100% validity in these cases. We believe it is an interesting direction to study both the effect that partial products have on the performance of automated reverse engineering, and on alternative algorithms for constructing these models. For example, if we have *a priori* information on the presence of partial products we could better account for the increase in variability. We also see incomplete feature models as a starting point for *interactive feature modeling*. If we have domain experts who lack SPL expertise, the idea of starting with an incomplete model could allow them to refine and specify the true feature model they had in mind.

9.5 Towards a Domain Specific Language

Last, we see 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. These tools are orthogonal to the feature model. As suggested by (Hubaux et al. 2012) domain specific languages and feature modeling can be used together to provide a rich SPL engineering environment.

In software product line engineering, one method of describing variability within C code is by use of `ifdefs`. Each `ifdef` can represent a feature that will either be compiled or not depending on some programmatic conditional. Similarly, we can think of segments of DNA (contained in parts) as `ifdefs`. These parts will be implemented (or not) depending on conditionals that are based on the biological relevance of the parts. As a small example, if we represented the variability from our example cell-to-cell signaling FM (Fig. 1) as `ifdefs`, it might look like the pseudocode in Listing 1. We see functions defined for both the sender and receiver which are mandatory. The sender contains an ifdef for HIGH_EXPRESSION. If HIGH_EXPRESSION is defined then we would choose to use the part that encodes for Protein_A, otherwise we would use Protein_B. Similarly, the receiver checks whether

```
//Choose sender protein
Sender(){
    #ifdef (HIGH_EXPRESSION)
        Protein_A();
    #else
        Protein_B();
    #endif
}

//Choose receiver protein
Receiver(){
    #ifdef (LOW_SENSITIVITY)
        Protein_M();
    #else
        Protein_N();
    #endif
}

//Choose reporter protein (optional)
#ifdef (REPORTER)
    Reporter(){
        #ifdef (FLUORESCE)
            Protein_X();
        #else
            Protein_Y();
        #endif
    }
#endif
```

Listing 1 Pseudocode defining the variability in a cell-to-cell signaling system

LOW_SENSITIVITY is defined to decide which protein gets compiled into the DNA plasmid. Since the reporter is optional its functionality is only written if REPORTER is defined. Then the choice of protein depends on the definition of FLUORESCE.

There are other related tools such as GenoCAD which defined a grammar for parts in the BioBrick repository (Cai et al. 2010). There has also been recent work on automation of genetic designs (Storch et al. 2019). Combining these types of automated approaches with the design principles of software product lines could be an interesting avenue for future work to provide end users with a domain oriented toolkit. Furthermore, assembly of synthetic DNA sequences can be automated by the use of DNA synthesis methods (Hughes and Ellington 2017; Ma et al. 2012). By combining the design of constructs with a DSL and automated assembly, could lead to a fully automated design process.

10 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; 3) how common SPL techniques can be used to benefit product line development in this domain; and 4) whether we can leverage reverse engineering tools.

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. We also uncovered a set of challenges leading to a roadmap for new research in SPL engineering.

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. To further explore the difference between reverse engineering from complete versus partial products, we plan to design a follow-up study by artificially creating a set of partial products from our set of complete products.

Acknowledgements We would like to thank the Haynes Lab at Emory University (formally Arizona State University) for sharing additional artifacts with us, especially Dr. Karmella A. Haynes and Dr. Stefan Tekel. This work is supported in part by NSF Grant CCF-1901543, National Institute of Justice Grant 2016-R2-CX-0023, and NSF Grant CBET-1805528. This work was also supported by ORNL Post-Doc Educational Investment funds, and funding was provided by the Center for Bioenergy Innovation (CBI) under Contract No. DE-AC05-00OR22725. The Center for Bioenergy Innovation is a U.S. Department of Energy Bioenergy Research Center supported by the Office of Biological and Environmental Research in the DOE Office of Science.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommonshorg/licenses/by/4.0/.

