Interpretability of configurable software in the biosciences

by

Mikaela Cashman

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Myra B. Cohen, Major Professor
James Lathrop
Robyn Lutz
Samik Basu
Julie Dickerson

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

DEDICATION

This thesis is dedicated to my husband John and our feline children Leela and Romana.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

I would like to begin my thanking my advisor Dr. Myra Cohen. Had she not dropped what she was doing to come meet me when I visited Nebraska for graduate information day, I would not be here. Her enthusiasm for her research and genuine interest in me as an individual convinced me she was the right fit. I have grown tremendously under her advisement and am grateful for the many lessons I have learned that I will continue to apply throughout my professional and personal life. I would also like to thank Sasha for being part of our recent meeting.

I would like to thank the other members of my committee: Dr. James Lathrop for introducing me to the world of molecular computing and pushing my research to new bounds, and to his mentorship and beautiful cats; Dr. Robyn Lutz for her research inspirations, professional advisement, and personal mentorship; Dr. Samik Basu for his advisement and welcoming me to ISU; and to Dr. Julie Dickerson for being my systems biology guide.

I would like to thank all my colleagues both at the University of Nebraska-Lincoln and at Iowa State University. Thank you to my former lab-mates at ESQuaReD including: Jonathan Saddler, Justin Firestone, Mitch Gerrard, Eric Rizzi, Thammasak Thianniwet, Wayne Motycka, and Brady Garvin. Thank you to my friends Jaycee Han, Tony Schneider, Jennie Cattlet, Zahmeeth Sakkaff. Thank you also to Dr. Suzette Person, Dr. Nicole Buan, and Dr. Matthew Dwyer for assisting in my professional growth and mentorship. A huge thanks to the staff at HCC. Thank you David Swanson for reminding me a Twins fan is never far away.

I would like to thank my colleagues at Iowa State University, especially those who welcomed me into the department. Thank you to Junaid Babar, Chuan Jiang, and Shruti Biswal. I am sad to leave the inaugural member of the LAVA lab including Urjoshi Sinah, Michael Gerten, and Ibrahim Mesecan.

I would like to especially thank Natasha Pavlovikj for not only her friendship but also her role in my success. Our study dates were excellent at keeping me on pace and reminding me to enjoy the little things. She was a constant resource of information and never laughed at my stupid questions. I can not thank her enough for her ongoing support of me as an individual and as a researcher.

I would like to thank my co-authors: Justin Firestone, Thammasak Thianniwet, Wie Niu, Priya Ranjan, and Robert Cottingham. I would like to thank KBase and Robert Cottingham for encouraging my research collaborations. Thank you also to Meghan Drake for your friendship, and Jon White for encouraging me to never stop learning.

I would like to thank my two amazing cats Leela and Romana. They have been by my side through the majority of my entire education, never afraid to inform me when I need to take a break. I am so grateful they have been in my life.

I would like to thank my family. Thank you Grandma for passing on your never ending curiosity. Mom thank you for encouraging me I could do anything, your love and faith in me has been the fuel for this journey. Thank you Adam for your unconditional love and support. Thank you Josh for sharing your passions with me. Thank you Dayna and Laura for bringing the party in my life. To the Cahill family, thank you for always being supportive of my endeavors and encouraging me all these years.

Thank you to the McDevitt family for their love and support of my dreams: Bev, Kevin, Joe, Julie, Jackie, April, Matt, Grant, Drew, Carson, Molly, Emily, Everly, Gavin, Sophie, Layla.

Finally, I would like to give a enormous thank you to my husband John. You have stood by me on every step of this journey from choosing a college and making a big move, to moving again when my lab transferred, and everyday in between. You have been instrumental to my happiness over these years and I am so grateful we took this journey together. Thank you for always believing in me. John — you are so fetch and always bring blue skies — thank you.

This work was supported in part by National Science Foundation grants #CCF-1901543 and #CCF-1745775.

ABSTRACT

Users of bioinformatics software tools range from bench scientists with little computational experience, to sophisticated developers. As the number and types of tools available to this diverse set of users grow, they are also increasing in flexibility. The customization of these tools makes them highly-configurable — where the end user is provided with many customization (configuration) options. At the same time, biologists and chemists are engineering living organisms by programming their DNA in a process that mimics software development. As they share their designs and promote re-use, their programs are also emerging as highly-configurable.

As these bioscience systems become mainstream tools for the biology and bioinformatics communities, their dependability, reliability, and reproducibility becomes critical. Scientists are making decisions and drawing conclusions based on the software they use, and the constructs designed by synthetic biologists are being built into living organisms and used in the real world. Yet there is little help guiding users of bioinformatics tools or those building new synthetic organisms.

As an end user equipped with minimal information, it is hard to predict the effect of changing a particular configuration option, yet the choice of configuration can lead to a large amount of variation in functionality and performance. Even if the configuration options make sense to an expert user, understanding all options and their interactions is difficult or even impossible to compute due to the exponential number of combinations. Similarly, synthetic biologists must choose how to combine small DNA segments. However, there can be millions of ways to combine these pieces, and determining the architecture can require significant domain knowledge.

In this dissertation we address these challenges of interpreting the effects of configurability in two areas in the biosciences: (1) bioinformatics software, and (2) synthetic biology. We highlight the challenges of configurability in these areas and provide approaches to help users navigate their configuration spaces leading to more interpretable configurable software in the biosciences.

First, we demonstrate there is variability in both the functional and performance outcomes of highly-configurable bioinformatics tools, and find previously undetected faults. We discuss the implications of this variability, and provide suggestions for developers. Second, we develop a user-oriented framework to identify the effect of changing configuration options in software, and communicate these effects to the end user in a simplistic format. We demonstrate our framework in a large study and compare to a state of the art method for performance-influence modeling in software.

Last, we define a mapping of software product line engineering to the domain of synthetic biology resulting in organic software product lines. We demonstrate the potential reuse and existence of both commonality and variability in an open source synthetic biology repository. We build feature models for four commonly engineered biological functions and demonstrate how product line engineering can benefit synthetic biologists.

CHAPTER 1. INTRODUCTION

Bioinformatics software is becoming increasingly sophisticated and prevalent in its day-to-day use. Scientists use these tools in their daily research to simulate and validate laboratory experimentation, model natural phenomena, analyze data, and much more. Consider the most widely used bioinformatics tool, the *Basic Local Alignment Search Tool* (BLAST), which searches a database to find similar matches to biological sequences. According to two separate studies analyzing a database of publications from 1990 to 2014 [1], Thomson Reuter found BLAST to be the 12th and 14th (due to two different versions of the tool) most cited paper, and Google Scholar (with a broader database) found BLAST to be the 24th and 26th most cited paper. The two original BLAST publications have a current combined citation count of 156,587[1]. The number of different bioinformatics software tools has also grown exponentially in recent years [4]. OMICtools, which is an open-access database of resources for the bioscience community, contained over 18,500 tools in 2017 [5] compared to 4,400 in 2014 [6].

Each tool comes with its own flexibility to cater to the needs of the end user. For example, BLAST has multiple variants (e.g. web versus standalone, searching for proteins versus searching for nucleotides). Most tools also provide the end user with many customization options. The command line version of Nucleotide BLAST (BLASTn) has over 40 options that a user can modify such as `-html` which produces an HTML style output, and `-num_threads=<X>` which instructs BLASTn to run on `X` number of threads. The customization and flexibility of these tools indicates bioinformatics is entering a mature phase of software development — that of highly-configurable software. Highly-configurable software contains configuration options that can be turned on or off in varying combinations depending on user preferences and environmental conditions. The selection

[1] Citation count obtained via Google Scholar on April 15th 2020 with 83,411 citations for *Basic local alignment search tool* [2], and 73,176 citation for *Gapped BLAST and PSI-BLAST: A new generation of protein database search programs* [3].

of configuration options (a configuration) can lead to different portions of the code being activated, which can have a direct effect on the program's behavior [7–11].

At the same time, synthetic biology — the practice of engineering living organisms by modifying their DNA — has advanced rapidly over the last 30 years [12]. Synthetic biologists design new functionality (such as instructing an organism to glow green when it detects a toxin), encode this in DNA strands using the genetic alphabet (A,T,C, and G), and insert the new DNA into a living organism such as the common K-12 strain of the bacteria *Escherichia coli (E. coli)*. These synthetic biology constructs are created by combining short DNA sequences. There are sequences required to make any functional system, as well as a plethora of optional sequences, each with their own unique function. A set of DNA sequences together creates the functional product. In essence, the synthetic biologist is *programming* the organism, and the choices of which DNA sequences to combine directly affects how their creation functions. As they share their designs and promote re-use, their programs are also emerging as highly-configurable.

As these bioscience systems become mainstream tools for the biology and bioinformatics communities, their usage become critical. Scientists are making decisions and drawing conclusions based on the software they use, and the constructs designed by synthetic biologists are being built into organisms and used in the real world. Improper use or bugs in tools can lead to incorrect results, false interpretations, and even lead to publication retractions [13, 14]. However, in a study examining how users interact with bioinformatics software [15], users were quoted as saying:

"*I don't necessarily know enough to make sure I'm picking our settings correct,*"

and

"*At the end of the day, you just have faith that the program's working or not.*"

This indicates that the bioscience user community needs support towards understanding the impact configurability has in their results, and their role in managing these highly-configurable systems.

The concept of highly-configurable software has been well studied within the software engineering community [7, 9–11, 16–21]. Faults may be visible under only specific combinations of configuration options, and different code paths of a program are executed under different combinations of

options [7–11]. However, configurability may lead to more subtle issues in a scientific domain. Since many bioscience tools simulate or estimate naturally occurring phenomena (such as growth of an organism in a particular environment, or how related two species are), a known expected result is often difficult or even impossible to obtain [22]. This means the tool can not be easily validated for accuracy, an issue known as the oracle problem [23]. Even if a scientific result is known, they can also suffer from excess variability due to their non-deterministic nature [24]. Additional challenges such as the large configuration spaces and large-scale heterogeneous data have been noted in the scientific domain of *computational materials science* [24].

Furthermore, the users of these systems are not assumed to be computational experts. Many users of bioinformatics software have their primary study in the laboratory sciences which we refer to as *bench scientists*. Surveys have shown that these scientists learn how to use tools primarily from their peers and self-study, not from any formal training [25]. There can also be communication issues between bench scientists and software engineers that can lead to incorrect or misleading software tools, or over-configurability due to misunderstandings about the requirements [26, 27].

Given these challenges, it can be difficult to manage configurability. Even if the configuration options make sense to an expert user, understanding all options and their interactions is difficult or even impossible to compute due to the exponential number of combinations. Biologists often develop a lab lore of what to change and often leave the rest of the options alone which can lead to wasted functionality or to incorrect or misleading results. Similarly, synthetic biologists must choose how to combine small DNA segments — often advertised as LEGO® pieces — to create their desired functionality. However, these LEGO® pieces come with no building instructions. There can be millions of ways to combine these pieces and determining the architecture can require significant domain knowledge to develop.

In this dissertation we address the above challenges of configurability in the biosciences in two areas: (1) bioinformatics software, and (2) synthetic biology. We highlight the challenges of configurability in these areas and provide approaches to help end users navigate the configuration spaces. A common theme of this dissertation is to be inspired by research in software engineering

and software testing, and apply those methodologies to the biosciences. Along the way we uncover novel challenges that call for new research in the software engineering community. We continue with two motivating examples next.

1.1 Motivation

Consider an end user who wants to run BLAST, which searches a single (or multiple) database for the match to a single (or multiple) DNA sequence (via queries). They navigate to the online interface, and can either run the query with the default settings, or open a screen that allows them to change the core search algorithm options (as seen in Figure 1.1). In this screen they find five pull down menus, five check boxes, and two text boxes that take numeric inputs for a total of 12 configurable options. Each of the pull down menus has a number of values (or options) that can be selected. They can change any or all of these options.

The user runs their query through with `max target sequences` set to 250 and leaves the rest of the options alone. They get 188 hits (or matches) returned to them. Then they run it again, but change the `word size` from 28 to 64. The new result is 180 hits returned, they lost eight hits.[2] This change in their result prompts the user to look at the other options. However, suppose this user isn't well versed with the configuration options of BLAST, they may desire to explore the options further.

Suppose we limit exploration to just five options for each of the pull down menus and text boxes. In that case we have five check boxes options with two values each, four pull down menus with five values each, and two text boxes with five values each. This leads to a possible space of $(2^5 \times 5^5 \times 5^2)$ which is two and a half million configurations. If running one configuration took 1 second it would take them more than 28 days to explore all configurations.

The user decides they want to learn more about one particular option called `Match/Mismatch Scores`, so they click on the blue circled question mark next to the option's name for additional help. The following text is displayed: *'Reward and penalty for matching and mismatching bases.*

[2]This example is running one of our *Saccharomyces cerevisiae* (*Baker's yeast*) sequences from our study in Chapter 3 searching against the nr/nt database.

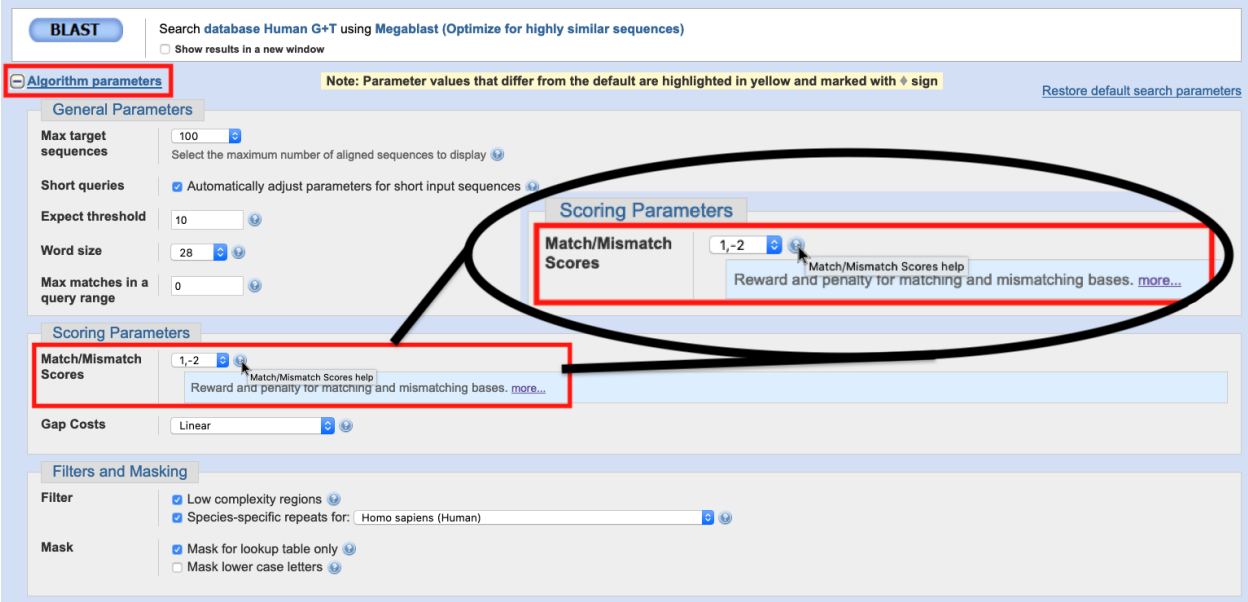

Figure 1.1: BLASTn online algorithm configuration options via NCBI. Emphasized elements are discussed in the text.

more...'. This helps them know what this option does, but they still don't know how to correctly set it for their use case. So they click on *'more'* and are taken to a webpage with a longer description of these configuration options, and links to one research article and the user manual for BLAST. This may be helpful to an expert user, but may confuse the novice, and still leaves the bioinformatics expert with a less than desirable outcome.

To understand the full impact of this configuration option, a biologist would need to set up a series of interactive experiments. Furthermore, this is just one configurable option. If we consider combinations with other choices, the number of configurations grows exponentially with the number of configurable options.

Now consider a synthetic biologist who wants to implement a specific piece of functionality into a living organism. This can be accomplished by choosing (or *designing*) the DNA segments of the living organism. This user probably has in mind the type of construct or device they want to build (such as a vaccine). They may also have basic information such as the core elements (or *parts*) of this construct; similar to an architect knowing a house needs walls and a roof. However they may

not know the specifics of the parts they need (such as tin or wood roofing, concrete or brick walls in the case of an architect). There are two things the designer needs to do; (1) choose the parts they need, and (2) create or obtain these parts.

Let us assume the synthetic biologist knows they want to create a vaccine that utilizes a particular protein called PRO_A. The synthetic designer can obtain parts either by manually writing the DNA sequences by hand and have them synthesized, or they can order existing parts. In some fields, designers have access to a catalog containing a list of all the different options available (such as a roof supply catalog). Analogously, if the synthetic designer is part of a company they may have their own private catalog of basic DNA parts, otherwise public repositories also exist [28]. The synthetic designer could search their catalog for all parts relating to PRO_A. However this method does not help them figure out how to put these parts together to create a functional device.

As an alternative, they could also see if the catalog has examples of existing devices that are related to their protein of interest. However this will only show particular examples, and may not generalize to all possible methods of creating the device. While both of these search methods are helpful, neither provides all the information the designer needs to make the most educated decision for their new design. It may be helpful if this information (all part options and examples of complete products) could be combined in some way to help this designer create their new device in an unbiased manner.

These two scenarios lead us to ask what the situation is like for the various users of bioinformatics software and developers of synthetic biology constructs. We do not expect broad expertise in software engineering from these users, therefore we want to understand what the implications might be for those using these systems. In this dissertation we provide a set of configuration-aware techniques to help users navigate these configuration spaces, and in the process we find new challenges due to the unique characteristics of these systems that call for new research from the software engineering community leading to more interpretable configurable software. The contributions of this dissertation include:

- Motivating the need for configurability awareness in bioinformatics software (Chapter 3) through:
  - An empirical case study on three different configurable bioinformatics programs and four configuration models;
  - Identifying the impact configuration options have on the program's functional output, performance (measured by runtime), and in finding bugs; and
  - Exploring if common sampling methods (random, covering arrays, option-wise, and negative option-wise) are effective at identifying the effects of configuration options.
- A framework to help end users navigate configurable spaces (Chapter 4). Elements of this work include:
  - A framework called ICO which heuristically identifies the effects of configuration options in highly-configurable software tools;
  - A large-scale study evaluating this framework on two bioinformatics programs and one software engineering tool; and
  - Comparison our framework to the state of the art in prediction modeling.
- A mapping of software product line engineering to the domain of synthetic biology (Chapter 5). Individual contributions of this work include:
  - Definition of *Organic Software Product Lines* (OSPLs);
  - Empirical evidence demonstrating the potential reuse and existence of both commonality and variability in the largest open source repository of DNA parts;
  - An empirical study with four manually and automatically reverse engineered feature models; and
  - A discussion on limitations of existing techniques for organic software product lines and its implications for the software product line engineering community.

We present the reader with background in Chapter 2. In Chapter 3 we motivate the need for configurability awareness in bioinformatics software and present the ICO framework to help end users navigate these configuration spaces in Chapter 4. Chapter 5 presents *organic software product lines*. We summarize in Chapter 6 and discuss future work.

CHAPTER 2. BACKGROUND

This chapter presents the reader with background information and related work for the following main topics in this dissertation: configurability in software systems, software product lines, bioinformatics, and synthetic biology. Additional background and related work specific to a study can be found in its relevant chapter.

2.1 Configurability in Software

As software systems have evolved over the years they have also increased in their complexity [29]. Software is being designed to be flexible enough to satisfy diverse requirements and be applicable to multiple use cases. This is often accomplished the form of customization. This creates what we know as *highly-configurable software* where the end user is provided with many customization options [7, 30].

One example of a highly-configurable software system is the Linux kernel, which has been reported to have over 13,000 configuration options leading to more than $2^{13,000}$ configurations [31]. Another example is the GNU compiler — `gcc` — which has been reported to have as many as 10^{61} configurations [18]. `gcc` has basic command line options such as `-o <file>` which defines the output file to write the compiled code to. Another commonly used configuration option is `-w` which suppresses all warnings. `-w` is a configuration option of type *flag* which is a specific type of *Boolean* option. A Boolean configuration option can be set to either of the values `{TRUE, FALSE}`, where a flag type option has the options `{Enabled, Disabled}`. We often refer to flag type configuration options as Boolean in this work as a simplification since their semantics are the same.