References

Acher M, Cleve A, Collet P, Merle P, Duchien L, Lahire P (2011) Reverse engineering architectural feature models. In: Proceedings of the 5th European conference on software architecture, ECSA. Springer, Berlin, pp 220–235

Ananieva S, Kehrer T, Klare H, Koziolek A, Lönn H, Ramesh S, Burger A, Taentzer G, Westfechtel B (2019) Towards a conceptual model for unifying variability in space and time. In: Proceedings of the 23rd international systems and software product line conference—volume B, SPLC '19, New York, pp 44–48

Andersen N, Czarnecki K, She S, Wąsowski A (2012) Efficient synthesis of feature models. In: Proceedings of the 16th international software product line conference, vol 1, SPLC '12, pp 106–115

Anderson J, Strelkowa N, Stan GB, Douglas T, Savulescu J, Barahona M, Papachristodoulou A (2012) Engineering and ethical perspectives in synthetic biology. EMBO Rep 13(7):584–590

Aponte-Ubillus JJ, Barajas D, Peltier J, Bardliving C, Shamlou P, Gold D (2018) Molecular design for recombinant adeno-associated virus (rAAV) vector production. Appl Microbiol Biotechnol 102:1045–1054

Arizona State University (2017) ASU iGEM 2017: engineering variable regulators for a quorum sensing toolbox. Last Accessed: June 13, 2019

Assunção WKG, Lopez-Herrejon RE, Linsbauer L, Vergilio SR, Egyed A (2017) Multi-objective reverse engineering of variability-safe feature models based on code dependencies of system variants. Empir Softw Eng 22(4):1763–1794

Ayala I, Amor M, Fuentes L, Troya J (2015) A software product line process to develop agents for the IoT. Sensors 15(7):15640–15660

Benavides D, Segura S, Trinidad P, Ruiz-Cortés A (2006) A first step towards a framework for the automated analysis of feature models. Managing Variability for Software Product Lines: Working With Variability Mechanisms 85:86

Benavides D, Segura S, Trinidad P, Ruiz-cortés A (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, pp 129–134

Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later: a literature review. Inf Syst 35(6):615–636

Bereza-Malcolm LT, Mann G, Franks AE (2014) Environmental sensing of heavy metals through whole cell microbial biosensors: a synthetic biology approach. ACS Synth Biol 4(5):535–546

Bornholt J, Lopez R, Carmean DM, Ceze L, Seelig G, Strauss K (2016) A DNA-based archival storage system. ACM SIGARCH Comput Architect News 44(2):637–649

Cai Y, Wilson ML, Peccoud J (2010) GenoCAD for iGEM: a grammatical approach to the design of standard-compliant constructs. Nucl Acids Res 38(8):2637–2644

Cameron DE, Bashor CJ, Collins JJ (2014) A brief history of synthetic biology. Nat Rev Microbiol 12(5):381–390

Cashman M, Firestone J, Cohen MB, Thianniwet T, Niu W (2019) DNA as features: organic software product lines. In: Proceedings of the international systems and software product line conference, SPLC, pp 1–11

Cetina C, Giner P, Fons J, Pelechano V (2009) Using feature models for developing self-configuring smart homes. In: 5th International conference on autonomic and autonomous systems, pp 179–188

Cleland-Huang J, Vierhauser M, Bayley S (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, pp 109–112

Clements P, Northrop L (2002) Software product lines: practices and patterns, Addison-Wesley, Boston

Cohen DM, Dalal SR, Fredman ML, Patton GC (1997) The AETG system: an approach to testing based on combinatorial design. IEEE Trans Softw Eng 23(7):437–444

Damiani F, Lienhardt M, Paolini L (2019) A formal model for multi software product lines. Sci Comput Program 172:203–231. https://doi.org/10.1016/j.scico.2018.11.005

Daniel R, Rubens JR, Sarpeshkar R, Lu TK (2013) Synthetic analog computation in living cells. Nature 497(7451):619–623

Elowitz MB, Leibler S (2000) A synthetic oscillatory network of transcriptional regulators. Nature 403:335–338

Firestone J, Cohen MB (2018) The assurance recipe: facilitating assurance patterns. In: Proceedings of the international conference on computer safety, reliability, and security (SAFECOMP), ASSURE workshop, pp 22–30

Freiburg Bioware (2010) Freiburg bioware iGEM 2010: virus construction kit for therapy. Last Accessed: 6 Nov 2019

Galindo JA, Benavides D, Segura S (2010) Debian packages repositories as software product line models. Towards automated analysis. In: ACoTA, pp 29–34

Gardner TS, Cantor CR, Collins JJ (2000) Construction of a genetic toggle switch in Escherichia coli. Nature 402:339–342

Garvin BJ, Cohen MB, Dwyer MB (2011) Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empir Softw Eng 16(1):61–102

Garvin BJ, Cohen MB, Dwyer MB (2013) Failure avoidance in configurable systems through feature locality. In: Assurances for self-adaptive systems—principles, models, and techniques, vol LNCS 7740. Springer, pp 266–296

HCC (2020) Holland computing center documentation

Hemaspaandra E, Schnoor H (2011) Minimization for generalized boolean formulas. In: Proceedings of the 22nd international joint conference on artificial intelligence, pp 566–571

Hinterreiter D, Nieke M, Linsbauer L, Seidl C, Prähofer H, Grünbacher P (2019) Harmonized temporal feature modeling to uniformly perform, track, analyze, and replay software product line evolution. In: Proceedings of the 18th ACM SIGPLAN international conference on generative programming: concepts and experiences, GPCE 2019, pp 115–128

Holl G, Grünbacher P, Rabiser R (2012) A systematic review and an expert survey on capabilities supporting multi product lines. Inf Softw Technol 54(8):828–852