Configuration options can also be of the *numeric* type. These options can have an integer or float value, and often there is a bound on the range of values such as [0,100]. An example for `gcc` is `-fdiagnostics-hotness-threshold=<number>` which sets a threshold for optimization remarks

being output based on their profile count. Configuration options can also be of the *enumeration* type where its values are defined by a specific set of allowed values (usually a set of strings). For example the `gcc` configuration option `-fembed-bitcode=<option>` can be set to: `all`, `none`, `function`, or `custom`.

Formalization

We formally define some commonly used terms in this dissertation related to configurability. We refer to each software setting or feature as a *configuration option* (CO). A *configuration* ($\mathcal{C}$) is a full assignment of configuration options as defined as:

$$\mathcal{C} = \{CO_0, CO_1, ..., CO_x\}$$

Each of these configuration options can take on multiple values (v). For example if CO_0 can take on y different values, its values are defined as $\mathrm{V}_0 = \{v_0^0, v_1^0, ..., v_y^0\}$ where the superscript denotes the configuration option and the subscript denotes the value. Values for all configuration options ($\mathcal{V}$) can be defined as:

$$\mathcal{V} = \bigcup_{i=0}^{x} \{v_0^x, v_1^x, ..., v_{|\mathrm{V}_x|}^x\}$$

where x is the number of configuration options. We can define all configurations options and their values into one *configuration model* defined as a tuple of the configuration options and all values $(\mathcal{C}, \mathcal{V})$.

In this work we also allow a configuration option to have a value of $\emptyset$ or `NULL`. Setting a configuration options to `NULL` means we do not change its value. To do this we explicitly do not use the option (i.e. not writing a command line argument in our function call). We note that some configuration models may explicitly include a value of `NULL` for a configuration option.

Software Testing of Highly-Configurable Software

The concept of highly-configurable software has been well studied within the software engineering community [7, 9–11, 16–21]. The software testing community has found that faults may be visible

under only specific combinations of configuration options, and different code paths of a program are executed under different combinations of options [7–11] which has led to many *configuration-aware* testing techniques [9–11, 17, 18].

Other research has investigated how evolving configuration models affect regression testing [9, 16], and how to predict performance [32–36]. There also exist tools for studying and dealing with configurations and variability in software that are not strictly testing related. For example iGen [10] is dynamic inference tool that focuses on finding interaction between attributes (which could be configuration options). iGen returns the resulting interactions it learns in the form of invariants.

Sampling of Highly-Configurable Software

Exhaustive testing of these highly-configurable systems is often impossible due to the exponential size when you consider all combinations of configuration options. Most real software systems have too many configurations to enumerate and test. To address this, systematic sampling of the configuration spaces can be applied.

Common sampling method for Boolean configuration options include using random, and option-wise, and negative option-wise testing which has been used in configuration performance tuning [33]. In option-wise testing, each configuration option CO_x is set to `TRUE` while all remaining configuration options are set to `FALSE`. If the configuration option is included in a constraint, then the valid configuration with the minimal number of excess options set to `TRUE` is chosen. Its counterpart negative option-wise is the inverse, where all configurations are set to `TRUE` except the option under test.

To sample numeric configuration options, methods from *Design of Experiments* [37] such as One Factor At a Time (OFAT), Box-Bohnken, Plackett-Burman, Central Composite, D-Optimal, and random have been applied to highly-configurable software [32, 38]. The OFAT method is analogous to the option-wise sampling for Boolean configuration options where each configuration option's values are perturbed one-by-one. Instead of all other options being set to `FALSE` they are left at some other unchanged value (e.g. the center of their value domains) [38].

Another type of sampling technique is combinatorial interaction testing (or CIT) [18, 19, 39, 40] which is an adaptation of orthogonal arrays [41] from Design of Experiments. CIT has been used for testing of highly-configurable software to sample for fault detection [10, 17, 19]. CIT samples broadly and systematically across factors (configuration options) by generating small (optimized) samples that cover all t-way combinations of factors in at least one configuration. The variable t is called the strength and determines how broadly we sample. The most common sampling used in software is $t = 2$ or pairwise testing. In pairwise testing all pairs of features appear at least once in the sample. Research in software testing suggests that t between 2 and 6 should be sufficient and that 2 or 3 finds most faults [42].

Underlying most CIT sampling is a mathematical object called a covering array (CA) which defines the sample. Many algorithms and tools have been developed to find CIT samples such the Automatic Efficient Test Case Generator (AETG) [43], the In Parameter Order General (IPOG) algorithm [44] implemented in ACTS [45] or simulated annealing, implemented by Cohen et al. [46] and in CASA [47]. In other related work, iTree combines low-strength covering arrays with machine learning to generate samples [48], and SPLat uses heuristics to iteratively prune configurations [49].

Influence Models

In order to formally explain the effect that changing configuration options has on a program, one method that has been applied is building influence models; specifically, regression equations where the variables are the configuration options.

The tool SPL Conqueror utilizes these influence models to be able to explain the impact of configuration options on a program's performance such as runtime or memory consumption [32, 33]. Input to SPL Conqueror consists of (1) a configuration model and (2) a set of measurements. The configuration model takes the form of a feature model (a representation of the configurations of the system, see Section 2.2 for more). SPL Conqueror uses its own xml formatting for this model. The measurements define the actual measured performance for various configurations. The measurements can either be the entire configuration space, or a sample. SPL Conqueror provides

several sampling methods in its implementation such as option-wise, negative option-wise, pair-wise, and more. Other work has looked at applying transfer learning to learn prediction models [50, 51].

Highly Configurable Software in Other Domains

Some recent research has looked at variability in programs that calculate partial differential equations [35, 52], numerical solvers [53], and configurable robotics [54]. None of these are in the bioinformatics domain. Recent work that studies bugs in numerical software [55] does not address the issues of configurability, nor does other work on testing bioinformatics software [22, 56].

2.2 Software Product Lines

Software product line (SPL) engineering is a best practice for modeling, building, and managing reuse in families of software systems [57–59]. 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 the *product*. An essential component of an SPL is the *feature model* which is a representation of the entire product space as modeled by its features [57]. Some features will be mandatory to all products, some will be optional, and there are additional relationships such as alternative and OR groupings. There can also be constraints between features.

As an example consider a company that develops a line of cellular phones and the accompanying feature model in Figure 2.1. The product (cell phone) is depicted as the root of a tree. This 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 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 both of these feature. There is also a cross-tree constraint in this model where `Music_Casting` requires the `Bluetooth`

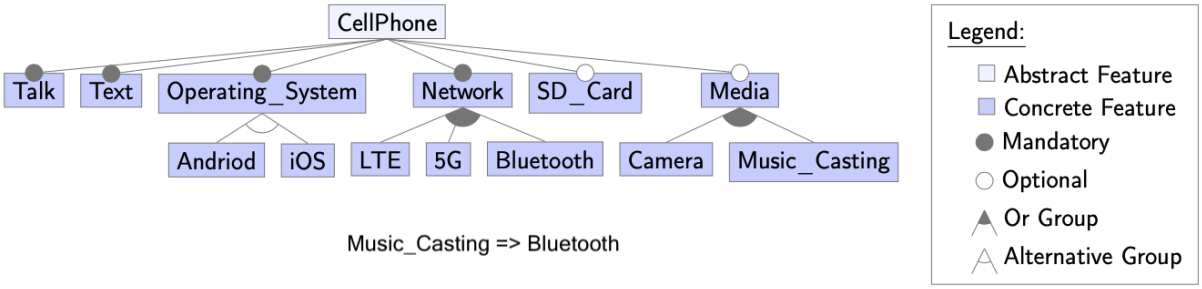

Figure 2.1: An example feature model for a product line of cell phones.

feature to be included. Each cell phone in their line of products will be a combination of these features.

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` [60, 61].

Further still, product line engineering has recently been applied in many emerging domains including drones and nanodevices [62, 63]. There is also a push towards open-source product lines [61, 64, 65] and the concept of a software ecosystem, where the community modifies and customizes product lines using a common platform and look [66]. Lutz et al. [63] 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 [67]. This work is complimentary to ours.

Reverse Engineering of Feature Models

A feature model for a software product line does not always automatically exist [68]. Manually creating a feature model can be time consuming and require significant domain knowledge. To tackle this challenge there has also been growing interest in automating the process of reverse engineering feature models from products and features. An early example by [69] used search-based techniques such as genetic algorithms to generate proper subsets of feature models based on feature sets.

Another trend has been to focus on large, open-source product lines [61, 64, 65] and the concept of a software ecosystem, where the community modifies and customizes product lines using a common platform and look [66]. It is possible to reverse engineer feature models from large, open-source projects such as the Linux, Debian, and FreeBSD kernels [70, 71]. 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).

2.3 Bioinformatics

Bioinformatics can be broadly described as the study of collecting and analyzing biological data [72]. Some popular areas of study in bioinformatics include: omics studies where genomic, proteomic, and transcriptomic data are analyzed [73, 74]; sequence analysis where DNA, RNA, or peptide sequences are studied to find patterns or discover its evolutionary structure [75, 76]; and systems biology which abstracts out from a complex and integrated biological system [77, 78]. Often the tasks in bioinformatics are computational in nature such as DNA alignment which is a form of a string search problem. There are a wide array of tools to automate these tasks. Other tools include those to simulate or model living systems. One example is *Flux Balance Analysis* (or FBA) which simulates the growth of an organism living in a particular environmental condition [79]. FBA uses linear programming as the underlying optimization technique.

There also exist open-source collections of bioinformatics tools such as the U.S. Department of Energy's System Biology Knowledgebase (KBase) [80]. KBase has more than 220 apps deployed

at the time of this dissertation, providing a wide range of functionality including read processing, genome assembly, genome annotation, sequence analysis, comparative genomics, metabolic modeling, expression, and microbial communities. KBase tools have been used in over 100 publications to date[1], and the methods implemented in the tools in thousands of publications. The KBase environment provides a workflow system based on Jupyter notebooks in a format they call *narratives* allowing users to freely explore their data in a sandbox environment. These narratives can be published and shared allowing others to view and recreate their experiments.

2.4 Synthetic Biology

Synthetic biology is the practice of engineering living organisms by modifying their DNA. Its applications include pollution mitigation [81], development of synthetic biofuels [82], engineering new forms of cellular communication [83], emerging medical applications [84, 85], and basic computational purposes [86]. 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.

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. In essence, the biochemist is programming the organism to behave in a new way.

This analogy of *programming biology* is not a new concept. There is even a programming language called the Synthetic Biology Open Language (SBOL) which defines a common way to represent biological designs [87]. Researchers have used synthetic biology to create a context-free grammar using BioBricks [88], automated design of genetic circuits with NOT/NOR gates [89], and bacterial networks to use DNA for data storage [90, 91]. There are also several examples of organic

[1] 108 publications counted April 2020 at http://kbase.us/publications/

programs inspired by classic computer science constructs such as a genetic oscillator [92], a genetic toggle switch [93], and a time-delay circuit [94].

As DNA strands have become easy to engineer by simply purchasing a desired sequence, the field of synthetic biology has rapidly grown [12]. For instance, each year 300+ teams of students (high school through graduate) compete in the International Genetically Engineered Machine (iGEM) Competition [95]. 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.

Participants 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) contains over 45,000 DNA parts and can be viewed as a Git repository for DNA [28].

CHAPTER 3. CONFIGURABILITY IN BIOINFORMATICS SOFTWARE

The bioinformatics software domain contains thousands of applications for automating tasks such as the pairwise alignment of DNA sequences, building and reasoning about metabolic models or simulating growth of an organism. Its users range from sophisticated developers to those with little computational experience. In response to their needs, developers provide many options to customize the way their algorithms are tuned. Yet there is little or no automated help for the user in determining the consequences or impact of the options they choose. In this chapter we describe our experience working with configurable bioinformatics tools. We find limited documentation and help for combining and selecting options along with variation in both functionality and performance. We also find previously undetected faults. We summarize our findings with a set of lessons learned, and present a roadmap for creating automated techniques to interact with bioinformatics software. We believe these will generalize to other types of scientific software.

3.1 Motivating Examples

The bioinformatics software user community is broad, therefore developers should consider a spectrum of users. Figure 3.1 shows the three primary users that we consider in this work. At the top level are the domain experts (or non-software experts). These users have a deep understanding of the biological problems they are trying to solve, but will unlikely be familiar with the configurable options and/or know which defaults are set in the software. Often, they work from a user interface that limits their options with respect to configurability. We examine one model in our study that is geared towards this group. As pointed out by Jin et al. [97], configurable systems usually have multiple ways to interact with configuration options, and the highest level (such as menus or a graphical interface) often only provide a subset of those options.

The material in this chapter was published In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)* in September 2018 [96].

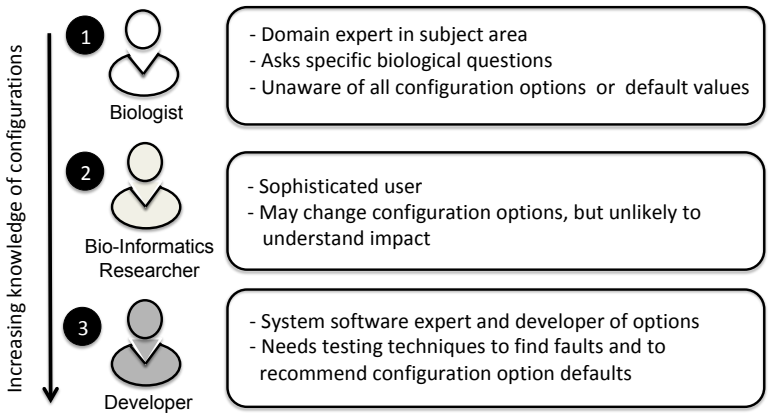

Figure 3.1: Three primary users.

The next type of user in this figure is the bioinformatics expert user who has both a deep knowledge of the biological domain, and with computation and automation. This type of user will often run experiments using scripts and will have some knowledge of which configuration options can be altered. However, they are unlikely to understand the full impact of those changes, nor do we expect them to be experts in software development or testing.

Finally, the expert developer is someone with extensive development experience with the domain, who has also built the application. They will be familiar with all of the configuration options and will have experimented with each, however, they may lack techniques and methods to test the large configuration space for correctness or to optimize and determine the best default values of each.

During the course of this research we have acted in the first two roles (both as biologist end users and as bioinformatics specialists). We have also reached out to developers with our findings and report on those experiences.

We next describe our experience with an application that assembles a set of short DNA strings called *reads* into larger continuous strings called *contigs* using an online web interface (i.e. again from a biologist's perspective). We attempted to recreate data from a tutorial for an application called MEGAHIT from the Department of Energy Systems Biological Knowledgebase (KBase) [80, 98]. The web version is simply a wrapper for a commonly used command line tool also called MEGAHIT [99].

We used the default values for each of the configuration options used in the tutorial. We noticed that one configuration option (`k-max`) did not make sense since it represents a read length (the number of elements in the sequence to read) and was set to 141, higher than the input sequence (100) used in the tutorial - typically something a user will not do. We ran the application anyway and got a completed assembly with 284 contigs. We then read the log and noticed that the system reported it changed the `k-max` for us to 119. But this was done silently (no notification) and was only found by our exploration of the log. This number is also greater than the maximum read size from the input set of reads. We then ran the tool and set the `k-max` to 119. This time we saw 285 contigs and thus inconsistent results. Last we set the `k-max` to 99 (below the read length) and expected to see one of the above numbers, but instead got 289 contigs.

We contacted a developer of the web app and were told that the problem is due to edge cases. The mismatch is due to the use of configuration options that combined with the input data do not make sense, and that the system autocorrects for us. However, the software did not document this clearly to the end user. Since the software was modifying values automatically to correct the incorrect `k-max` option value, we were getting inconsistent results. The developer confirmed that this is a bug and said that they would fix this in a future release.

This scenario lead us to ask what the situation is like for the various users of bioinformatics software and what the impact of modifying configuration options might be. We do not expect broad expertise in software engineering from these users, therefore we want to understand what the implications might be for those using these systems. We also want to understand if configuration-aware techniques can help to uncover differences in either or both functional and performance outcomes of these tools and if there is a cautionary tale for this community. Our answer for both is yes.

3.2 Case Study - Experimenting with Configurations

In this section we present our case study. We explore the following research questions:

RQ1 Does manipulating configurations in bioinformatics software lead to failures?

RQ2 What is the impact of manipulating configurations on program behavior? To answer this we ask two sub questions:

a) What is the impact on program functionality?

b) What is the impact on program performance?

Our last research question focuses on the ability to use common sampling techniques that might reduce user effort and improve scalability when exploring configuration options.

RQ3 How effective is sampling in these configuration spaces?

3.2.1 Bioinformatics Programs

We select three bioinformatics programs, all of which are configurable, and all available for different types of end users. These tools come from three different areas of study: genome annotation, genome assembly, and metabolic modeling. In the metabolic modeling tool we study two different versions. The first is meant for non-expert users and run from a graphical user interface (GUI). The second is geared towards the experienced bioinformatics user, and run on the command line. This gives us four unique configuration spaces. For each area, we chose a tool that (1) is widely used, (2) is configurable, (3) has some tutorial or other documentation that provides common inputs along with expected outputs to avoid author bias, and (4) can be run through automated scripts.

Table 3.1 shows the three bioinformatics program subjects along with the lines of code (LOC) using CLOC [100]. We describe each application below. We also show the total number of configuration options, along with those chosen for each model broken down by type. For example, the FBA-MFA subject has 21 Boolean and 14 float options. Last we give the size of the full configuration space including constraints (Section 3.2.2 contains details for each model).

1. BLAST Our first subject is Nucleotide BLAST *Basic Local Alignment Search Tool* (BLASTn), a popular bioinformatics tool developed by the National Library of Medicine [75].[1] Given an unknown DNA sequence (a query), the application attempts to align the sequence to a database of known DNA sequences in order to learn its likely function. Matches are returned with quality scores

[1]The BLAST software contains different variants. Table 3.1 shows the LOC for the full BLAST software system. We utilize only one variant of BLAST (BLASTn) in this work.

Table 3.1: Case Study Subjects

	BLAST	MEGAHIT	FBA	
			GUI	MFA
LOC	1,095,106	38,093	28,909	28,909
Config. Opts.	58	7	17	52
Configuration Options Selected for Model				
Boolean	3	-	3	21
Int	-	4	-	-
Float	3	-	5	14
String	1	-	-	-
Total	7	4	8	35
Config. Space	3,000	260	25,000	1.28×10^{16}

(percentage ID and e-values) that indicate how confident the search is with respect to each possible match. The queries can be run for either a single DNA sequence or a set of DNA sequences. We use as our input query, a file of 10,000 sequences from a yeast organism *Saccharomyces cerevisiae*, also known as *Baker's yeast* [101]. This input was used as an example in a tutorial on BLAST for bioinformatics researchers at the University of Nebraska-Lincoln [102].

Our other subjects are the two most commonly used analysis tools taken from the open source GitHub repository of the U.S. Department of Energy's System Biology Knowledgebase (KBase) [80]. We can confirm results using their online web interface and are able to use their tutorials to obtain sample inputs/outputs along with descriptions of how each of the tools are used.

2. MEGAHIT Our next subject is the DNA assembler *MEGAHIT* [103]. The KBase tool is a wrapper for an existing version of MEGAHIT [99]. When DNA is sequenced the result is a set of small sequences of base-pairs called *reads*. In order to combine these small sequences, assembly is used to connect them together into longer sequences called *contigs*. This has been used 1,120 times inside of the KBase narrative as of April, 2018. We use the default paired-end-library from KBase which contains 386,106 reads all of length 100 basepairs (bp) from the *Rhodobacter* bacterium. This library is used in the KBase online tutorial [98].

3 and 4. Flux Balance Analysis The last program is a metabolic modeling toolkit that performs a flux balance analysis (FBA). The metabolism of an organism can be modeled as a set of concurrent chemical reactions (a reaction network). FBA optimizes reaction fluxes (flows) through an organism's metabolic reaction network to predict the organism's growth [79]. It uses linear programming as the underlying optimization technique. We utilize the open source standalone version (toolkit) of the FBA from the KBase GitHub software distribution kit [104]. KBase has a web version of the FBA application (we refer to this as the FBA-GUI model) that has been run 7,803 times as of April, 2018. The standalone toolkit (we call this the FBA-MFA) has a larger set of configurable options available to the user. A subset of these options are available within the online version in KBase. To understand if the web narrative provides similar behavior with respect to configurability, we model these independently. We fix the organism (*Escherichia coli*) and growth medium (Carbon-D-Glucose) to match that of the online tutorial which describes how to use FBA in KBase.

3.2.2 Creating the Models

Our first step is to develop configuration models for each application. We use available documentation, such as tutorials, the application interface, online documentation, and we hold discussions with domain experts. Since we are evaluating functionality as one objective, we try to avoid modifying configuration options that are clearly functional, therefore we leave those out of our models. For example, we do not change the growth media in FBA as that will clearly impact how the organism grows. Other configuration options are removed for reasons including: dependencies with default configuration options, they only impact output formatting, filter/threshold, and unique string values that could not be put into equivalence classes. In choosing the values for the configuration options we always include the default option and then try to evenly distribute equivalence classes for other values when it makes sense. We rely heavily on guidance from the documentation (or online developer forums) to determine values.

We found that it was non-trivial to construct a configuration model for each subject. Some configuration options are ill-defined, there is a lack of documentation, or there are conflicting summaries in multiple help documents. We also found that the explicit ranges of values are often missing and we had to reverse engineer default values. We were quickly lost in the maze of configuration options. In order to find our way out we communicated with developers and expert users of the tools to identify configurations of interest for each model. We discuss some of these experiences next.

BLAST We began with the full list of 58 command line configuration options from the user manual and scheduled two consultations with an expert user (who is not one of the authors of this paper). The user was only familiar with some of the configuration options, therefore their input was useful, but we still ran into several challenges. We first removed any clearly functional, string, and set configuration options giving us an initial model of 18 configuration options. However, in running preliminary experiments, we ran through a total of five iterations of the model due to errors and inconsistencies in the documentation. At each step we had to stop and trace the errors back to their cause. We do not list them all, but summarize the key learning points. In preliminary testing we came across one error: *BLAST query/options error: Greedy extension must be used if gap existence and extension options are zero. Please refer to the BLAST+ user manual.* This error message is due to incompatible configuration options, but does not name them. After searching web forums with little success, we searched the source code and found that the *no_greedy* option had constraints. The configuration option *no_greedy* can not be used when *gapopen* and *gapextend* are zero. In our model we were not changing these values because they are functional, and their default is 0, therefore we are unable to include the *no_greedy* option in our model either.

We also discovered configuration options that behave as filters that were not obvious when studying the user manuals. Identifying such functional options was a hard task. Even after looking though two user manuals and consulting two users some configuration options' functionality were still not clear. The `word_size` option was removed as it was discovered to be a filtering configuration

Table 3.2: BLAST Configuration Model

Option Name	Type	Values
dust	String	yes, “20 64 1”, no
soft_masking	Boolean	True, False
lcase_masking	Boolean	True, False
xdrop_ungap	Real	0, 0.1, 0.5, 20, 100
xdrop_gap	Real	0, 0.1, 0.5, 30, 100
xdrop_gap_final	Real	0, 0.1, 0.5, 10, 100
ungapped	Boolean	True, False

Table 3.3: MEGAHIT Configuration Model

Option Name	Integer Range	Values
min-count	[2,10]	2, 4, 6, 8, 10
k-min	[1,127] odd only	15, 21, 59, 99, 141
k-max	[1,255] odd only	15, 21, 59, 99, 141
k-step	[2,28] even only	2, 6, 12, 20, 28

Constraints: k-min $<$ k-max, k-step $<$ k-max$-$k-min

option and would clearly change the functional outcome of the query. We discuss the importance of this in Section 3.4.

One modification to the model was triggered by a sanity check we performed to confirm the default configuration reported in the manuals was in fact the default. We compared the configuration option manually set (`blastn -db [DATABASE] -query [QUERY] -out [OUT] -word_size 11 -dust "20 64 1" -soft_masking true -xdrop_ungap 20 -xdrop_gap 30 -xdrop_gap_final 100 -window_size 40 -off_diagonal_range 0`). To our surprise, the results were not the same. We first referred back to the manual and command line help menu to ensure the default values of the configuration options were correct (they were). After continued inspection we noticed an implicit configuration option called `task`) that did not have its own declaration in the manual or the help menu. BLASTn can be used with four different `tasks` and we use the default. However, the default value listed for a BLASTn option we do use (`window_size`) is for a different task, and the task we use has no default value listed for `window_size`. So we removed `window_size` from our model to avoid this constraint. There are no constraints in the final model.

Table 3.4: FBA Configuration Models

Option Name	Type	Values
flux variability analysis	Boolean	True, False
minimize flux	Boolean	True, False
simulate all single KOs	Boolean	True, False
MaxC	Float	0, 25, 50, 75, 100
MaxN	Float	0, 25, 50, 75, 100
MaxO	Float	0, 25, 50, 75, 100
MaxP	Float	0, 25, 50, 75, 100
MaxS	Float	0, 25, 50, 75, 100
Example of Additional MFA Options		
maxDrain	Float	0, 250, 500, 750, 1000
minDrain	Float	0, -250, -500, -750, -1000
deltaGSlack	Float	0, 5, 10, 15, 20
find_min_media	Boolean	True, False
allRevMFA	Boolean	True, False
useVarDrainFlux	Boolean	True, False

MEGAHIT. There are seven configuration options accessible from the web interface. Many of these options determine a list of integer values called `k-list`. This controls the length of the substring used in the database search. For example, we can set a `k-min`, `k-max`, and `k-step` or explicitly set the entire list with the `k-list` configuration option. Similarly the option `parameter preset` is a different method of fixing the same options - we exclude both from the model because they impact functionality. We also fix the configuration option `min-contig-length` because it directly impacts our functional result. We confirmed this model with a KBase developer. When working with this model we ran into several issues of consistency which we discussed in Section 3.1.

The MEGAHIT model has constraints between the configuration options (such as `k-min` < `k-max`). This constraint was not listed in the documentation, but is considered common knowledge (min should be less than max) and was mentioned to us by the developer. We also confirmed this constraint by running initial experiments where we received an error message. Although it properly returned an error, the message was not descriptive enough for an end-user to understand. The final MEGAHIT model is in Table 3.3 and has 4 configuration options.

FBA For the FBA-GUI subject we first fixed all of the input files (describing the growth conditions) and ruled out configuration options that were of type *set*. Set options were removed as the number of configuration options is on the order of 10^{258} with no logical method of partitioning. We also do not use an optional input (`expression data set`) to keep the experiment simple. There was one float type configuration option that was not included in the command line version of the tool, so it was excluded from the model as well to ensure we could compare the two. Table 3.4 displays the configuration options for the FBA models. The FBA-GUI model consists of the first eight configuration options, three Boolean, and five floats evenly partitioned into five values.

The FBA-MFA model contains 27 additional advanced configuration options, five of which are displayed in the table (please find the complete configuration model in Appendix A). The complete FBA-MFA model contains 35 configuration options, 21 Boolean and 14 floats evenly partitioned into five values. There are no constraints in either model. We believe that the lack of constraints may be partially due to our selection of configuration options, and leave further exploration of this as future work.

3.2.3 Measures of Variability

For program functionality we choose common use cases for each tool. For performance we measure the program execution time (in seconds). We describe each of the functional metrics next[2].

In BLAST the input is a set of query sequences of unknown function that are checked against a database of query sequences. There are two main use cases: (1) the user inputs a *set* of query sequences and then looks at the best quality hits (defined as 100% basepair matches and an e-value of 0.0) or (2) the user inputs only one query sequence at a time and checks for any hits against that sequence. We explore both use cases. In the first, since there are large numbers of hits returned, we look at the Top 5, 10, 20, and 100 highest quality hits and compare them to the Default Top 5, 10, 20, 100. We keep the number small since the analysis of the returned hits is often a manual

[2]We provide raw data on our supplementary website: https://github.com/mikacashman/ASE18SupResources.

effort by the user. For the second use case we count the *number of hits* for each individual query sequence.

MEGAHIT is used to assemble short DNA reads from sequencers into continuous DNA sequences. The functional dependent variable is the *number of contigs* generated. This represents how many continuous reads of DNA the algorithm was able to assemble. The objective of the tool is to get the smallest number of long contigs.

The result of the FBA analysis is called the *Objective Value* (OV) which is a measurement of how much the organism grew. We use this as the functional output.

3.2.4 Experimental Setup

The BLAST model has a configuration space of 3,000 which is small enough for us to enumerate and evaluate. We exhaustively run all configurations for our experiments. We run the command line *BLASTn* version 2.6.0 on a LINUX cluster with 100 jobs in parallel, each job on 1 node with 5GB memory.

The configuration model for MEGAHIT has 260 configurations.We exhaustively run all configurations in this space as well. The MEGAHIT app is version used is 2.2.8 in KBase. We run these experiments in parallel on 10 virtual machines. Each machine runs CentOS-7-x86_64 with 80 GB disk and 2MB RAM.

The FBA subjects' source code is extracted and run on a LINUX cluster in order to scale the experiments and not overload the KBase servers. Since the total configuration space for FBA-MFA is too large to study (1.28×10^{16}) we randomly selected 125,000 of these combinations. We use FBA version 1.7.4 running 1,000 jobs in parallel, each job on 1 node with 10GB memory. In order to test all configurations in the FBA-GUI model (25,000) we test it directly from the MFAToolkit also running 1,000 jobs in parallel. We also test and confirm that our functional results matched on a sample of these when running manually in the KBase online narrative. In early iterations of our study on FBA-MFA, some of our runs were hanging (not completing overnight) and this caused the experiments to stop. We added a timeout factor of 1,000 seconds to prevent this problem and kill

any job that goes beyond this time. We note that we did have some runs of exactly 1,000 seconds (seen in our data) that did complete and produce valid outputs. We do not explore the best cutoff time further in this work, but believe these are outliers.

3.2.5 Sampling

To evaluate RQ3 we applied a common configuration-aware testing technique, combinatorial interaction testing (CIT) [16, 19, 39, 40]. CIT combines all t-way combinations of configuration options together in at least one run of the sample. Please refer to Section 2.1 for more information on CIT. We used the CASA tool [47] for our lower strength samples, but were unable to build some of the higher strength samples so used the ACTS [45] tool for those. When using CASA we generated 30 samples since it uses randomness and each sample is different. With ACTS the generation was deterministic therefore we have a single CIT sample. We also applied a variant of option-wise testing [32] which has been used in configuration performance tuning. Option-wise sampling implies binary configuration options and tests each configuration option being enabled once while the remaining configuration options are all disabled. Since the configuration options in our models are not all binary, we instead use the default configuration setting as our baseline. Then for each configuration option and for each value of the configuration option, we change that one option in the default configuration.

3.3 Results

In this section we present the results for each of our research questions. We discuss the implications in Section 3.4.

3.3.1 RQ1: Failure Detection

BLAST We went through several iterations of the BLAST model (discussed in Section 3.2.2) in which we ran into many error messages. However these turned out to be due to incorrect

configurations and violated dependencies. Once we fixed these, we did not find any other errors while running the 3,000 configurations.

MEGAHIT 23 configurations failed (9.62%). Five configurations reported an "*Error occurs when running sdbg builder count/read2sdbg*" and 18 configurations reported the error "*Error occurs when assembling contigs for k = 99*" error. We contacted the developers and were told that both cases are valid configurations, but incompatible with the input data. The first is due to setting two configuration options (`k-min` and `k-max`) both to 141. The second error only occurs when `k-min`=99 and `minCount`>4. In these cases the program should have resulted in no contigs, but produced an error. We note that we worked with bioinformatics experts who use this tool to model our configuration space, therefore these are not obvious edge cases. The developer explained that these are both cases where there were no assembled reads and the tool was failing ungracefully. We did see 35 more cases where there was no assembly, but these did not fail so we consider them in RQ2. In total 237 out of 260 configuration ran without errors.

FBA-MFA Of the 125,000 configurations tested, 1,285 (1.03%) of these configurations failed. We see five distinct error messages (Table 3.5 summarizes these). 631 runs timed out after 1,000 seconds. 44 configurations caused an error in the linear programming solver (called SCIP). 54 tests reported an objective value (OV) of *negative 0*, however the smallest OV should be 0. A total of 447 runs aborted, and 27 were segmentation faults.

Table 3.5: FBA-MFA - Errors

Total runs	**125,000**
Timeout	631 (0.50%)
SCIP Error	44 (0.04%)
Negative 0 OV	154 (0.12%)
Aborted	447 (0.36%)
Segmentation Fault	27 (0.02%)
Total w/out Errors	**123,715 (98.97%)**

We first re-ran these tests to confirm that the failures were deterministic and then we reported some of these on the KBase help forum and communicated directly with developers of the FBA

program. Several of the errors were confirmed. Some were unsurprising (again due to nonsensical combinations of configuration options) but some were also tagged as real bugs that should be fixed via patches to the program. In response to the SCIP error, the help board told us it was due to the linear programming solver, not the FBA. We got slightly different feedback from the developer who was confused by the error. The resulting OV value it gives us when this occurs is 0 (no growth) so the functional result is correct. The Aborted error is a memory allocation issue in the C++ code. We are continuing dialog with the developer to investigate this further. We discuss more developer feedback in Section 3.4.

FBA-GUI Of the 25,000 combinations, 16 of these combinations resulted in an error. All were the SCIP error (similar to the FBA-MFA example). We confirmed some of these within the KBase online GUI to ensure that the error occurs there as well. All again returned the correct zero OV (no growth), but there was an error reported in the online log. In this case the error may not be obvious to the user, because the GUI handles this gracefully and returns a result. The user must examine the log explicitly to see the error.

Summary of RQ1. Three of the four applications suffered from failures due to us using invalid combinations of configuration options (i.e. missed dependencies). We also saw some errors that are due to either external libraries (linear solvers) or the bioinformatics application itself.

3.3.2 RQ2 (a): Functionality

BLAST We examined two different use cases for BLAST as discussed in Section 3.2.3. For Use Case (1) Table 3.6 displays the count of Top X hits that match the default settings exactly (in terms of sequence ID and order in in list). For example, the count for Top 5 means that 2,000 of the 3,000 configurations' Top 5 hits match the Top 5 hits from the default configuration. Overall we see that ⅔ of the configurations do not vary in the Top 20 hits. The remaining 1,000 have a different list of hits than the default. If we look at the Top 100 hits only 640 configurations gave the same result as the default options.

Table 3.6: BLAST Functional Variance for Use Case 1

	Count (percent)
# hits match Top 5	2,000 (66.67%)
# hits match Top 10	2,000 (66.67%)
# hits match Top 20	2,000 (66.67%)
# hits match Top 100	640 (21.33%)
Total in sample	**3,000**

For Use Case (2) we look at all 10,000 input sequences as individual test cases (Table 3.7). We see that over half (53.69%) never find a hit across all 3,000 configurations which is typical behavior. Another 17.60% of the query sequences always have 1 or 3 hits (no variation). The remaining 2,871 (28.71%) of the query sequences have a varying number of hits across the configuration options we explored.

Table 3.7: BLAST Functional Variance for Use Case 2

Total # Query Sequences	**10,000**
Sequences w/out variance (0 hits)	5,369 (53.69%)
Sequences w/out variance (1-3 hits)	1,760 (17.60%)
Sequences w/ variable hits	**2,871 (28.71%)**

MEGAHIT Of the 260 configurations, we removed the 23 that resulted in an error. Another 35 (14.50%) resulted in zero assembled contigs and produced no output. Therefore the total number of configurations that gave valid assemblies was 202. In Figure 3.2 we show the number of contigs for the 202 configurations with a valid assembly (of at least 1). We see a large variation in the number of contigs. The median number of contigs is 324, but the variation range is from 1 to 672 contigs. This means that configurations explored have a large impact on the functional result of MEGAHIT.

FBA-MFA We removed the 1,285 error configurations from the functional analysis. Of the remaining 123,715 configurations we saw 39 unique objective values. This suggests that the configuration options we are changing have a large impact on functionality. The majority of the combinations (96.13%) do not grow resulting in an OV of zero. This is interesting because we fixed the input model and growth media to match that of the online tutorial using all default values and that always

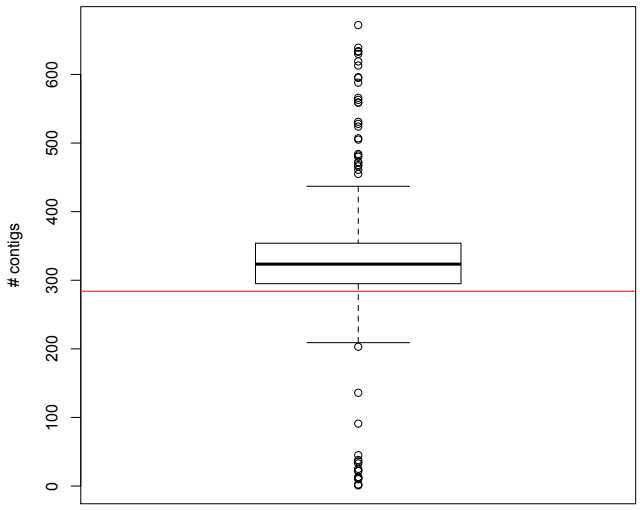

Figure 3.2: MEGAHIT - The red horizontal line represents the default number of contigs at 284.

grows (0.507834). Figure 3.3 shows a distribution of OVs with the `x-axis` representing OVs, and the `y-axis` representing the count. Note that the left most bar (a zero OV) is elided to fit into this graph. We have identified the default growth on the graph. It turns out that this is not the most frequently observed (positive) OV. But it is second (705 occurrences) to a slightly higher OV that occurs 978 times. This suggests that the default values are not optimal for this very common organism.

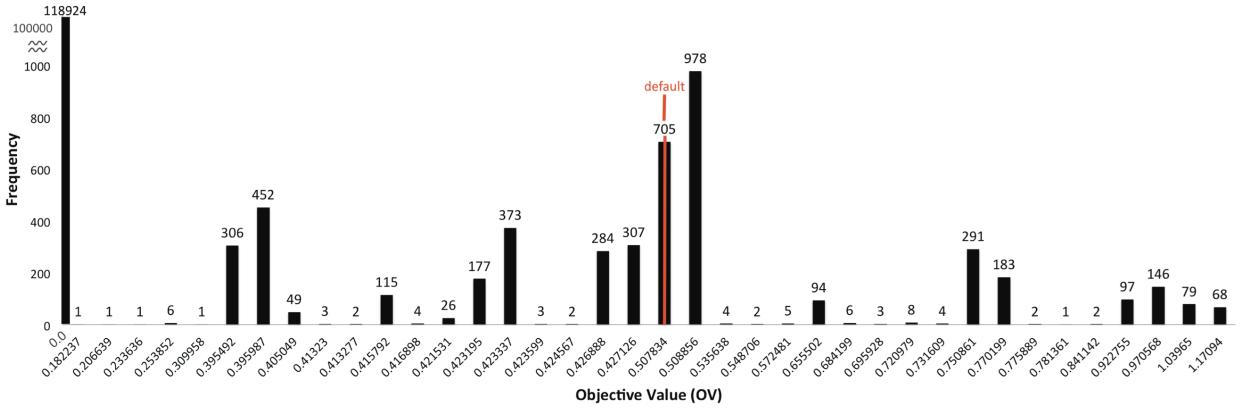

Figure 3.3: MFA - Bar chart of the frequency of OVs returned from the FBA analysis.

FBA-GUI For these 25,000 tests we saw four unique objective values (Table 3.8). Compared with the FBA-MFA tests this has less variation. However, the majority (67%) still do not grow. Only 18.40% match the online tutorial with default values, and interestingly, we can not obtain the higher OV values using the GUI variant of this application suggesting that the configuration options of the command line tool provide additional functional benefit to this application (or they are modifying the "fixed" media indirectly).

Table 3.8: FBA-GUI - Functional Variance

OV	Frequency
0	16,808 (67.23%)
0.395492	2,048 (8.19%)
0.423195	1,536 (6.14%)
0.507834	4,608 (18.43%)

Summary of RQ2 (a). We saw functional variability in all of our subjects. The implication for reporting scientific results may be significant. The greatest variability was in MEGAHIT signaling the configuration options in the model may be related to functionality.

3.3.3 RQ2 (b): Performance

BLAST For BLAST we did not see much variation in the runtime. Most configurations run in 6 seconds (2,959 test cases) and the remaining 41 tests run in 11 seconds.