Hollander M, Wolfe DA, Chicken E (2013) Nonparametric statistical methods, vol 751. Wiley, New York
Hubaux A, Jannach D, Drescher C, Murta L, Männistö T, Czarnecki K, Heymans P, Nguyen T, Zanker M (2012) Unifying software and product configuration: a research roadmap. In: Proceedings of the 2012 international conference on configuration, CONFWS'12, vol 958, pp 31–35
Hughes RA, Ellington AD (2017) Synthetic dna synthesis and assembly: putting the synthetic in synthetic biology. Cold Spring Harbor Perspect Biol 9(1):a023812
iGEM API (2018) Registry of standard biological parts API. iGEM Foundation. Last Accessed: 13 June 2019
iGEM Competition (2018) International genetically engineered machine competition. iGEM Foundation. Last Accessed: 13 June 2019
iGEM Registry (2018) Registry of standard biological parts. iGEM Foundation. Last Accessed: June 13:2019
iGEM Viral Vectors (2018) Viral vectors based on the adeno-associated virus. iGEM Foundation. Last Accessed: 13 June 2019
Kang K, Cohen S, Hess J, Novak W, Peterson A (1990) Feature-oriented domain analysis (foda) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA
Kenner A, Kästner C, Haase S, Leich T (2010) Typechef: toward type checking #ifdef variability in c. In: Proceedings of the 2nd international workshop on feature-oriented software development, pp 25–32
Kis Z, Pereira HS, Homma T, Pedrigi RM, Krams R (2015) Mammalian synthetic biology: emerging medical applications. J R Soc Interface 12(106):1–18
Kuiter E, Krieter S, Krüger J, Leich T, Saake G (2019) Foundations of collaborative, real-time feature modeling. In: Proceedings of the 23rd international systems and software product line conference—volume A, SPLC. ACM, pp 257–264
Levine F, Leibowitz G (1999) Towards gene therapy of diabetes mellitus. Mol Med Today 5(4):165–171
Lopez-Herrejon RE, Galindo JA, Benavides D, Segura S, Egyed A (2012) Reverse engineering feature models with evolutionary algorithms: an exploratory study. In: Fraser G, Teixeira de Souza J (eds) Search based software engineering. Springer, Berlin, pp 168–182
Lopez-Herrejon RE, Linsbauer L, Galindo JA, Parejo JA, Benavides D, Segura S, Egyed A (2015) An assessment of search-based techniques for reverse engineering feature models. J Syst Softw 103:353–369
Lotufo R, She S, Berger T, Czarnecki K, Wkasowski A (2010) Evolution of the linux kernel variability model. In: Proceedings of the 14th international conference on software product lines: going beyond, SPLC, pp 136–150
Lutz RR, Lutz JH, Lathrop JI, Klinge TH, Mathur D, Stull DM, Bergquist TG, Henderson ER (2012) Requirements analysis for a product family of DNA nanodevices. In: Proceedings of the 20th IEEE international requirements engineering conference, RE, pp 211–220
Ma S, Tang N, Tian J (2012) Dna synthesis, assembly and applications in synthetic biology. Curr Opin Chem Biol 16(3–4):260–267
Marques M, Simmonds J, Rossel PO, Bastarrica MC (2019) Software product line evolution: a systematic literature review. Inf Softw Technol 105:190–208
Miller MB, Bassler BL (2001) Quorum sensing in bacteria. Annu Rev Microbiol 55(1):165–199. pMID:11544353
Mitschke R, Eichberg M (2008) Supporting the evolution of software product lines. In: ECMDA traceability workshop (ECMDA-TW), pp 87–96
Montalvillo L, Díaz O (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, pp 111–120
Nadi S, Berger T, Kästner C, Czarnecki K (2014) Mining configuration constraints: static analyses and empirical results. In: Proceedings of the 36th international conference on software engineering, pp 140–151
Nadi S, Berger T, Kästner C, Czarnecki K (2015) Where do configuration constraints stem from? An extraction approach and an empirical study. IEEE Trans Softw Eng 41(8):820–841
Naso MF, Tomkowicz B, 3rd WLP, Strohl WR (2017) Adeno-associated virus (AAV) as a vector for gene therapy. BioDrugs 31:317–334
Nieke M, Seidl C, Schuster S (2016) Guaranteeing configuration validity in evolving software product lines. In: Proceedings of the tenth international workshop on variability modelling of software-intensive systems, VaMoS '16. Association for Computing Machinery, New York, pp 73–80
Nielsen AA, Der BS, Shin J, Vaidyanathan P, Paralanov V, Strychalski EA, Ross D, Densmore D, Voigt CA (2016) Genetic circuit design automation. Science 352(6281):aac7341-1–aac7341-11
Plakidas K, Stevanetic S, Schall D, Ionescu TB, Zdun U (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, pp 89–98