MEGAHIT We see a large variation in the runtime of the tests that produce assemblies. Figure 3.4 shows a boxplot with a median runtime of 313 seconds and a range from 145 to 1,148 seconds. All tests that lead to an error run in 39 seconds.

FBA-MFA The runtime varies by configuration from less than 1 second to the timeout of 1,000 seconds. 99.50% of the tests finish in under 100 seconds. We also see a variance in runtime within the same OV shown in Figure 3.5 with the `x-axis` representing the OV values, and the `y-axis` representing the runtime in seconds. Most of the OV values have a median runtime between 6 and 40 seconds, but there are many cases of outliers that take significantly longer to run. This tells us we can achieve the same objective value in some cases in 1 or 1,000 seconds. We also confirmed

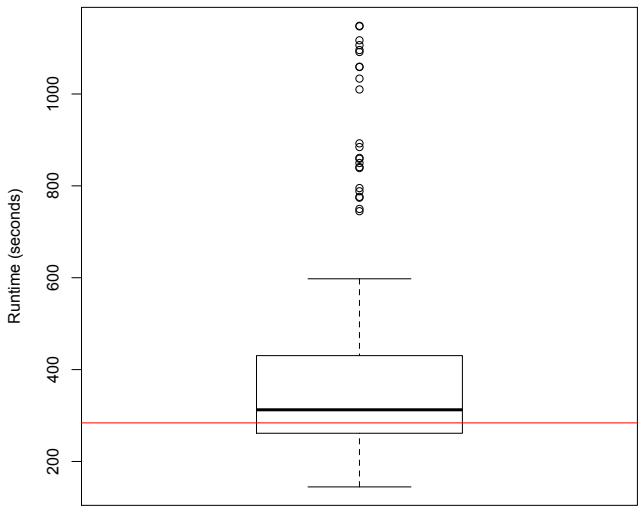

Figure 3.4: MEGAHIT - Runtimes. The red horizontal line represents the default runtime of 284 seconds.

these runtimes are consistent. We randomly picked four OVs and ran all of their combinations (562 in total) 10 times each. There was a low variance across each the 10 runs (median of 2.5 seconds).

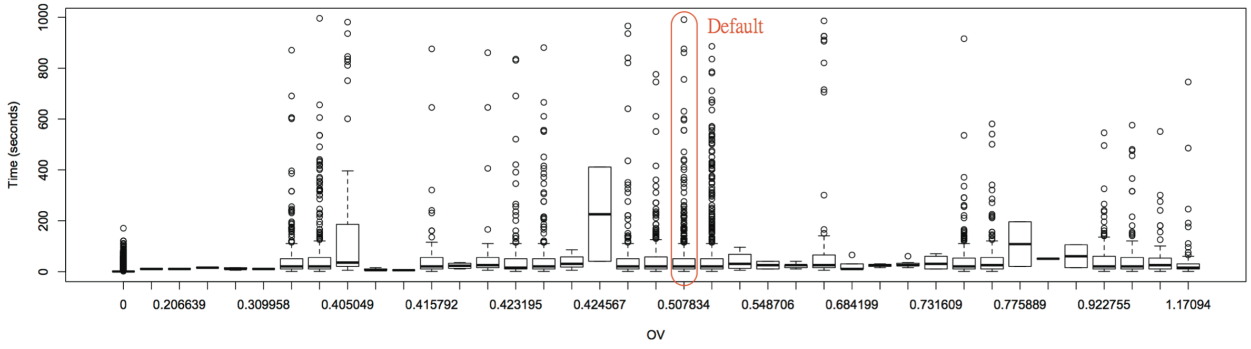

Figure 3.5: FBA-MFA - Boxplot runtime per OV

FBA-GUI All of the configurations finished in less than 42 seconds with the majority (99.28%) finishing in under 12 seconds. We see the least amount of variability when the OV is zero (the model does not grow). However it can still fail to grow but, take up to 21 seconds. The non-zero OVs

all have a median runtime of 6 seconds, but can range up to 41 seconds. Similar to the FBA-MFA model, we see we can achieve the same OV with different runtimes as seen in Figure 3.6.

Summary of RQ2 (b). For two of the applications, the performance was not impacted by modifying configurations. However in FBA the performance varies quite a bit by configuration and it is possible to get the same OV with very different runtimes.

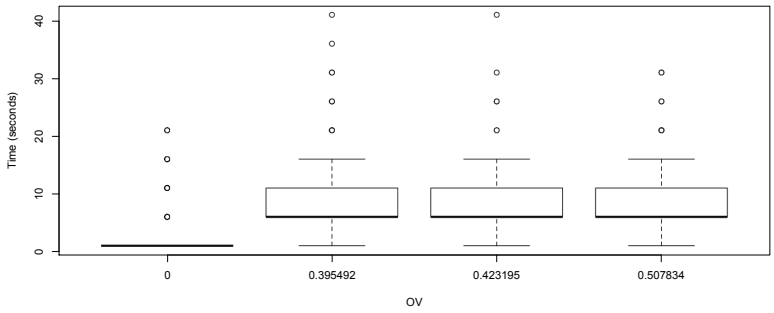

Figure 3.6: FBA-GUI - Boxplot runtime per OV.

3.3.4 RQ3: Sampling

For this RQ we wanted to understand if some common sampling techniques can be used for efficiency. We leave a full analysis as future work. We only present data from the two FBA tools since we could not exhaustively test those configuration spaces.

FBA-MFA Results can be seen in Table 3.9. For functional output we see that a 2-way CIT sample only produces three unique objective values compared to the 39 we saw in our large (125,000) sample. We don't see at least half of these unique OVs until we scale up to a 5-way CIT sample. Similarly, if we analyze the samples for errors we don't consistently see all five errors, until the 4-way CIT sample. This suggests that the software has higher order interactions. The option-wise testing method only captured five unique OVs and only one of the errors.

FBA-GUI Results can be seen in Table 3.10. The total number of unique OV values (4) in all the samples, show that the CIT samples can sample the GUI functionality. The SCIP error doesn't show up consistently (in at least half the runs) until the 5-way CIT sample.

Table 3.9: FBA-MFA Sampling Results

	2-way CIT (30 runs)	**3-way CIT** (30 runs)	**4-way CIT** (1 run)	**5-way CIT** (1 run)	**6-way CIT** (1 run)	**Option -wise** (1 run)	**Random** (1 run)
Average # in sample	43	324	2,423	14,346	79,372	78	125,000
Average # uniq OVs	3	10	19	29	40	5	39
Errors (times seen in total # of runs)							
Timeout	4	24	1	1	1	0	1
SCIP	0	9	1	1	1	1	1
Aborted	4	15	1	1	1	0	1
Segmentation	0	7	1	1	1	0	1
Negative 0	3	16	1	1	1	0	1

Table 3.10: FBA-GUI Sampling Results

	2-way CIT (30 runs)	**3-way CIT** (30 runs)	**4-way CIT** (30 runs)	**5-way CIT** (1 run)	**6-way CIT** (1 run)	**Option -wise** (1 run)	**Total Configs**
Average # in sample	25	152	6,759	3,129	6,250	24	25,000
Average # uniq OVs	4	4	4	4	4	4	4
Errors (times seen in total # of runs)							
SCIP	0	2	14	1	1	0	1

Summary of RQ3. The 5-way CIT samples detected functional variance and all errors using only a fraction of the configuration space, however the lower strength CIT was not as effective. Option-wise testing did not prove effective in this case. The results do suggest there is potential for sampling to help improve efficiency.

3.4 Discussion and Lessons Learned

Further Developer Feedback One of the developers of the FBA tools was very responsive to our information sharing and confirmed that this type of analysis would be useful for developers. He confirmed that many of these errors are due to real faults in the system. Some were unsurprising to him because they were due to combinations of configuration options that were never meant to be run together, some others were puzzling and are being examined now to find the root cause for a bug patch. We believe the first set of errors can be fixed by either preventing their combination

from happening via code, or through better documentation. The second class of errors will lead to program repair - both valuable additions.

He provided us additional feedback including:

- It was surprising to see some particular configurations result in a positive OV value meaning the organism grew. According to his expertise these should not have grown signaling a possible bug.
- He was surprised that some cases of a zero OV (no growth) took a significant time to run, suggesting cases of no growth are not currently optimized in the code.
- He remarked that the SCIP error was both "interesting" and "confusing" and asked for more analysis to help them locate the bug(s).

In summary, the developer was able to (1) learn about unknown errors and (2) learn about new ways to potentially optimize runtime. He also asked us for more details of our analysis and is actively working on repairing these issues. Finally, he asked us to run additional experiments with his models to explore some additional configuration options.

For MEGAHIT, the developer plans to update the documentation due to our feedback which may help to improve user understanding and experience of that tool.

3.4.1 Lessons Learned

We learned several valuable lessons about configurable bioinformatics software, some of which we believe can be generalized to other scientific software.

1. The meaning of configuration options and constraints between options are often not clearly documented or easy to extract even by experienced users.
2. Changing configuration options can have a large effect on functionality and it is hard to determine from documentation which configuration options matter.

3. Determining an exact functional output (oracle) for the problem was not straight forward. This suggests that the results of some experiments using these tools can be left open to interpretation.

4. The same functionality can be achieved with varying performance.

5. Unknown and unexpected errors exist in these systems under specific configurations of parameters.

6. Sampling may be an effective tool for testing these systems, but needs more exploration.

3.4.2 Suggestions for Improvement

1. **Create clear distinctions between functional and performance parameters.** Despite our effort to exclude functional options we were unsuccessful. We suggest developers divide configuration options into classes by their intended impact on functionality and document this well.

2. **Provide automated and transparent methods for handling dependencies.** We found dependencies that caused errors, and one tool auto corrected to fix dependencies. Both caused problems. We believe that developers should provide automated techniques to avoid incompatible options, but that the dependencies should also be clearly explained to the end user.

3. **Provide automated tuning algorithms.** Despite our combined expertise with these tools and our discussions with expert developers, we found that there was no explanation for some of the behavior we observed. There is an opportunity for developers to create automated tuning algorithms to guide all levels of users when manipulating and exploring configuration spaces.

4. **Developers can benefit from automated configuration-aware testing techniques.** Our experiments suggest that developers can utilize existing automated configuration-aware

testing tools to build more dependable software. Not only can it help them find latent faults, but it can help to understand the differences in functional and performance variance.

3.5 Conclusions and Future Work

Bioinformatics software is increasingly being used by different types of users and has become highly configurable. We have studied this issue to learn how we can build automated techniques for these tools. We show via a case study that manipulating configuration options in three popular bioinformatics programs can lead to lots of variability. We find that the functionality of these systems are dependent on the chosen configuration options. We also find that understanding the configuration models and underlying constraints is non-trivial and that developers can benefit from configuration-aware testing tools. Last we see that common existing sampling techniques may be beneficial. We have provided a set of actions that developers can take to improve the repeatability and dependability of these systems.

In future work we plan to build automated approaches to help address the problems that we found. We may leverage existing frameworks such as SPL Conquerer [32] which has already been used in the scientific domain [34] and will explore domain specific languages to help the users. We also plan to expand our models to incorporate more complex constraints. Doing so may require adapting work from [105]. Finally, our approach to creating the configuration models was largely manual. In the future we plan to automate this process as seen in other work [106].

CHAPTER 4. END USER INTERPRETABILITY FRAMEWORK

In Chapter 3 we motivated the need for configurability awareness in bioinformatics software tools. In this chapter we present a framework for understanding the effects of configuration options. Our method reduces the complexity of information reported to the user leading to what we call *interpretable configuration options*. We evaluate this framework on three subjects, and compare to a state of the art prediction tool.

4.1 Introduction

Scientists are becoming heavy users of computational tools. They rely on such tools to make scientific discoveries and to validate their laboratory findings. The tools they utilize have been developed to provide both flexibility and generality across a wide range of problems and types of data. When users interact with these tools they often have a plethora of configuration options they can choose from, some of which can change the outcome of their computational result, and others which can change the system's performances (such as runtime and CPU usage). It is unlikely that end users fully understand the impact of manipulating all configuration options. In Chapter 3 we detailed some potential pitfalls and implications of configurability in common tools used for bioinformatics. We found that configuration options can have a large effect on functionality, and it is hard to determine from the existing documentation which configuration options actually matter. Furthermore, the answer can change depending on the question the user is asking. While there has been a significant amount of research on software testing for configurable software, and there is research that aims to build performance models which can predict the behavior of non-functional properties, there has been little work that addresses the needs of the end user. Simple questions such as:

"*What does manipulating a particular configuration option do to my output?*",

"*What other configuration options interact with this one if I change it?*", and

"*Is there a better configuration for my specific use case?*"

are still non-trivial to answer.

The aim of this work is to provide *interpretable configuration options*. Interpretability has been increasingly studied in the field of machine learning for example in *explainable machine learning* [107], *explainable artificial intelligence* [108, 109], and *interpretable machine learning* [110–112]. Often the terms *explainability*, *interpretability*, and *understandability* are used interchangeably, however *interpretability* has gained more traction in recent years [110]. In this work we view *explainability* as more broadly explaining how the internal algorithm works or explaining the reason for a specific machine learner's output, whereas *interpretability* is presenting a solution in a format that is understandable for human comprehension. We focus on the later definition. Interpretable machine learning leads to an increase in transparency of the resulting models that has a benefit for both the end user and the developer [111]. Due to the added transparency, the end user gains more trust in the models and are more likely to utilize them, and developers can better determine why certain cases may result in poor prediction that can be resolved to improve the models. We see the same benefit potential in what we call *interpretable configuration options* in the domain of software engineering. Interpretable configuration options can help end users understand the effects of options to better make experimental decisions, and developers can better understand the effects of their options and identify any unexpected effects.

Interpretability has been descried as the union of three goals: understandability, accuracy, and efficiency [113]. We address understandability by displaying the effects of configuration options in a simple tabular format sorting by the most relevant results at the top. Our approach dynamically runs software tools covering accuracy by providing real measurements. We aim to achieve efficiency by heuristically sampling the configuration space using an iterative algorithm which prioritizes configurations that are likely to result in a positive effect on the program's output.

4.2 Motivation

In this section we discuss state of the art methods in handling configurability, and motivate our work by highlighting their limitations.

4.2.1 State of the Art

We will highlight two types of techniques and discuss their challenges for the end user; (1) optimization, and (2) prediction.

Optimization. The goal of optimization is to produce at least one configuration that optimizes the goal of the user (this could be a performance metric such as run-time, or a functional output). For example ACOVE is a tool for finding the optimal configuration options of `gcc` [114]. It uses a genetic algorithm to find a configuration that compiles the given code in the quickest time. One draw back of this approach is that the user doesn't gain information on any of the other configurations. Consider if the user has a constraint, they may need to know alternative configurations in order to adhere to their constraint. For example assume ACOVE returns `-frename-registers` as a configuration option to achieve the quickest compile time when running `gcc`.[1] This option makes use of left over registers for efficiency, but this configuration option results in debugging becoming impossible[115]. If the user requires debugging, this optimized configuration will be invalid for their use case.

Another consideration is that in many cases the actual values of the configuration options are obfuscated from the user or missing in documentation. As an example, we had to eliminate configuration options from the models in this study because we could not locate a default value. An advanced user may seek out the values, but a novice user is left without any clues. An optimization method may also ignore the distance (in terms of how many configuration values differ) between the optimal and current configuration. However, research shows that end users prefer to stay close to the default configuration [116].

[1] We remark this option is one of the options enabled when using the level 3 built in optimization method (`-O3`) in `gcc`.

Prediction. Another technique is prediction which uses a model to estimate results based on a set of variables. Regression is one example of a prediction method. Consider the tool SPL Conqueror [32, 33] which builds a regression equation to predict the effect any given configuration has on performance. Given a small model with one configuration option A such as in Equation 4.1, it is easy to see that if we have A change from 0 to 1, the output would increase by 5.

$$8 + 5 \times A \tag{4.1}$$

However these regression equations are typically much larger and have terms of degree higher than 1. Some of the best models produced in this work have over 30 terms with highest degree of 6. Consider a relatively simple example pulled from our results seen in Equation 4.2.

$$\begin{aligned} 1 + 2725.26 \times A \times B + -2725.26 \times A \times B \times C + 2732.64 \times A \times B \times D + 2793.86 \times A \times B \times E \\ + - 6972.97 \times A \times B \times F + -302.81 \times A^2 \times B \times G^2 + 302.81 \times A \times B \times H \end{aligned} \tag{4.2}$$

If every parameter is set to 1 as a default, how does the result change if B is changed to 0? These could be numeric options as well; what happens if A is changed to 2? These questions can be answered, but not without time to compute the effects. This format is not ideal for a user who wants to quickly be able to interpret the effect changing configuration options will have. Furthermore, this assumes the user knows the values of all configuration options A-H. We know that this isn't always known information to a user, and the user could even be mistaken about values (for example due to a hidden constraint). In Chapter 3 we found that the default value for the configuration option `k-step` in the tool MEGAHIT was incorrectly listed as 12 (the actual default value is 10), and the option `k-list` was involved in a hidden constraint.

This mirrors a classic problem in machine learning. In machine learning the primary objective is to be the most accurate, which can come at the cost of understanding how one gets to that result and has led to research in *explainable artificial intelligence* [109] and *interpretable machine learning* [112]. Furthermore, prediction methods are estimators; each prediction model will have an error associated with it (how far off is the predicted value from the true value).

4.2.2 Towards Interpretability

The fundamental goal we aim to achieve in this work differs from both optimization and prediction. Our goal is not pure optimization where we sacrifice a variety of solutions, nor is it prediction where we sacrifice interpretability and potentially accuracy. We aim to address user concerns that are missed by current methods through: (1) providing the user with multiple solutions to choose from, (2) providing configuration effect information in a transparent and easy to read format, (3) prioritize configurations close to their current configuration, (4) and provide this information with minimal overhead.

In this chapter we present a framework called ICO for *interpretable configuration options*. The core algorithm of our framework is a variant of the One Factor At a Time (OFAT) technique (see Chapter 2.1). Starting at a baseline configuration (such as the default), we iteratively test configuration options, and use a heuristic for higher-order configurations.

The contributions of this work are:

- A framework we call ICO for providing interpretable configuration options to an end user;
- Study evaluating ICO on two bioinformatics tools and one software engineering tool; and
- A study and discussion comparing our method to the state of the art in prediction modeling.

The rest of this chapter is laid out as follows. In the next section we discuss related work. We present our framework — ICO — in Section 4.4. In Section 4.5 we evaluate our framework on three scientific software systems and four different inputs in total. We then present results (Section 4.6) and present related work and conclude in Section 4.7.

4.3 Related Work

SPL Conqueror is an existing tool for performance-influence modeling [32, 33] and has been applied to scientific domains such as stencil codes [34, 52] and to multigrid methods, a technique for solving partial differential equations [35] (see Section 2.1 for additional background). Related work

investigates how different machine-learning techniques and sampling strategies affect performance prediction models [36, 38]. This work differs from ours in that they exclusively look at machine learning methods for prediction whereas we present a framework based solely on real measurements.

Zhang and Ernst [117] present *ConfSuggester*, a tool to troubleshoot configurations errors due to software evolution using diffs of execution traces. Their work differs in that they only look at a single configuration and use static analysis on multiple version of software tools in order to diagnose errors. Ko et al propose a debugging paradigm called *interrogative debugging* [118–120]. In interrogative debugging the questions "why did" and "why didn't" are asked. This differs from work in that we want to know the "what if". Another difference is that their focus is on debugging (a developer task) and not using (a user task). Xu et al. discuss the issues attributed to "over-designed configuration in system software" [116], but they do not provide any techniques for configurability management. Finally, the tool *Config2Code* is a helper tool for developers to better manage their configuration options [121]. Our focus is on the user's needs after configurations have been implemented.

4.4 ICO Framework

Next we present our framework for interpretable configuration options called ICO.

4.4.1 Distance

In order to talk about how far away one configuration is from another we need to define a metric of distance. We use *Hamming distance* [122], which measures how many symbols are different between two aligned words of the same size. Two equal words are distance 0. For each symbol that differs between the two words, a value of 1 is added. Consider the following example of two words of size 3:

$$\text{CAT} \;\text{-}\; \text{CAT} \rightarrow \text{Hamming distance} = 0$$

$$\underline{\text{C}}\text{AT} \;\text{-}\; \underline{\text{M}}\text{AT} \rightarrow \text{Hamming distance} = 1$$

$$\underline{\text{CAT}} \;\text{-}\; \underline{\text{FUR}} \rightarrow \text{Hamming distance} = 3$$

For simplicity we will refer to a configuration distance 1 away from the starting configuration as *D1*, distance 2 as *D2*, and distance 3 as *D3*. Distance 0 is the starting configuration.