Pohl K, Böckle G, van Der Linden FJ (2005) Software product line engineering: foundations, principles and techniques. Springer Science & Business Media
Quan J, Tian J (2009) Circular polymerase extension cloning of complex gene libraries and pathways. PloS One 4(7):1–6
Quinton C, Rouvoy R, Duchien L (2012) Leveraging feature models to configure virtual appliances. In: Proceedings of the 2nd international workshop on cloud computing platforms, CloudCP. ACM, pp 2:1–2:6
R Core Team (2013) R: a language and environment for statistical computing, R Foundation for Statistical Computing, Vienna. http://www.R-project.org/
Reiser MO, Weber M (2006) Managing highly complex product families with multi-level feature trees. In: 14th IEEE international requirements engineering conference (RE'06). IEEE, pp 149–158
Rossello RA, Kohn DH (2010) Cell communication and tissue engineering. Commun Integr Biol 3(1):53–56
SBOL (2019) Synthetic biology open language. SBOL Research Group. Last Accessed: 13 June 2019
Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3/4):591–611
She S, Lotufo R, Berger T, Wąsowski A, Czarnecki K (2011) Reverse engineering feature models. In: Proceedings of the 33rd international conference on software engineering, ICSE. ACM, pp 461–470
Sincero J, Schirmeier H, Schröder-Preikschat W, Spinczyk O (2007) Is the linux kernel a software product line? In: Proceedings of the 2nd SPLC workshop on open source software and product lines, pp 1–4
Stirling F, Bitzan L, O'Keefe S, Redfield E, Oliver JW, Way J, Silver PA (2017) Rational design of evolutionarily stable microbial kill switches. Mol Cell 68(4):686–697.e3
Stirling F, Naydich A, Bramante J, Barocio R, Certo M, Wellington H, Redfield E, O'Keefe S, Gao S, Cusolito A, Way J, Silver P (2019) Synthetic cassettes for pH-mediated sensing, counting and containment. bioRxiv
Storch M, Haines MC, Baldwin GS (2019) DNA-BOT: a low-cost, automated DNA assembly platform for synthetic biology. bioRxiv
Swanson J, Cohen MB, Dwyer MB, Garvin BJ, Firestone J (2014) Beyond the rainbow: self-adaptive failure avoidance in configurable systems. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp 377–388
Tavella F, Giaretta A, Dooley-Cullinane TM, Conti M, Coffey L, Balasubramaniam S (2018) DNA molecular storage system: transferring digitally encoded information through bacterial nanonetworks. IEEE Transactions on Emerging Topics in Computing 1801.04774
Tekel SJ, Smith CL, Lopez B, Mani A, Connot C, Livingstone X, Haynes KA (2019) Engineered orthogonal quorum sensing systems for synthetic gene regulation in Escherichia coli. Front Bioeng Biotechnol 7(80):1–12
Thianniwet T (2016) SPL-XFactor: a framework for reverse engineering feature models. The University of Nebraska-Lincoln
Thianniwet T, Cohen MB (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, pp 1–16
Thianniwet T, Cohen MB (2016) Scaling up the fitness function for reverse engineering feature models. In: Symposium on search-based software engineering, SSBSE, pp 128–142
Thüm T, Apel S, Kästner C, Schaefer I, Saake G (2014a) A classification and survey of analysis strategies for software product lines. ACM Comput Surv (CSUR) 47(1):1–45
Thüm T, Kästner C, Benduhn F, Meinicke J, Saake G, Leich T (2014b) FeatureIDE: an extensible framework for feature-oriented software development. Sci Comput Program 79:70–85. Experimental Software and Toolkits (EST 4): a special issue of the workshop on academic software development tools and techniques (WASDeTT-3 2010)
Trujillo-Tzanahua GI, Juárez-Martínez U, Aguilar-Lasserre AA, Cortés-Verdín MK (2018) Multiple software product lines: applications and challenges. In: Mejia J, Muñoz M, Rocha Á, Quiñonez Y, Calvo-Manzano J (eds) Trends and applications in software engineering, pp 117–126
Tzeremes V, Gomaa H (2018) A software product line approach to designing end user applications for the internet of things. In: ICSOFT
Valverde S, Porcar M, Peretó J, Solé RV (2016) The software crisis of synthetic biology. bioRxiv
Weber W, Fussenegger M (2012) Emerging biomedical applications of synthetic biology. Nat Rev Genet 13(1):21–35
Weber W, Stelling J, Rimann M, Keller B, Daoud-El Baba M, Weber CC, Aubel D, Fussenegger M (2007) A synthetic time-delay circuit in mammalian cells and mice. Proc Natl Acad Sci USA 104(8):2643–2648

Whitaker WB, Sandoval NR, Bennett RK, Fast AG, Papoutsakis ET (2015) Synthetic methylotrophy: engineering the production of biofuels and chemicals based on the biology of aerobic methanol utilization. Curr Opin Biotechnol 33:165–175

Whitford CM, Dymek S, Kerkhoff D, März C, Schmidt O, Edich M, Droste J, Pucker B, Rückert C, Kalinowski J (2018) Auxotrophy to Xeno-DNA: an exploration of combinatorial mechanisms for a high-fidelity biosafety system for synthetic biology applications. J Biol Eng 12(1):1–28