4.4.2 Algorithm

Our algorithm operates iteratively starting from a given configuration (such as the default configuration), exploring new configurations one step further away from the starting configuration in each step. A summary of our algorithm operating on example data can be seen in Figure 4.1. In this example we have three configuration options $CO1$, $CO2$, and $CO3$. $CO1$ is Boolean with values {True,False}. $CO2$ is a numeric option with values {1,2,3}, and $CO3$ is of the enumeration types and can take on three string values {a,b,c}. Pseduocode for our algorithm can be seen in Algorithm 1 and Algorithm 2.

Algorithm 1 ICO Framework

```
1: procedure MAIN
2:     Configuration Options CO ← co_0, co_1, ..., co_x
3:     Values V[x] ← [[v_0, v_1, ..., v_y0], ..., [v_0, v_1, ..., v_yx]]   ▷ values for each option in CO
4:     Default D[x] ← {∅_0, ..., ∅_x}                                      ▷ soft-default
5:     result_D ← RunProgram(D)
6:     PosOpts1 ← ∅                          ▷ Set of tuples for option and effect of distance 1
7:     PosOpts2 ← ∅                          ▷ Set of tuples for option and effect of distance 2
8:     PosOpts3 ← ∅                          ▷ Set of tuples for option and effect of distance 3
9:     Distance1(D,PosOpts1)
10:    Distance2(D,PosOpts2)
11:    Distance3(PosOpts1,PosOpts2)
12:    return (PosOpts1,PosOpts2,PosOpts3)
```

We walk through the steps of our algorithm using the example in Figure 4.1. Though ICO can begin with any starting configuration defined by the user, we will use the default configuration in our example.

Step 0: Default Configuration In the initial step, we run the starting configuration (default) though the program to get the baseline output. This value is used in the subsequent steps to determine if the effect of a new configuration is equal, greater than (a positively effecting configuration), or less than (a negatively effecting configuration) the starting configuration[2].

[2]In the case of minimizing the output, greater than represents a negatively effecting configuration and less than represents a positively effecting configuration.

Algorithm 2 Distance Procedures

```
1: procedure DISTANCE1(initial_config C, pos_opts PO)
2:     for i ← 0 to x do                                   ▷ For every configuration option
3:         for j ← 0 to y_i do                             ▷ For every configuration value
4:             c ← C                                       ▷ Reset the configuration
5:             c[i] ← V[i][j]
6:             result ← RunProgram(c)
7:             if result > result_D then
8:                 effect ← result − result_D
9:                 PO.insert(v[i][j], effect)
10: procedure DISTANCE2(starting_config C, pos_opts PO)
11:     for i ← 0 to x do                                  ▷ For every configuration option
12:         for j ← 0 to y do                              ▷ For every configuration value
13:             c ← C                                      ▷ Reset the configuration
14:             c[i] ← V[i][j]
15:             Distance1(c,PO)        ▷ Take all steps distance 1 from this new configuration
16: procedure DISTANCE3(pos_opts PosOpts1, pos_opts PosOpts2)
17:     for P_1,CO_1 in PosOpts1 do                        ▷ For all positive effects of distance 1
18:         for P_2,CO_2 in PosOpts2 do                    ▷ For all positive effects of distance 2
19:             c ← C                                      ▷ Reset the configuration
20:             c[i] ← V[position(P_1)][O_1]
21:             c[i] ← V[position(P_2)][O_2]
22:             result ← RunProgram(c)
23:             if result > result_D then
24:                 effect ← result − result_D
25:                 PosOpts3.insert({v[P_1][CO_1], v[P_2][CO_2]}, effect)
```

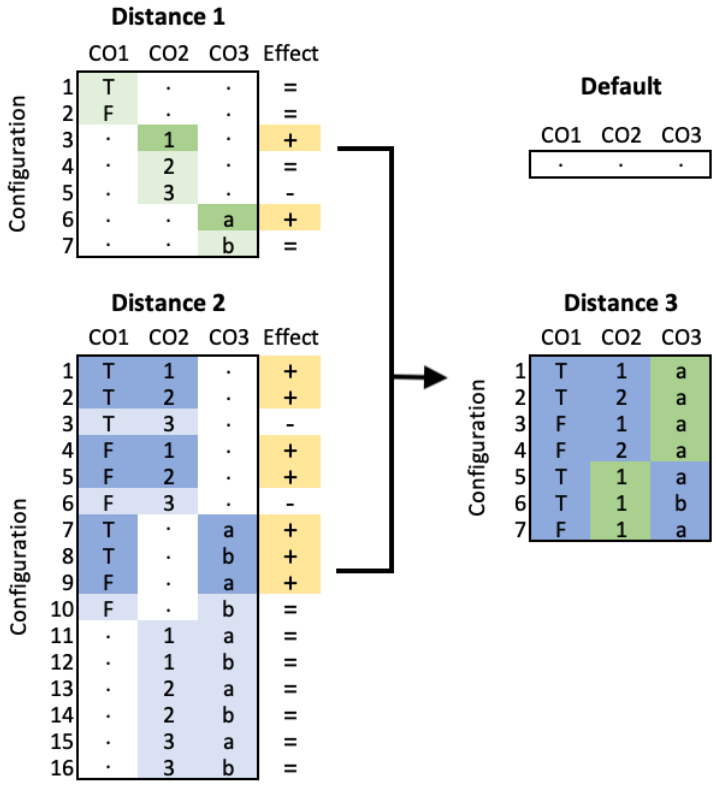

Figure 4.1: The three steps of the ICO-Framework. The default configuration has no configuration option set. We test each value of all configuration options in isolation in D1. In D2 we test all pairs of configuration options. In D3 we combine all the configuration from the D1 and D2 that has a positive effect on the functional output.

CO: Configuration Option

Step 1: Distance 1 (D1) We iterate over each value for each configuration option, changing them one at a time. In Figure 4.1 we see the first two configurations (1 and 2) under D1 are set to both possible values (True and False) of configuration option $CO1$. The remaining configuration options stay at their default value. This allows us to identify which configuration options have an impact on the output in isolation. We run all D1 configuration through the subject.

Step 2: Distance 2 (D2) In order to begin to investigate interactions we perform a D2 analysis to test pairs of configuration options. To avoid running unnecessary configurations, we begin by modifying the configuration model to remove any values that cause an error, a warning explicitly

citing the option has no effect, or a constraint with the default configuration. This is a heuristic decision made to prevent those errors or timeouts from propagating to a large number of configurations in **Step 2**. Then again starting from the default configuration, we iterate through all pairs of configuration options (and their values). As seen in Figure 4.1 under D2 we iterate over all pairs of configuration values. We run all D2 configuration through the subject and report all configurations and their values that result in a positive effect on the functional output.

Step 3: Higher-Order Interactions (D3) Previous research has shown that in software interactions between configuration options tend to follow a hierarchy [32]. So in order to investigate higher-order interactions, we follow this logic. As a heuristic, we reduce the number of configuration to measure by only combining the configuration options that had a positive effect on the output. We combine each positively influencing configuration value from **Step 1** with the each positively influencing configuration value from **Step 2** to generate a set of D3 configurations. In our example in Figure 4.1 the positively affecting configurations appear on a darker background and the effect is highlighted. All combinations between D1 and D2 can be see in the D3 table on the right.

4.4.3 ICO Implementation Details

The ICO framework takes a starting configuration as an input. We design ICO to be able to handle any configuration as a starting place, however for this study we choose the default configuration. Since the aim of this work is to investigate how the average end user interacts with configurable tools, the default is a natural starting place.

A default configuration can either be run explicitly by setting all of the configuration options, or implicitly by setting no configuration options. Setting all configuration options to `NULL` (meaning not explicitly setting the option) is one method of running the default configuration we refer to as a *soft-default*. We can also explicitly set all configuration options to their default which we call a *hard-default*. For example, if a user runs `tail [FILE]` and `tail [FILE] -n 10` the result is the same (the last 10 lines of `FILE` are printed to standard out) because the default value for the

configuration option `-n` is 10. In this example `tail [FILE]` is the soft-default and `tail [FILE] -n 10` is the hard-default.

It is natural to assume the hard- and soft-default configurations would produce equivalent results, however this is not always the case; there can be interactions in the lower level code. Take for example the hidden dependency discovered in Chapter 3 in MEGAHIT. In that example there were three configuration options — `kMin`, `kMax`, and `kStep` — which define another option called `kList` (were not explicitly manipulating `kList` in that study). The default values for each are 21, 141, and 12 respectively making the `kList` {21,33,45,57,69,81,93,105,117,129,141}. However, it turned out that if you didn't explicitly set any of those three configuration options, `kList` is defaulted to {21,29,39,59,79,99,119,141}. This is one example of why the soft-default and hard-default may not match. Other causes could be incorrect documentation (wrong default value is listed), misleading documentation (like the previous example), or hidden dependencies in the code. All three of which we saw in Chapter 3.

We remark that even though we may see hidden effects due to configuration changes, in this work we use soft-default configuration options. Our reasoning is simple – this work is aimed at the end user experience, and the typical end user will not explicitly set the default values for all configuration options.

4.5 Study Methodology

We evaluate the ICO framework to investigate if it leads to more interpretable software by asking the following research questions:

RQ1. Can we convey information to assist in choosing of configuration options based on user goals?

The user goal we evaluate is improving the functional output by manipulating the fewest configuration options. We display these results in a simplistic format to the user, and evaluate the reduction of configuration options and values a user is presented with.

RQ2. Is the information ICO provides to a user useful? Although we do not perform a user study, we ask if we can address common questions asked by users. To do this we ask if ICO identifies configuration choices that improve over the starting configuration, how far away from starting configuration is this improvement, and how do the result compare to a proxy for optimization.

RQ3. How does the state the art in prediction compare to ICO?

We ask if SPL Conqueror is effective at achieving our defined user goal; we evaluate SPL Conqueror in two different ways with following sub questions:

(a) What is the accuracy of SPL Conqueror trained on various samples?

(b) Can SPL Conqueror identify the effects of configuration options in isolation and in pairs (D1 and D2)?

4.5.1 Subjects

We select three study subjects, two are bioinformatics programs — BLAST and MFA — from our previous study in Chapter 3, both of which are highly-configurable. To test if our work can extend beyond bioinformatics, our third subject is a software engineering tool used for performance prediction — SPL Conqueror. We use SPL Conqueror as both a subject, and as a tool to compare against our framework. We refer the reader to Section 2.1 for more information on SPL Conqueror.

BLAST. Our first bioinformatics tool is the genome annotation tool BLAST which is the most commonly used bioinformatics tool for alignment of DNA sequences. We used version 2.7.1 and ran all experiments with 12GB memory on a single node with a 15 minute timeout (sufficient for most cases). We use the number of quality hits as our functional output. We consider a hit to be *quality* if it has 100% Identity and 0.0 e-value. We note that this is only one method of interpreting quality, our framework is agnostic to the metric but we must choose one for this study. We leave it as future work to explore other functional output metrics.

MFA. Our second bioinformatics subject is the metabolic modeling tool Flux Balance Analysis which simulates the growth of an organism in a media. We use a command line implementation

called the MFAToolkit (MFA for short) due to its large configuration space. We ran all experiments with 3GB memory on a single node with a 15 minute timeout (sufficient for most cases). The functional output for MFA is the amount the organism grew measured by the *Objective Value* (OV). We use the same input data as Section 3.2.1 for these first two subjects.

SPL Conqueror. Our third subject is a software engineering tool used for prediction modeling called SPL Conqueror [32, 33]. Based on a set of input configurations and measurements (measurements are the result of the configuration after running the subject program), SPL Conqueror will create a regression equation that can be used to predict the program's output on future configurations.

We choose two input models from a previous SPL Conqueror publication [32]. Their subjects included both synthetic and real-world systems. We chose two of the real-world systems for more realistic experiments. Due to long runtime of SPL Conqueror we choose two of their smaller subjects, the multi-grid solver Dune and and image processing framework called Hipacc.

A primary use case of prediction methods is to achieve the lowest accuracy, therefore we use the *error rate* as our functional output. Note that we aim to minimize this output (we aim to maximize our other subjects). Since SPL Conqueror can be computationally intensive we used a 24 hour timeout as a cutoff. We find that was more than sufficient for the Dune subject, and minimal number of Hipacc subjects timed out. We used the docker version[3] of SPL Conqueror and ran all experiments with 50GB memory on a single node.

All subjects were run on an Intel(R) Xeon(R) Gold 6244 CPU 2.60GHz RedHat Enterprise Linux 7 Server [123].

4.5.2 Configuration Models

A summary of all three configuration models can be seen in Table 4.1. A sample of the following configuration models is displayed below. To see the full configuration models please refer to Appendix A. Our configuration models range from 22 to 31 configuration options. Each model has

[3]Docker image pulled on February 17th 2020 from christiankaltenecker/splconqueror on dockerhub. Digest:sha256:b88a98b1da34cf250303e564efea552a11f2b11aad9536724b6f354cdb3b870e

at least one of the three types of values: Boolean/flag, numeric (integer or float), and enumeration described in Section 2.1.

Table 4.1: Summary of configuration models for BLAST, MFA, and SPL Conqueror

	BLAST	**MFA**	**SPL Conqueror**
Number of configuration options	22	31	26
Number of total values	76	68	61
Size of complete configuration space	9×10^{10}	1.64×10^{13}	2.58×10^{9}
Boolean/Flag	10	21	16
Numeric	10	9	9
Enumeration	2	1	1

For the BLAST model, in order to include as many options as possible we only removed an option if: (1) it was of type string (e.g. a file path), set, list due to difficulty in partitioning, or (2) no default value was found. We also removed the options `evalue` and `perc_identity` since we use those values to compute our functional output. A sample of the configuration model for BLAST can be seen in Table 4.4; the full model consists of 10 Booleans, 10 numeric, and 2 enumeration types. Note this configuration model has been expanded from seven to 22 configuration options compared to the model in Chapter 3.

Similarly for the MFA model we did not use a configuration option if it was of type set or string, or if the type was unknown. A sample of the configuration model can be seen in Table 4.5; the full model consists of 21 Booleans, 9 numerics, and 1 enumeration type. For the full names of the configuration options as found in the source code file please see Appendix A.7.

The configuration options in SPL Conqueror can be broken into three categories: machine learning, solver, and sampling. The machine learning options affect how the regression model is built. The solver options determine which of two solvers are used (Z3 or Microsoft Solver Foundation). The sampling options affect what sampling methods are used for both the binary and numeric options (some affect either binary or numeric and some affect both). We choose to fix the solver and we do not utilize the sampling functionality of SPL Conqueror. Therefore, we use only the machine learning options. A sample of the configuration model for SPL Conqueror can be seen in

Table 4.2: Sample of configuration models for each subject. Complete models can be found in Appendix A.

Table 4.4 Sample of BLAST configuration model

Option Name	Type	Values	Default
soft_masking	Boolean	TRUE, FALSE	TRUE
sum_stats	Boolean	TRUE, FALSE	FALSE
lcase_masking	Flag	TRUE, FALSE	FALSE
ungapped	Flag	TRUE, FALSE	FALSE
max_target_seqs	Numeric	1, 10, 100, 500, 1000	500
min_raw_gapped_score	Numeric	NULL, 0, 10 ,100, 1000	NULL
xdrop_gap	Numeric	0, 0.1, 0.5, 30, 100	30
xdrop_gap_final	Numeric	0, 0.1, 0.5, 10, 100	100
dust	Enumeration	yes, “20 64 1”, no	“20 64 1”
strand	Enumeration	“both”, “minus”, “plus”	both

Table 4.5 Sample of MFA configuration model

Option Name	Type	Values	Default
useVarDrainFlux	Boolean	TRUE, FALSE	FALSE
decompDrain	Boolean	TRUE, FALSE	FALSE
decompRxn	Boolean	TRUE, FALSE	FALSE
conObj	Numeric	0, .05, .1, .15, .2	0.1
maxDrain	Numeric	0, 250, 500 ,750, 1000	0
minDrain	Numeric	0, -250, -500, -750, -1000	-1000
minFluxMult	Numeric	0, 1, 2, 3, 4	1
solver	Enumeration	NULL, CPLEX, GLPK, SCIP, LINDO	NULL

Table 4.6 Sample of SPLC configuration model

Option Name	Type	Values	Default
parallelization	Boolean	TRUE, FALSE	TRUE
limitFeatureSize	Boolean	TRUE, FALSE	FALSE
quadraticFunctionSupport	Boolean	TRUE, FALSE	TRUE
epsilon	Numeric	0, 0.01, 0.5	0
baggingNumbers	Numeric	0, 3, 100	100
minImprovementPerRound	Numeric	0.01, 0.1, 0.25	0.1
scoreMeasure	Enumeration	RELERROR, INFLUENCE	RELERROR

Table 4.6; the full model consists of 16 Boolean, 9 numeric and 1 enumeration type. This model has 26 configuration options.

4.5.3 Handling of Enumeration Configuration Options

SPL Conqueror it is not designed to handle enumeration type options. In such cases we translate the enumeration option into binary options when creating the measurements file. For an enumeration type parameter EO with values $v_1, v_2, .., v_n$ we create n new Boolean configuration options (CO_n). The value of CO_n for a given configuration C is defined by Equation 4.3.

$$CO_n = \begin{cases} \texttt{TRUE} & \text{if } C[EO] = v_1 \\ \texttt{FALSE} & \text{otherwise} \end{cases} \tag{4.3}$$

For example in BLAST the configuration option `dust` can have the following values: `no`, `yes`, `‘‘20 60 1’’`. We define three new Boolean options called `dust_no`, `dust_yes`, and `dust_20641`. If a configuration has `dust` set to `yes` the new options would be set to `FALSE`, `TRUE`, `FALSE` respectively.

4.5.4 Metrics

In this section we define several metrics used in the results.

4.5.4.1 Additive Effects

Every configuration will have a measurement of *effect* which is the difference between its functional result, and the functional result under the default configuration (Equation 4.4).

$$\text{Effect}(\text{Configuration}_X) = \text{FunctionalResult}(\text{Configuration}_X) - \text{FunctionalResult}(\text{Default}) \tag{4.4}$$

When we consider configurations of more than distance 1 from the default, we define metric for an *additive effect*. A D2 configuration will have two configuration options that change from the default, for example {`A`,`B`}. Each of these options has an effect: Effect(`A`) and Effect(`B`). The D2 configuration also has an effect we will refer to as Effect(`AB`). The additive effect is the addition of

the effects for each subset of the configuration's options. For a D2 configuration this calculation is straight forward.

$$\text{AdditiveEffect}(\text{Configuration}_{\texttt{AB}}) = \text{Effect}(\texttt{A}) + \text{Effect}(\texttt{B}) \tag{4.5}$$

Note that the effect of a configuration option may be equal to 0 as well. It may be intuitive that this additive effect is always equal to the effect of the D2 configuration, but we find this is not always the case.

When we consider D3 configurations the additive effect becomes a set of effects. This is because there could be interactions between our configuration options. Consider a D3 configuration with options {A,B,C}. There are several different additions that could be occurring:

$$\text{Effect}(\texttt{A})+\text{Effect}(\texttt{B})+\text{Effect}(\texttt{C}), \quad \text{Effect}(\texttt{AB})+\text{Effect}(\texttt{C}),$$
$$\text{Effect}(\texttt{AC})+\text{Effect}(\texttt{B}), \text{ or } \quad \text{Effect}(\texttt{BC})+\text{Effect}(\texttt{A})$$

Again some of these effects may be equal to zero. These cases are covered in the above equation, but to be explicit, the effect of {A,B,C} could also be equal to any of the following:

$$\text{Effect}(\texttt{A}), \quad \text{Effect}(\texttt{B}), \quad \text{Effect}(\texttt{C})$$
$$\text{Effect}(\texttt{AB}), \quad \text{Effect}(\texttt{AC}), \text{ or } \quad \text{Effect}(\texttt{BC})$$

To generalize, we will consider the additive effect to be a set of the effects of the power set of the configuration options in the configuration as seen in Equation 4.6

$$\text{AdditiveEffect}(\text{Configuration}_{\overrightarrow{C}}) = \mathcal{P}(\text{Effect}(\overrightarrow{C})) \tag{4.6}$$

where $\overrightarrow{C}$ is a vector of all configuration options not set to their defaults.

In this work we measure if the effect of a D2 or D2 configuration is equal to (EQ), greater than (GT), or less than (LT) any of its additive effects. For a D3 configuration we measure if it is simple equal to (EQ) or not equal to (NEQ). This tells us if the effect of that configuration is due to an interaction between configuration options. Note that this is the equivalent of the subset sum problem.

4.5.4.2 Confusion Matrix

To compare our framework to state of the art tools in prediction modeling we utilize a confusion matrix which is a frequently used reporter of machine learning methods [124]. Consider a study where the default configuration on a program P gives us a result of 30. Now we run P on ten new configurations resulting in the following outputs {0,0,10,10,30,30,30,30,90,100}. In this case 4 of the configurations gave us a result less than the default (LT), 5 gave us a result equal to the default (EQ), and the last gave us a result greater than the default (GT). Now we use a prediction tool which predicts what the output will be for those same ten configurations. The tool predicts the following {0,0,0,0,0,30,30,30,30,100}. We can align these two sets of outputs to get a better view of the differences.

Actual ⟶	LT	LT	LT	LT	EQ	EQ	EQ	EQ	GT	GT
	0	**0**	**10**	**10**	**30**	**30**	**30**	**30**	**90**	**100**
	0	**0**	**0**	**0**	**0**	**30**	**30**	**30**	**30**	**100**
Predicted ⟶	LT	LT	LT	LT	LT	EQ	EQ	EQ	EQ	GT

There are nine different situations that can occur. If the actual result was LT, the predictor can be LT, EQ, or GT, this follows for the actual result being EQ or GT. We can represent these nine cases in a confusion matrix. Figure 4.9 shows a traditional confusion matrix where the predictor either predicts a yes/True (1) or a no/False (0). The four outcomes are: true negative (TN), false negative (FN), false positive (FP), and true positive (TP). Figure 4.10 shows what the matrix would look like in the above example. If the predictor were 100% accurate, all the values would fall on the diagonal as seen in Figure 4.11. But we can see in the matrix that there were two incorrect cases. In one case the actual output was EQ the default when the predictor predicted it would be LT which occurred in the fifth configuration, and when the actual output was GT but the predicted predicted EQ (ninth configuration). Note that this evaluation does not consider the scale of effects. For example the third configuration has an actual value of 10 when the predictor predicted 0, this is counted as a correct classification since both values are less than the default. Additional metrics could be used in future work, but this metrics allows us to address our end user use case which is

to identify *any* effects (and cases where the configuration is not equal to the default). We utilize this metric in Section 4.6.3.

4.5.4.3 SPL Conqueror Output Metrics

We report on several metrics from SPL Conqueror. The first is the relative error which is the average error rate over all the measurements:

$$\frac{|\text{measured} - \text{predicted}|}{\text{measured}} \times \frac{1}{sizeof(\text{measurements})} \tag{4.7}$$

We comment that the current implementation of SPL Conqueror can not take in a separate test data set to compute the error rate. Therefore the reported error rates are calculated on the training data. To address this we apply a second evaluation of SPL Conqueror (see results).

Next is the *model complexity* which is equal to how many variables are in the regression equation. For example if the regression equation is $(5 \times \texttt{root}) + (6 \times \texttt{CO}_A \times \texttt{CO}_A \times \texttt{CO}_B) + (4 \times \texttt{CO}_A)$, the complexity is 5. This complexity will help us determine how interpretable the equation will be to a user (the lower the complexity, the more interpretable). We note this does not include the degree of these terms (the highest degree term in the example is 3) which adds to the cognitive complexity.

4.5.5 Threats to Validity

4.5.5.1 External Validity

We acknowledge our evaluation only considers three study subjects and may not generalize to all software systems. To increase diversity our two bioinformatics tools come from different areas of research. We also add SPL Conqueror as a subject to see how ICO scales beyond bioinformatics software. We also acknowledge we fix one use case to each subject and different functional outputs may result in different results. We attempted to choose use cases that represent a common scenario for an end user, but leave additional use cases to future work.

4.5.5.2 Internal Validity

We also acknowledge the author chose the configuration options to be used in the models. We used documentation and asked expert users of the tools for assistance when needed. We only removed a configuration option in specific cases, however we acknowledge we do not have full knowledge of all options in these tools. We provide the exact models in our Appendix for transparency and reproducibility.

Configurations were run used automated scripts and could have been susceptible to faults. To address this we ran each program in its own thread and recorded the return value. If this return value was non-zero the test was marked as a failure and we either rerun or removed from the study. We mention these occurrences when applicable.

4.5.5.3 Construct Validity

We acknowledge the metrics we chose are not the only choices. To evaluate the errors of the regression models from SPL Conqueror we use the metrics provided by the tool itself. When we compare to our ICO framework we decided to use a standard metric: mean absolute error (MAE). We also recorded additional metrics such as mean squared error (MSE) and mean relative error (MRE) and provide these in the Appendix.

4.6 Results

Next we present our results for our three research questions.

4.6.1 RQ1: Can we convey information to assist in choosing of configuration options based on user goals?

We present the output of our framework, and ask what information it conveys to assist a user. We ask if we can reduce the number of parameters and options that will positively affect the output of the program and communicate that information to the user.

We summarize results for all four cases in Table 4.12. The first three rows show how many configurations resulted in a positive effect on the functional output under each step (D1, D2, D3). These are the configurations ICO reports to the user. We report on the percentage reduction as a metric for how much of the space is reduced from the user's consideration. Since D3 is a heuristic most of the reported configurations have a positive effect. In all four experiments we found configurations that led to a better result than the default. In all D1 tests we reduced the number of configurations a user has to look at by as much as 94.57% and as few as 88.52%. For example in BLAST, there are 76 difference configurations distance 1 away from the default, but only six of them have a positive effect on the result. When we test all D2 configuration the reduction is between 79.14% and 89.83%. This is still a large reduction in configuration, but less than D1. We see most of the D3 configurations return a better result, but not all. For example in BLAST, 9 of the D3 configurations returned a worse result. This shows that it might not always be better to combine positively affecting options together. We also saw this in the SPL Conqueror subjects.

Not only can ICO reduce the number of configurations a user has to look at, but it can reduce the number of configuration options. The next two rows displays how many configuration *options* have a positive effect. We again report the reduction since in our use case the user is focused only on configuration options that have the potential to improve their result. In BLAST, six configurations were found in D1 to have a positive effect. However they are a result of only 3 of the 22 configuration options (one configuration option `xdrop_gap_final` has a positive effect when set to any of four values: `0`, `0.1`, `0.5`, or `10`). We see similar results for our other subject. The greatest reduction was in MFA from 31 to 2 configuration options, and the smallest reduction in SPL Conqueror-Dune from 26 to 7. The reductions under D2 vary greatly between 53.85% and 90.32%. This signals we may want to explore more heuristics to further reduce this.

The next three rows show the *effect range* for D1, D2, and D3 showing the smallest and largest improvement found in each sample. Recall that we are minimizing the functional output (error rate) in the SPL Conqueror subjects. In BLAST under D1 the smallest effect added 3 hits to the results, and the largest effect adding 1,532 hits. We see the maximum effect increases in D2 and

D3, but we also still see small effects of adding only 2 hits. If we look at MFA, we see that the D1 configurations only had a maximum effect of +0.012. This might be too small for a user to value. However, if the user takes two steps away from the default they can increase their result by +33.395. Interestingly, if they take three steps from the default there is no better effect observed. In the other subjects we see the maximal effect continues to improve as distance is increased. However in SPL Conqueror-Hipacc the change is small moving from D2 to D3 only improved the result by 0.02. In that case the user may just want to stay distance 2 from the default.

The remaining rows show the additive effects, or how many configurations returned new interactions (GT or LT in D2, and NEQ in D3). Recall an equal (EQ) means the effect of that configuration is equal to a subset of the configuration effects of smaller distances representing no new interaction (see Section 4.5.4.1). We see the majority of D2 configurations have effects equal to their additive effects. This means they are simply a combination of the D1 effects. In the cases of GT or LT these are interactions occurring between configuration options. In BLAST we see 23 D2 configurations have an effect greater than their additive effect, and 20 less than than their additive effect. We also see new interactions in the D3 configuration for all subjects. In MFA, none of the D3 configurations were due to an additive effect. These new D3 interactions tell us there are higher-order interactions occurring in these tools.

ICO Output

We display the output of ICO for a sample of our results. All outputs are in Appendix B. Table 4.15 shows the output of ICO for D1 and D2 configurations for BLAST. At the top the functional result under the default configuration is displayed. Below is all positively effecting D1 configurations. Here we see the 3 configuration options (`ungapped`, `sum_stats`, and `xdrop_gap_final`) and their values resulting in six configurations. Below those are the D2 configurations. The first result in D2 is an interesting case as the first configuration option (`xdrop_gap_final`) is reported in D1, however the second option `xdrop_gap` is new. This means a configuration opinion can have no effect in isolation, but interact with other options. Even more curious is that in isolation `xdrop_gap_final`

only increases the result by +3, but in combination it has the largest effect of any D2 configuration at +3478. In the D3 configurations (Table 4.16) we see a largest effect of $+10,923$ which occurs when we combine the best configuration from D2 with `sum_stats`.

Summary of RQ1

ICO can reduce the number of configurations a user has to look at by as much as 94.57%, and reduce the number of configuration options by as much as 93.55% by only reporting positively affecting configurations. This allows the user to only focus on the configurations and options that will improve their result, and will not waste their time investigating other options. This also helps a novice user who may not know where to begin. ICO can provide them will a reduced list of options to explore. We also find that combining positively impacting parameters does not always result in a better solution. Not only is it more convenient for a user to stay close to their current configuration, but it shows them that they would be wasting their time by trying more combinations. We find we can convey information to assist users in choosing configuration options using ICO by displaying configurations that improve the functional result. The results are provided in such a way to reduce the users effort by prioritizing them based on (1) distance from their current configuration and (2) scale of effect on their result.

4.6.2 RQ2: Is the information ICO provides to a user useful?

Although we do not perform a user study, we see if we can address common questions asked by a standard end user. We ask if ICO can identify configuration choices that improve over the starting configuration, and identify how close to the starting configuration these improvements are. We also compare ICO's best output to a proxy for optimization using random samples of the configuration spaces.

We run 100,000 random configuration of any distance from the default in BLAST and MFA. For BLAST one of these configurations resulted in one error four were duplicates leaving 99,995 configurations, and in MFA 2,205 configurations were removed leaving 97,795 configurations. Due

to resource restrictions we ran 1,000 random configurations though SPL Conqueror. A total of 16 configurations timed out in Dune and 9 timed out in Hipacc resulting in 984 and 991 configuration respectively. Though the random sample is not exhaustive, it will give us an idea of how close ICO can get us to the best value and can be used as a proxy for optimization.

A summary of results can be seen in Table 4.17. We look at the best performing configurations under D1, D2, and D3, and compare it to the best result seen in the random sample. We also record the minimal distance of the best performing random configuration to the default configuration.

In BLAST the output improves as we take further steps from the default. The best output ICO found was 10,923 and was distance 3 from the default. In 100,000 random configurations (of any distance from the default) the highest we see is 18,766 which occurred in 48 configurations (or 0.048%). In this case ICO was not able to find a configuration to achieve the best observed output. However, values higher than 10,923 (the highest seen by ICO) occur in only 2,227 (or 2.23%) of random configurations; even though ICO could not find the highest value we observed, it did reach a value in the top 97% of all random configurations. To obtain the minimal distance of the best random configurations, we compared the 48 configurations that resulted in the best output. We identified the following options were common to all configurations: `soft_masking`=`FALSE`, `ungapped`=`FALSE`, `word_size`=`28`, `strand`=`both`, `sum_stats=TRUE`, `xdrop_ungap`=`0.1` or `0.5`, `xdrop_gap`=`0`,`0.1`,or `0.5`, and `xdrop_gap_final`=`0`,`0.1`, or `0.5`. We confirmed we could achieve the optimal output value (18,766 tests) in as few as five configuration options[4].

In MFA we see a small impact on the effect distance one from the default, then a large impact at distance 2. However at distance 3 there is no improvement in the effect, in fact it lowers. In this subject the user may not want to explore any further than distance 2. The highest achieved OV found in the random configuration was 34.069 in the random configurations which ICO did find in the D2 configurations. We see this highest random value occurs in only 0.44% (430 times) in the random configurations. The majority of the random configurations (89,961 or 91.99%) returned a

[4] `blastn -db yeast_db/yeast.GCF -query yeast.nt.fasta_20000 -out aligns_test/best.alignments -soft_masking False -word_size 28 -strand both -sum_stats True -xdrop_ungap 0.1 -xdrop_gap 0 -xdrop_gap_final 0 -outfmt 6`

growth of 0. This tells us if the user randomly tried configurations they would likely result in no growth. We also saw this in Chapter 3. We repeated the same analysis as BLAST to identify the distance of the best random configuration and found it to be distance 2. Therefore, the minimal number of steps a user needs to take for an optimal configuration is 2. So in MFA, ICO was able to find the best solution and provide it in the shortest distance to the default configuration.

Recall we are minimizing the error rate in the SPL Conqueror subjects. In the Dune subject the error rate decreases in D1, stays the same in D2, and decreases again in D3. On the Hipacc subject the error rate continues to decrease in each step (though minimally from D2 to D3). However under the random configurations tested we see a much smaller error rate for both subjects. We only observed one configuration for each subject resulting in the best error rate found to be distance 18 away for Dune and 17 for Hipacc. Due to only observing one configuration for each, we remark that the distance might be smaller but requires additional experimentation to determine. We believe the distance would still be much greater than 3 regardless. Observing this high distance for the best result signals there may be a trade-off between interpretability and optimization.

Summary of RQ2

In BLAST, though ICO did not find the highest value observed which was 5 steps from the default, the best solution ICO found was in the top 97.77% of results in random. ICO did locate the highest value observed in MFA. This optimal configuration is only distance 2 from the default. ICO is useful to a user by providing them with a better result then the default, and in minimal number of steps away. In SPL Conqueror ICO was able to identify better configuration, but the best observed was a significant improvement. This potential optimal was a large distance from the starting configuration identifying a possible trade-off between interpretability and optimization.

4.6.3 RQ3: How does the state the art in prediction compare to ICO?

4.6.3.1 SPL Conqueror Technique

In this research question we apply the state of the art performance-influence prediction tool SPL Conqueror. As our training data to SPL Conqueror we use 12 different samples. We use five 2-way and five 3-way covering arrays, a set of random configurations (the same as in RQ2 - 100,000 for BLAST and MFA and 1,000 for SPL Conqueror), and the combination of all distance 0 through 3 configurations used in ICO in RQ1. We choose covering arrays because they performed well with these subjects in previous SPL Conqueror work [32]. We use a simulated annealing tool [46] for the unconstrained models, and for the BLAST subject we used the CASA tool [47]. We use random to see how well SPL Conqueror can perform without constraining the sample set. The distance 0 through 3 (or D0-3) allows use to better compare the SPL Conqueror regression models with ICO by training it on the same data ICO was evaluated on.

SPL Conqueror uses a relative error rate calculation within its algorithm (Equation 4.7). Due to the fact that zero is a valid result for the measured value, SPL Conqueror can not handle those data sets as that would result in a zero in the denominator of the error calculation. SPL Conqueror acknowledges it has difficulty handing values of zero or near to zero. To combat this we perform the four following types of experiments:

1. No change (Reg) — input data as it
2. Absolute (Abs) — we use the absolute error calculation instead of the relative
3. Plus 1 (P1) — we add 1 to every measurement to shift all values to 1 or greater
4. Times 100 Plus 1 (T100P1) — we multiply every measurement by 100 then add one to scale then shift all values

Our motivation for choosing the last transformation was to try to distance outputs of zero from non-zero results by scaling all other outputs before adding 1. For the following results we will only present the best performing experiment. Note that we were not required to perform a transformation

for the SPL Conqueror subjects (the functional output is always larger than 1) so we report only on the *no change* data set. We do report on the results of the other transformations in our Appendix data. Results for all four transformations on each sample can be found in Appendix C.

As we know from RQ1, SPL Conqueror is highly-configurable. However we use the default options (with the exception of modifying the error formula for our transformations). The irony is not lost on us, but the purpose of this RQ is not to get the optimal result out of SPL Conqueror, but to see if its regression equations can provide useful information to the user.

4.6.3.2 What is the accuracy of SPL Conqueror trained on various samples?

For BLAST we translated the two enumeration type configuration options `dust` and `strand` into binary options. We ran five 2-way covering arrays of size 55-62 configurations and five 3-way covering arrays of size 393-411 configurations. For MFA we translated the two enumeration type configuration options `dust` and `strand` into binary options. We ran five 2-way covering arrays of size 37-38 configurations and five 3-way covering arrays of size 260-267 configurations. In SPL Conqueror we translated enumeration type option `scoreMeasure`. The covering arrays ranged from size 12-14 for 2-ways and 48-50 for 3-ways.

We choose the best performing transformation with respect to the relative error and display the results in Table 4.19. We report of the number of configurations in the sample, and on three metrics from the output of SPL Conqueror: the relative error, model complexity, and runtime in seconds. The current version of SPL Conqueror can not accept a separate custom test set to validate data on, thus the relative error reported is based on the training data. As a secondary evaluation we compute the mean absolute error against the set of random configurations to understand the accuracy of the predictors on new data. We report both the raw mean absolute error (MAE) and the percentage (relative to the mean of the true measurements as a normalization). 100% MAE% means the predictor is off on average by the average measured result.[5]

[5]Please see Appendix C for additional metrics.

BLAST and MFA both demonstrated high error rates in all trained regression models. The complexities of the models varied with a high over 68 and a low at 2. A regression model with complexity 2 (having two terms in the regression equation) would be interpretable, however the error is high so the model may not provide accurate information to the user. The error rates on the SPL Conqueror subjects are considerably lower. The model for Dune training on the random configurations has a relative error of 1.152 and a MAE of 0.121 (0.69%). However this model has a complexity of 59 challenging its interpretability. The model trained on the D0-3 data has a complexity of 6 which is much lower, but sacrifices accuracy.

We see there may be a trade-off between complexity (leading to interpretability) and accuracy in the SPL Conqueror regression models. We also see a difference between the performance on the bioinformatics subjects and the software engineering tool signaling there may be domain differences.

4.6.3.3 Can SPL Conqueror identify the effects of configuration options in isolation and in pairs (D1 and D2)?

For each configuration we recorded if the output was greater than (GT), less than (LT), or equal to (EQ) the default. We did this for the actual measurement as well as SPL Conqueror predictions and compare them. We present results of this analysis for the MFA and Dune subjects here, all subjects can be found in Appendix C. We utilize confusion matrices for this evaluation (refer back to Section 4.5.4.2 for review). Results for D1 can be seen in Table 4.20 and D2 in Table 4.28. Each table displays the confusion matrix trained on the random data, the D0-3 data, and on the best performing covering array. Recall an ideal result would be going down the diagonal (highlighted) where the predicted effect matches the actual measured effect.

At an initial glace we see the largest number does appear in the diagonal under all cases. However, the majority of configurations will be equal to the default (resulting in no change and an effect of 0) which is represented by the center column. So we are seeing that the regression models are accurate at identifying no effect in most cases. However, the positive and negative effects (GT, LT) are the more interesting cases to a user as it demonstrates cases where the options have an

effect on the output and is our use case. SPL Conqueror does not perform well in predicting those effects. In fact, the majority of the time SPL Conqueror is predicting no effect (EQ) for any given configuration (the middle row).

To summarize the results of the confusion matrices over all subjects we calculated the percentage of GT, LT, and GT + LT cases that the regression models can identify and report them in Table 4.36. For example relating to Table 4.22 (MFA on the regression model trained on Random data) we obverse that $\frac{0}{5}$ or 0% of the GT effects were correctly identified, $\frac{1}{5}$ or 20% of the LT effects were correctly identified, resulting in a total correct identification of $\frac{0+1}{5+5}$ or 10% of all effects. We see the percentage of effects identified are very low across all subjects. It is useful to see the accuracy over all effects, but due to our use case the GT effects for BLAST and MFA, and the LT effects for SPL Conqueror are the priority to correctly classify. The best BLAST 2-way covering array performs the best identifying about 66% of D1 and D2 GT effects. None of the MFA models can identify any of the GT effects. The best SPL Conqueror was for Hipacc trained on a 3-way covering array identifying 40% of D1 LT effects and 46% of D2 LT effects. The best model for Dune was trained on the D0-3 data. We find there is no consensus among these result on which training data set is most effective for identifying effects.