Winfree E (1995) On the computational power of DNA annealing and ligation. In: DNA based computers

Zhu J, Zhou M, Mockus A (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, pp 1–4

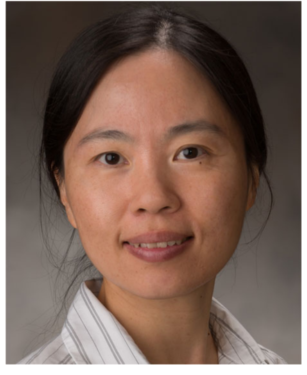
Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Mikaela Cashman is a postdoctoral research associate at Oak Ridge National Laboratory in the Biosciences Division. She received her Ph.D. in Computer Science at Iowa State University and her Masters in Computer Science from the University of Nebraska-Lincoln. Her research centers around the application of software engineering methods to the scientific domain (mainly the Biosciences). Her technical areas of interest include: software testing, machine learning, systems biology, and synthetic biology. She has further interest in bioinformatics workflows and algorithms, high performance computing, and in GPU programming. Her research aspirations focus on building bridges between the computational and biological sciences.

Justin Firestone is an Assistant Professor of Practice in the Jeffrey S. Raikes School of Computer Science and Management and in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln. He also holds a Juris Doctor degree from the University of Nebraska College of Law. While pursuing his Ph.D. in Computer Science, he was a recipient of the GRF-STEM fellowship from the National Institute of Justice. His interdisciplinary research focuses on the intersection of computer science, synthetic biology, and regulatory frameworks, including how traditional software engineering principles could apply to synthetic biology. In addition to teaching core courses at the Raikes School, he also teaches Cyberlaw at the law college.

Myra B. Cohen is a Professor and the Lanh and Oanh Nguyen Chair in Software Engineering in the Department of Computer Science at Iowa State University. She received her Ph.D. from the University of Auckland, New Zealand. Her research interests are in software testing of highly-configurable software, search based software engineering, applications of combinatorial designs, and synergies between systems and synthetic biology, and software engineering. She is the recipient of a National Science Foundation CAREER award, an Air Force Office of Scientific Research Young Investigator Award, and four ACM Distinguished Paper awards. She is an ACM Distinguished Scientist.

Dr. Thammasak Thianniwet is a lecturer and researcher in the School of Information Technology and DIGITECH at Suranaree University of Technology in Thailand. He received the B.Eng and M.Eng degrees in computer engineering from Suranaree University of Technology and the PhD degree in computer science from the University of Nebraska - Lincoln. He is interested in static software analysis, software testing, software product lines, software engineering, data analytics, AI, business intelligence, IoT and smart applications.



Dr. Wei Niu is an associate professor in the Chemical and Biomolecular Engineering Department at University of Nebraska - Lincoln. She received her PhD degree from Michigan State University. Her research focuses on metabolic engineering, protein engineering, and synthetic biology. She is interested in pathway design and strain engineering for the conversion of biomass into value-added products and the implementation of nonnatural amino acids for functional diversification of proteins.

Affiliations

Mikaela Cashman[1,2] · **Justin Firestone**[3] · **Myra B. Cohen**[1] · **Thammasak Thianniwet**[4] · **Wei Niu**[5]

Justin Firestone
jfiresto@cse.unl.edu

Myra B. Cohen
mcohen@iastate.edu

Thammasak Thianniwet
thammasak@sut.ac.th

Wei Niu
wniu2@unl.edu

1 Department of Computer Science, Iowa State University, Ames, IA 50011-1090, USA

2 Biosciences Division, Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA

3 Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588-0115, USA

4 DIGITECH and School of Information Technology, Suranaree University of Technology, Nakhon Ratchasima, Thailand

5 Department of Chemical and Biomolecular Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588-0115, USA