4.6.3.4 Discussion

SPL Conqueror reports the error and complexity on both the complete regression model, and on the *best candidate* which is the best single term of the entire regression equation. Since our goal is end user interpretability we ask if the simplest of SPL Conqueror's resulting regression equations returns accurate results to the user. This is important to note as one of our evaluations assessing if SPL Conqueror is interpretable is the complexity of the models.

We use the complete regression model for two reasons: (1) the primary output of SPL Conqueror is the full regression equation, and (2) a single term model will likely be unable to handle modeling effects of several options in combination. However, analysis of the best candidate warrants future work.

We also point out several challenges in using SPL Conqueror for our bioinformatics subjects. First is the transformations we had to perform on the data as SPL Conqueror has difficulty handing measured values of zero or near-zero due to numerical issues in calculating the error rate. We note this overhead in translating the data is an added step that had to be performed pre- and post-running SPL Conqueror.

We also had an issue evaluating the regression models of BLAST. SPL Conqueror assumes that numeric values are mandatory. However, our BLAST subject has one numeric configuration option (`min_raw_gapped_score`) that can be set to `NULL` which is its default value. Since `min_raw_gapped_score` showed up as a term in the regression equations we could not evalute its result on any cases where `min_raw_gapped_score` was set to `NULL`.

4.6.3.5 Summary of RQ3

We find SPL Conqueror has high error metrics on BLAST and MFA while SPL Conqueror has lower error metrics. However we find the complexity of the SPL Conqueror models to be very high signaling a possible trade-off between complexity (leading to interpretability) and accuracy in the SPL Conqueror regression models. We also observed a difference between the performance on the bioinformatics subjects and the software engineering tool signaling there may be domain differences.

We find SPL Conqueror can identify cases of configurations having no effect on the functional output (EQ). However, SPL Conqueror is not effective at identify positive or negative effects (GT or LT) in distance 1 and distance 2 configurations.

4.7 Conclusion and Future Work

We developed the ICO framework which has several key features towards interpretable configuration options:

- Providing the user with multiple solutions;
- Provide configuration suggestions to the user in a *readable* and *transparent* format;

- Prioritize configurations by distance from the current configuration; and
- Require a small amount of configurations to evaluate due to its heuristics.

In demonstrating ICO on three software tools and four inputs, we see it can identify the effects of changing configuration options. ICO can help a user identify how many configuration options away from the default they should explore to see improvement in their result. We found that ICO identified the best observed configuration for MFA in 2 steps, and found a configuration in the top 99.77% of solutions in BLAST.

The error rate of SPL Conqueror varies between sample and subject, and due to numerical challenges such as zero and near-zero outputs additional metrics should be explored. We find the interpretability of the raw output of SPL Conqueror to be low in most cases due to the size of the resulting complexity models. In the cases that the complexity is low, the accuracy is too high.

We see room for an extension in the form of an interface on top of SPL Conqueror to perturb the configuration options in the regression equations as we do in ICO to increase the interpretability of its results. In the field of interpretable machine learning a post-hoc explanation can be used increase interpretability of an existing model such as utilizing a simpler learner on top of the predictor in a such as called *mimic learning* or *surrogate intrinsically interpretable model* [108, 111].

In other future work we would like to perform a larger evaluation of ICO since this study used a limited number of subjects and always begins at the default configuration. As we aim to address interpretability in regards to end user use cases a human study would be highly beneficial. The efficiency of ICO also requires additional evaluation. We believe further heuristics will improve implementing ICO in practice. ICO also currently only displays the positively impacting configuration options to the user; however, the same method can be done for the negatively affecting options. We leave as future work to display these negative results in an effort to understand how each options could affect their output. We would also like to extend ICO to fit diverse users goal that could be multi-objective. For example, to identify effects for functional output *and* performance and display these results in a combined format. We would also like to integrating ICO into an existing configurable tools interactive user interface. For example ICO-for-KBase.

Table 4.7: Confusion Matrices

Table 4.9 Traditional confusion matrix

	Actual 0	**Actual 1**
Predicted 0	TN	FN
Predicted 1	FP	TP

Table 4.10 3-class confusion matrix

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	1	0	0
	EQ	1	3	0
	LT	0	1	4

Table 4.11 Optimal 3-class confusion matrix

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	2	0	0
	EQ	0	4	0
	LT	0	0	4

Table 4.12: RQ1 Summary

		BLAST	**MFA**	**SPL Conq. D**	**SPL Conq. H**
D1 + configs		6/76	5/92	7/61	5/61
(% reduction)		(92.02%)	(94.57%)	(88.52%)	(91.80%)
D2 + configs		278/1568	395/3885	348/1668	232/1500
(% reduction)		(82.27%)	(89.83%)	(79.14%)	(85.53%)
D3 + configs		102/111	4/4	80/81	28/29
D1 config. opts.		3/22	2/31	7/26	5/26
(% reduction)		(86.36%)	(93.55%)	(73.08%)	(80.77%)
D2 config. opts.		10/22	3/31	12/26	8/26
(% reduction)		(54.55%)	(90.32%)	(53.85%)	(69.23%)
D1	min	3	0.002	-0.0573	-0.112
range	max	1532	0.012	-1.7466	-1.006
D2	min	2	0.002	-0.0126	-0.112
range	max	3478	33.395	-1.7466	-2.200
D3	min	2	19.997	-0.0016	-0.032
range	max	6407	26.739	-2.0331	-2.230
D2 EQ		235	390	320	217
D2 GT		23	5	23	11
D2 LT		20	0	5	4
D3 EQ		66	0	9	17
D3 NEQ		36	4	71	11

Table 4.13: User displayed results for the ICO framework fro D1, D2, and D3

Table 4.15 D1 and D2 effects on subject BLAST. The functional value is the number of hits. *Effect* is the functional value of the row's configuration minus the functional value of the default. *Add. Eff.* is the type of additive effect.

EQ: The effect is equal to the added effects of the D1 configurations
LT: The effect is less than the added effects of the D1 configurations (new interaction)
GT: The effect is equal to the added effects of the D1 configurations (new interaction)

Starting Configuration						
Configuration				**# Hits**		
Default				4516		
Distance 1						
Config. Opt. 1	**Value 1**			**# Hits**	**Effect**	
ungapped	TRUE			6048	1532	
sum_stats	TRUE			5240	724	
xdrop_gap_final	0, 0.1, 0.5, 10			4519	3	
Distance 2						
Config. Opt. 1	**Value 1**	**Config. Opt. 2**	**Value 2**	**# Hits**	**Effect**	**Add. Eff.**
xdrop_gap_final	0, 0.1, 0.5	xdrop_gap	0, 0.1, 0.5	7994	3478	GT
ungapped	TRUE	sum_stats	TRUE	7980	3464	GT
ungapped	TRUE	xdrop_ungap	0.1, 0.5	7625	3109	GT
.....						
xdrop_gap_final	0, 0.1, 0.5, 10	xdrop_gap	30	4518	2	LT

Table 4.16 D3 effects on subject BLAST. The functional value is the number of hits. *Effect* is the functional value of the row's configuration minus the functional value of the default. *Add. Eff.* is the type of additive effect.

EQ: The effect is equal to the sum of the effects a subset of the configuration
NEQ: The effect is not equal to the sum of the effects of any subset of the configuration (new interaction)

Starting Configuration								
Configuration						**# Hits**		
Default						4516		
Distance 3								
Config. Opt. 1	**Value 1**	**Config. Opt. 2**	**Value 2**	**Config Opt. 3**	**Value 2**	**# Hits**	**Effect**	**Add. Eff.**
sum_stats	TRUE	xdrop_gap	0, 0.1, 0.5	xdrop_gap_final	0, 0.1, 0.5	10923	6407	NEQ
ungapped	TRUE	xdrop_gap	0, 0.1, 0.5	xdrop_gap_final	0, 0.1, 0.5	7994	3478	EQ
sum_stats	TRUE	soft_masking	FALSE	xdrop_gap_final	0, 0.1, 0.5, 10	7277	2761	NEQ
sum_stats	TRUE	ungapped	TRUE	soft_masking	FALSE	6513	1997	EQ
sum_stats	TRUE	xdrop_gap	0, 0.1, 0.5	xdrop_gap_final	10	5378	862	NEQ
.....								
ungapped	TRUE	xdrop_gap	30	xdrop_gap_final	0, 0.1, 0.5, 10	4518	2	EQ

Table 4.17: RQ2 Summary

Values are the best (maximum) functional output observed under all D1, D2, D3, and random configurations tested. The distance was measured by observing all random configurations that result in the best output, and finding the configuration closest to the default (minimal distance away).

Best Output	**BLAST**	**MFA**	**SPL C. Dune**	**SPL C. Hipacc**
Default	4,516	0.674	10.967	10.694
D1	6,048	0.686	9.221	9.688
D2	7,994	34.069	9.221	8.495
D3	10,923	27.413	8.934	8.465
Random	18,766	34.069	5.036	4.782
Distance	5	2	18	17

Table 4.19: SPL Conqueror regression model summaries on different test data sets. Random is 100,000 random configurations for BLAST and MFA and 1,000 for both SPL Conqueror subjects. D0-3 is all configuration distance 0 through 3 which is representative of the data presented by ICO. Five 2-way and five 3-way covering arrays were run, average and standard deviation (in parenthesis) are reported.

BLAST						
Training Data	**Num. of Configs**	**Relative Error**	**Complexity**	**Elapsed Seconds**	**MAE**	**MAE %**
Random	99,999	75.005	31	1632.434	3325.739	100.023%
D0-3	874	99.234	41	9.513	1.94×10^{11}	5.84×10^{9}%
2-way CA	55-62	37.625	68.800	527.352	5.23×10^{6}	1.57×10^{5}%
(STD)		(51.974)	(53.176)	(489.067)	(1.17×10^{7})	(3.51×10^{5})
3-way CA	387-411	76.245	10.800	2.243	1550.107	46.620%
(STD)		(39.733)	(0.837)	(0.068)	(1.686)	(0.051)
MFA						
Random	97,795	1.315	2	52.102	1.315	178.178%
D0-3	1,776	0.251	2	0.584	0.977	132.346 %
2-way CA	37-38	0.003	22	2.194	4.213	570.625%
(STD)		(0.006)	(13.711)	(3.695)	(4.733)	(641.089)
3-way CA	260-267	2.950	7.2	0.669	2.377	321.962%
(STD)		(2.231)	(5.541)	(0.542)	(1.648)	(223.198)
SPL Conqueror - Dune						
Random	984	1.152	59	239.348	0.121	0.690%
D0-3	668	4.174	6	1.881	2.944	16.824%
2-way CAs	13-14	0.509	11.4	0.626	5.839	33.365%
(STD)		(0.307)	(3.050)	(0.522)	(3.140)	(17.943)
3-way CAs	53-54	3.025	32	10.340	2.744	15.680%
(STD)		(2.398)	(7.746)	(7.099)	(0.929)	(5.307)
SPL Conqueror - Hipacc						
Random	991	4.260	46	168.855	0.705	3.138%
D0-3	548	1.911	11	13.561	4.547	20.241%
2-way CAs	12-14	0.584	14.8	1.348	9.978	44.417%
(STD)		(0.432)	(4.438)	(0.838)	(5.209)	(23.188)
3-way CAs	48-50	2.230	46.4	37.656	2.959	13.157%
(STD)		(0.865)	(11.415)	(21.098)	(1.597)	(7.109)

Table 4.20: Confusion Matrices on Distance 1 configurations. Results displayed evaluated on random, D0-3 (distance 0 through 3), best 2-way covering array, and best 3-way covering array.

Table 4.22 MFA - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	5	80	4
	LT	0	0	1

Table 4.23 MFA - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	5	80	4
	LT	0	0	1

Table 4.24 MFA - 3CA5

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	5	80	3
	LT	0	0	2

Table 4.25 Dune - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	2	0
	EQ	2	44	7
	LT	2	3	0

Table 4.26 Dune - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	3	0	0
	EQ	1	49	5
	LT	0	0	2

Table 4.27 Dune - 3CA4

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	2	4	0
	EQ	1	44	6
	LT	1	1	1

Table 4.28: Confusion Matrices on Distance 2 configurations. Results displayed evaluated on random, D0-3 (distance 0 through 3), best 2-way covering array, and best 3-way covering array.

Table 4.30 MFA - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	394	2987	336
	LT	0	0	88

Table 4.31 MFA - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	394	2987	336
	LT	0	0	88

Table 4.32 MFA - 3CA5

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	394	2987	249
	LT	0	0	175

Table 4.33 Dune - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	6	89	14
	EQ	98	882	313
	LT	116	129	21

Table 4.34 Dune - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	166	0	0
	EQ	52	1100	242
	LT	2	0	106

Table 4.35 Dune - 3CA4

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	112	170	28
	EQ	49	882	264
	LT	59	48	56

Table 4.36: RQ3 Summary

Values are the best (maximum) functional output observered under all D1, D2, D3, and random configurations tested. The distance was measured by observering all random configurations that result in the best output, and finding the configuration closest to the default (minimal distance away).

		Distance 1			Distance 2		
Subject	**Model**	**GT %**	**LT %**	**GT + LT %**	**pos. %**	**neg. %**	**all effects %**
BLAST	Random	0	0	0	2.16	0	1.13
	D0-3	0	0	0	2.52	0	1.31
	2CAs	66.67	40	63.63	69.78	47.45	59.1
	3CAs	0	40	18.18	0	42.75	20.45
MFA	Random	0	20	10	0	20.75	10.76
	D0-3	0	20	10	0	20.75	10.76
	2CAs	0	0	0	0	0.94	0.49
	3CAs	0	40	20	0	70.28	21.39
SPLC-D	Random	0	0	0	2.73	6.03	4.76
	D0-3	75	28.57	45.45	75.46	30.46	47.89
	2CAs	0	0	0	1.82	6.03	4.4
	3CAs	50	14.29	27.27	50.91	16.09	29.58
SPLC-H	Random	66.67	20	45.45	69.31	20.26	48.04
	D0-3	66.67	40	54.55	69.31	41.38	57.2
	2CAs	33.33	0	18.18	38.94	8.62	25.79
	3CAs	50	40	45.45	52.48	46.12	49.72

CHAPTER 5. ORGANIC SOFTWARE PRODUCT LINES

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 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 chapter we perform an empirical investigation of domain engineering that leverages an open-source repository of more than 45,000 reusable DNA parts. We show that we can identify 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. We demonstrate how we can use the feature models to help guide new experimentation, and end with an analysis of reverse engineering on both complete and incomplete sets of products. In the process of this study, we have uncovered limitations of existing SPL techniques and tools which provide a roadmap for making SPL engineering applicable to new and emerging domains.

5.1 Introduction

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

Part of the material in this chapter was published in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* in September 2019 [125].

directly to open-source systems by defining open-source product lines [64], and to other emerging domains such as cyber-physical systems [62] and the Internet of Things (IoT) [126–128]. Recently, Montalvillo and Díaz have proposed techniques to aid SPL practices within GitHub [65].

In this chapter, 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 [12]. It is being used for sensing heavy metals for pollution mitigation [81], development of synthetic biofuels [82], engineering cells to communicate and produce bodily tissues [83], emerging medical applications [84, 85], and basic computational purposes [86]. 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 [95]. 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) [28] contains over 45,000 DNA parts and can be viewed as a Git repository for DNA. In this work 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.

We explore the idea of using 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). We conclude that OSPLs do exist, and that we can leverage techniques from product line engineering to potentially help in the synthetic biology development process. We have also identified several limitations of current SPL engineering techniques, which we argue may be applicable to other emerging SPL domains as well. This includes the need to better support (1) SPL evolution; (2) duplicate features; (3) scalability of reverse engineering; and (4) a need for interactive feature modeling from incomplete sets of products.

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 limitations of existing techniques for organic software product lines and its implications for the software product line engineering community.

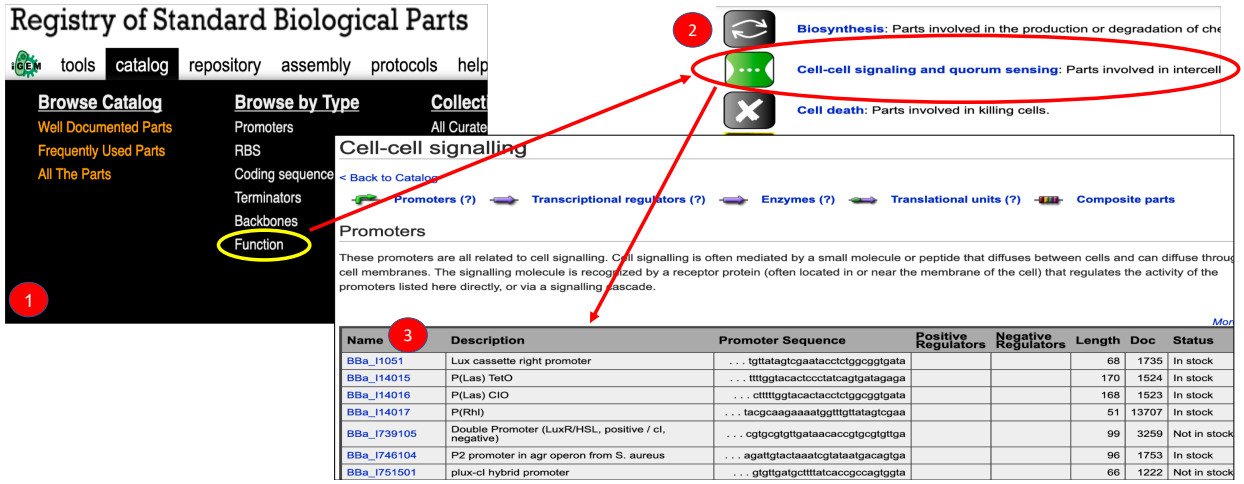

Figure 5.1: Browse by function → Cell-to-cell signaling

In the next section we present a motivating example. We then propose the notion of an OSPL (Section 5.4). We follow this with our empirical study (Section 5.5), threats to validity (Section 5.6), results (Section 5.7), and discussion (Section 5.8). We present related work in Section 5.3. We end with conclusions and future work.

5.2 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 some chemical *reporter*. Suppose an engineer wants to build a cell-to-cell signaling system from scratch. If the engineer has no additional resources other than an online repository and 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 5.1, steps #1 and #2, show this in the BioBrick repository. There are 10 functions listed including "Cell-to-cell signaling and quorum sensing." Parts are sorted (Figure 5.1 - 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) [129]. Looking on this team's web page would lead the user to find models for 30 composite parts for cell-to-cell signaling. While this is an improvement over the prior approaches and provides a roadmap to build the system (along with results of the study), it is limited to the 30 products that the ASU team chose to use in their experiments. As we will show, there are many more ways to build a cell-to-cell signaling system.

What if, instead of starting from scratch, the synthetic biologist begins this process with the feature model shown in Figure 5.2 (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.

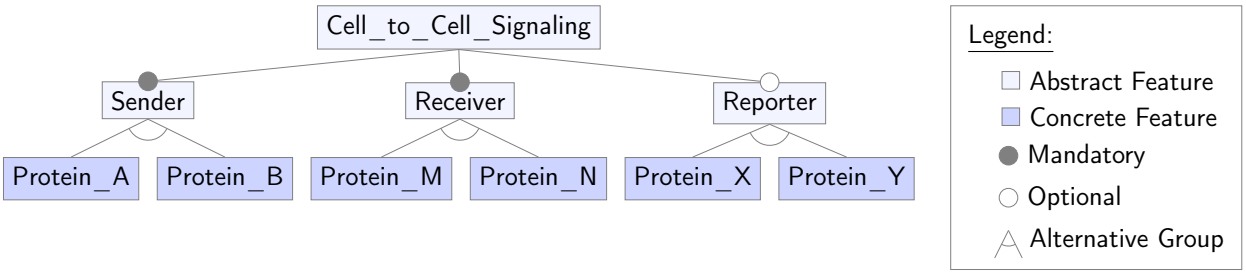

Figure 5.2: Example feature model for a cell-to-cell signaling system.

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

As further motivation, if we return to the ASU team's experiments, one of their goals was to investigate the 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 empirical study. And it is this motivation that leads us to propose organic software product lines.

5.3 Related Work

The most similar work to ours is that of [63], who study a family of DNA nanodevices. While they also look at DNA, they study chemical reaction networks (CRNs) instead of living synthetically

engineered organisms. CRNs are not necessarily part of living organisms, because they do not involve transcription or translation of DNA code [130]. Their line of organic programming leverages CRNs, which are sets of concurrent equations that can be compiled into single strands of DNA [67], and CRNs have been shown to be Turing complete [131].

Last, ours is not the first analysis of the BioBricks repository. Valverde et al. [132] 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.

5.4 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 SPL 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 [61, 64]. This has led to a broader view of SPLs. We ask the same question now of organic programs. Is there a mapping between traditional software product line engineering and synthetic biology that allows for managed development and reuse?

As Clements and Northrop state, the output of domain engineering should contain (1) a product line scope, (2) a set of core assets, and (3) a production plan [57]. 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.

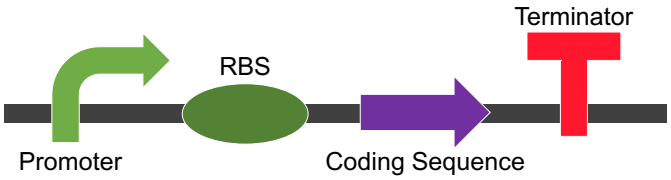

Figure 5.3: SBOL model of a transcription unit. This composite part is composed of four basic DNA parts: The promoter (also called a regulator), ribosome binding site (RBS), coding sequence, and terminator.

5.4.1 Assets

To begin, we need to identify what constitutes a core asset for this domain. Assets in traditional product lines can include a software architecture, reusable software components, performance models, test plans and test cases, as well as other design documents. In organic programs, we see similar elements.

First, 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 Figure 5.3. 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 [133] and higher level system architecture such as GenoCAD [88].

5.4.2 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 variable features and their relationships. An example of commonality is the transcription unit (Figure 5.3). The specific choices for promoters and binding sites defines the variability. When inserted into their host organisms at specific binding sites, the sets of DNA sequences define unique sets of *products*. Last, a production plan can be created in combination with a feature model and constraints. The feature model and constraints show how the DNA parts, as *features*, form a family of products, along with experimental notes on expected environmental conditions or other assumptions that are required for the program to run correctly.

5.4.3 Application Engineering

In this context, application engineering involves combining the expected parts using standard DNA cloning techniques for insertion into a living organism [134]. 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.

In the following sections we empirically evaluate our ability to apply these concepts to an existing DNA repository and a set of commonly engineered biological functions.

5.5 Empirical Study

We conduct an empirical study to evaluate the feasibility of using product line engineering in synthetic biology. Supplemental data for this study can be found on our website.[1]

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

Our overall goal is to ascertain the feasibility and benefits of SPL engineering when applied to organic programs. To evaluate our conjecture we ask the following four research questions:

RQ1: Does a DNA repository have functions with the characteristics of a software product line? This is a necessary requirement for our overall conjecture to hold.

RQ2: Can we build feature models representing families of products from an existing DNA Repository? If the repository has sets of products with SPL characteristics, then we would expect to be able to build feature models for these.

RQ3: What type of end-to-end analysis can we provide to developers of organic programs? Once we have feature models, can we leverage the feature models to answer questions which can benefit reuse and improve the quality of the SPLs? These are important aspects of application engineering.

RQ4: How effective is automatically reverse engineering feature models in this domain? Just as we find in many open source systems, feature models are often lacking. We ask whether we can utilize existing reverse engineering tools to approximate feature models as starting point for SPL engineering.

5.5.1 Subject Repository

We use as our subject repository the *Registry of Standard Biological Parts* (the BioBrick repository) as described in Section 2.4 [28]. We choose this subject as it is the largest open-source DNA repository and continues to grow (on average 2,995 parts are added each year).

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

5.5.2 Study Objects

For RQ1 we use the BioBrick repository. For RQ2 we select four biological functions in which to build a feature model (described in more detail below). 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. For RQ3 we selected a 2017 iGEM team from Arizona State University [129] whose project is a subcategory of cell-to-cell signaling. For RQ4 we use the feature models and sets of products from RQ2 to automatically reverse engineer.

5.5.2.1 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 [135].

5.5.2.2 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 [136] and pH-sensitive kill switches [137]. For a good survey of kill switches used in the iGEM competition, see [138].

5.5.2.3 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 [139]. 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 [140]. *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 [139, 141]. AAV can be customized for the desired gene to be transported into the cell and is the subject of our feature models.

5.5.3 Methodology

Next we describe the methodology for each of our research questions.

5.5.3.1 RQ1 - Characteristics of a Software Product Line

To study the core assets of the system, we use the BioBrick API to pull data for all parts up through December of 2018, consisting of 47,934 entries [142]. 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, RBS, 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 evaluate 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 researchers independently identify whether they contained each of these assets. We use two researchers 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 researchers next compare their responses, and any discrepancies are resolved between them, or in discussion with a third researcher. 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.

5.5.3.2 RQ2 - Manual Feature Models

For RQ2 we manually build feature models for the four different biological functions: cell-to-cell signaling, kill switch, and viral vectors.

Cell-to-cell signaling. 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 domain expert.

Kill Switch. 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 [133]. 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 domain expert.

Viral Vectors. 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 [143]. 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 domain expert.

5.5.3.3 RQ3 - End-to-end Case Study

In RQ3 we study the 2017 iGEM team from Arizona State University (ASU). We begin by referring to the team's project web page. 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 [144].

In this RQ we perform a slice of the cell-to-cell signaling model. To do this we use the slicing functionality in FeatureIDE [145]. We also generate samples of the ASU model and the cell-to-cell signaling models using combinatorial interaction testing. For this we use CASA [146].

5.5.3.4 RQ4 - Automated Reverse Engineering

We utilize an existing reverse engineering tool, SPLRevO developed by [147, 148]. We note that other similar tools could also be used. This tool accepts either (1) a set of constraints based on domain knowledge describing the compatibility of the DNA parts, or (2) a set of products which can be the known working composite components. The tool then uses a genetic algorithm to automatically build a feature model that represents all products. The fitness function (validity) aims to maximize the coverage of the set of desired products while minimizing any undesired (additional) products using a penalty.

We reverse engineer four models: the ASU team project, kill switch, viral vector GOI, and viral vector capsid. For the kill switch and viral vector models we use 200 generations and 100 runs. 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). We do not have a set of constraints for the models, so we use a set of products as inputs.

We perform two types of reverse engineering: (1) from a known oracle, and (2) from an incomplete set of products. The first method can apply when there is a domain expert who can generate a set of products representing the entire product line to use as a predefined oracle. The second method applies when a complete set of products is not possible to ascertain. In that case the user builds the model using any available resources. Note this is likely to create an *incomplete feature model* meaning not all products in the product line will be represented. This initial incomplete feature model serves as a starting point for engineers and can be refined over time and in collaboration with other engineers.

For the first method, we take the ASU and kill switch feature models from RQ3 and RQ2 respectively and generate the set of products. We automatically generate the products using FAMA, a framework for the automated analysis of feature models [149]. We use those products as our known oracle and serve as the inputs to SPLRevO.

For our second method, reverse engineering from an incomplete set of products, we use the viral vector models. First we split the list of 102 parts from the repository into the 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 5.5.3.1) 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 5.7.4.4. 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 22 products. The full set of products including the basic parts can be found on our supplementary website.

We choose one reverse engineered model from each subject to visually display in the results. We choose the model that obtained the highest validity (defined next) for each subject. 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.

Metrics. To evaluate 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 *input_products* and the products represented by the reverse engineered model as the *output_products*.

The precision is the ratio of matched products to all output products (Equation 5.1). *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 (Equation 5.2). Recall is a real value ranging from 0 to 100 where 100.0% recall occurs when the model covers all input products. The F-measure is the weighted harmonic mean of precision and recall (Equation 5.3). Intuitively, we need the model that covers the most input products but also has less additional products. Thianniwet and Cohen [147] show that the validity fitness function in SPLRevO performs the best to reverse engineering feature models. The validity fitness function is designed to guide the genetic algorithm to search for a model and converge towards covering all of the input products when possible, and gradually refine towards models that

contain fewer additional products. Thus, in this work, we will mainly focus on the validity fitness value as the overall model quality - 100% validity represents the optimal model (covers all input products, no more and no less).

$$Precision = \frac{input\_products \cap output\_products}{output\_products} \tag{5.1}$$

$$Recall = \frac{input\_products \cap output\_products}{input\_products} \tag{5.2}$$

$$F\text{-}Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5.3}$$

To compute runtime we use *long java.lang.management.ThreadMXBean.get CurrentThreadCpuTime()* to capture the total CPU time for the current thread in nanoseconds (beginning to end).

5.6 Threats to Validity

We present the threats to validity of our study here, along with our attempts to mitigate them.

5.6.1 External Validity

We acknowledge there is only one DNA repository in this study, which means our results may not generalize to all open source DNA systems. However, we chose the largest open source repository for our study, the BioBrick repository. We select four widely used and different biological functions to build feature models to ensure that our results are as broad as possible. Other functions may result in different models, however 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.

5.6.2 Internal Validity

We acknowledge we built the feature models, however for each subject, we used extensive documentation and asked external domain experts for help validating the models. When we had to make subjective decisions (e.g. in RQ1 in determining presence of assets), we used two separate researchers to independently perform an analysis and then discussed any differences to reach a resolution. We also acknowledge there may be some errors in our analysis tools, but for each of our research questions we attempted to reduce these threats by manually evaluating subsets of the data. We also have provided our data/results/models on an external website allowing others to recreate the results of our study.

5.6.3 Construct Validity

We also acknowledge that we might have chosen different metrics to answer each of our questions. When possible, we use commonly used metrics such as F-measure, recall, and precision. When this was not possible, we clearly describe the metrics chosen and provide data on our website for others to also evaluate.

5.7 Results

In this section we answer each of our research questions in turn. In RQ1 we quantify assets from the BioBrick repository and explore both commonality and variability between products. We focus on domain engineering in RQ2 and application engineering in RQ3. In RQ4 we investigate the use of automated reverse engineering.

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

In Section 5.4 we defined organic software product lines in terms of SPL engineering concepts. We start with the core assets, the code. There are 47,934 BioBrick parts at the time of this publication. Table 5.1 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.

Table 5.1: 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 next analyze 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 5.2 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. 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 [150].

Table 5.2: BioBrick use counts - # user requests.

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

Next we evaluate additional assets described in Section 5.5.3.1: 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 5.3 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.

Table 5.3: 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%

We also analyze the parts added to the repository over time to understand the degree of evolution. 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 Figure 5.4. 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 5.5 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, 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. In Section 5.8 we discuss how this may lead to the need for tool support for maintenance of evolving feature models.

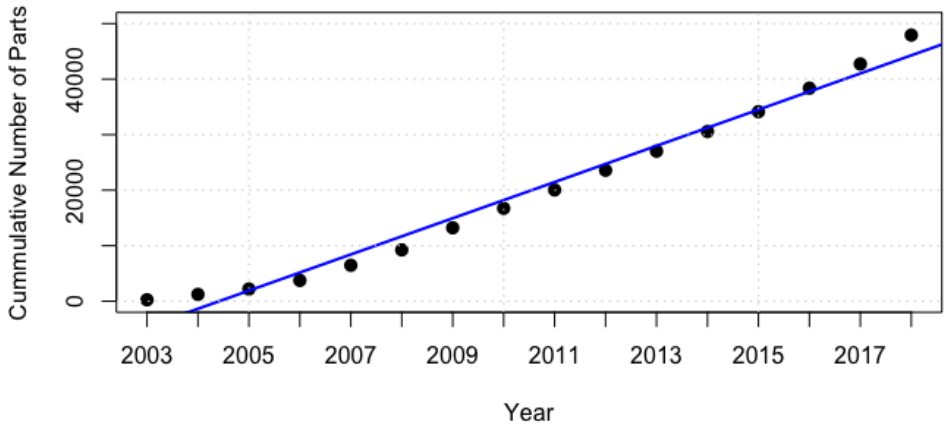

Figure 5.4: Cumulative number of parts over time fitted with a linear trend line (R^2 of 0.9813).

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

5.7.1.1 Variability

To examine variability we focus on the transcription unit, the most basic function (see Figure 5.3). There are 4,165 promoters, 769 RBSs, 10,265 coding sequences, and 518 terminators. If we underestimate the possible product space by counting one of each part (a standard practice is to use two terminators which will increase the space by a large factor) we have on the order of 1.7×10^{13} (17 trillion) products representing transcription units.

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

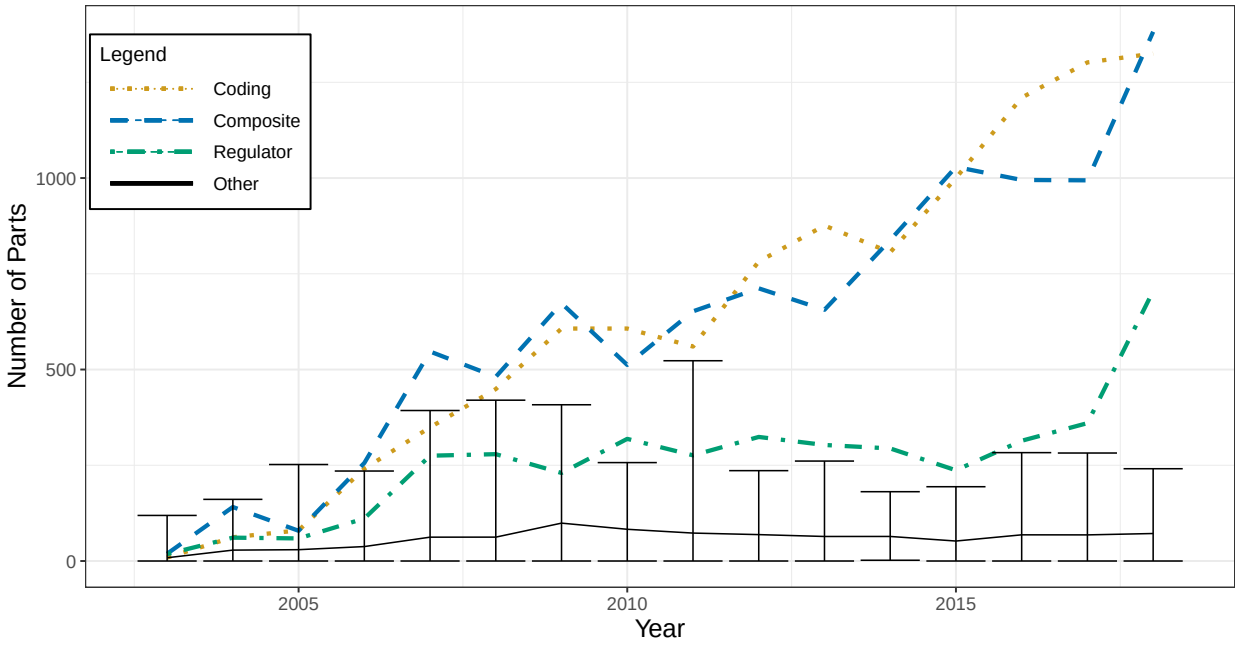

Figure 5.5: Number of parts by type added to the BioBrick repository each year. All other part types were averaged into the "Other" category. The error bars represent the minimum and maximum for each year in all other part type categories.

5.7.1.2 Commonality

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

5.7.1.3 Summary of RQ1.

We found the BioBrick repository does have the characteristics needed to build a software product line. The repository contains 47,934 core assets and is increasing 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.

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

For this research question we build four feature models for distinct functions in the BioBricks repository.

5.7.2.1 Cell-to-Cell Signaling

Figure 5.6 shows the overall cell-to-cell signaling feature model we constructed. Recall that a basic cell-to-cell signaling system has three different transcriptional units (sender, receiver, reporter). The feature model follows a hierarchical model with variation points at the transcriptional unit level as a sub-feature model. Because the full cell-to-cell signaling model is too large to show here, we visually present only some of these sub-feature models and describe the rest in text (the complete model is on our supplementary website). Note that any numerals appearing below features indicate the number of obfuscated features which will appear if those nodes are expanded.

Our model is too large for FeatureIDE to calculate the set of products, therefore we use a dedicated SPL analysis tool, FAMA [149] for this purpose. As seen in Table 5.4 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

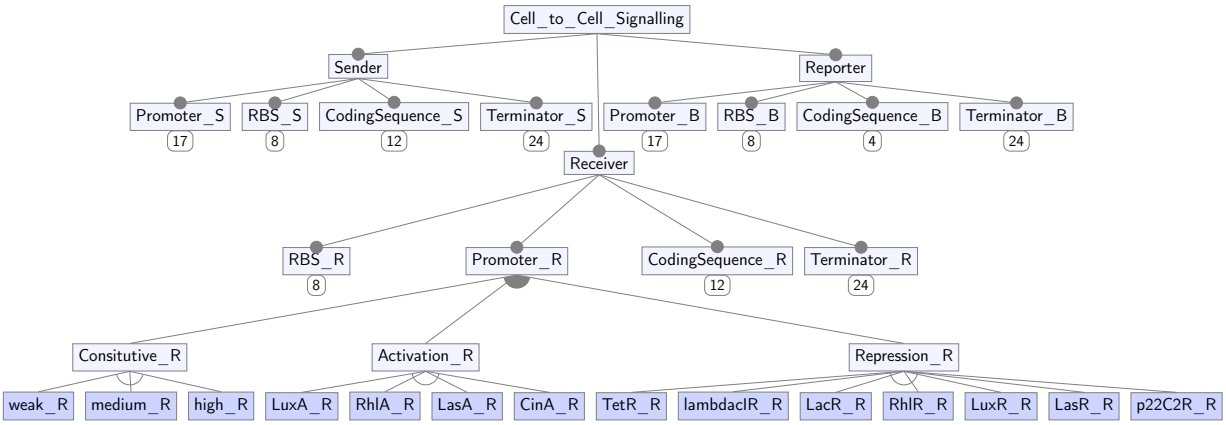

Figure 5.6: Top levels of the cell-to-cell signaling feature model.

Table 5.4: Summary of Feature models

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

by any protein). We found parts that categorize this as *weak*, *medium*, or *strong*. A promoter can also be activated (increased expression) by a protein. We identified four proteins listed under cell-to-cell signaling promoters. We 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 Figure 5.6.

RBS: The next high-level feature is the RBS. This part will have the same variation in the sender, receiver, and reporter. Since the cell-to-cell signaling catalog does not include RBS parts, we looked at all RBS parts 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 Figure 5.7.

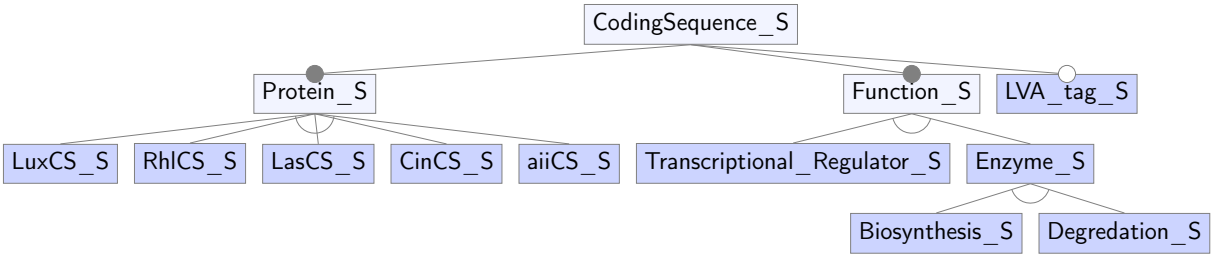

Figure 5.7: Sub-feature model of a protein coding sequence.

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

Terminator: The final part is the terminator. There are no terminators listed in the cell-to-cell signaling catalog, so we look at all terminators in the repository. Terminators can be in the forward direction, reverse direction, or 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.

5.7.2.2 Kill Switch

Based on manual review of 2017 iGEM teams who earned a gold medal, we build the feature model shown in Figure 5.8. 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

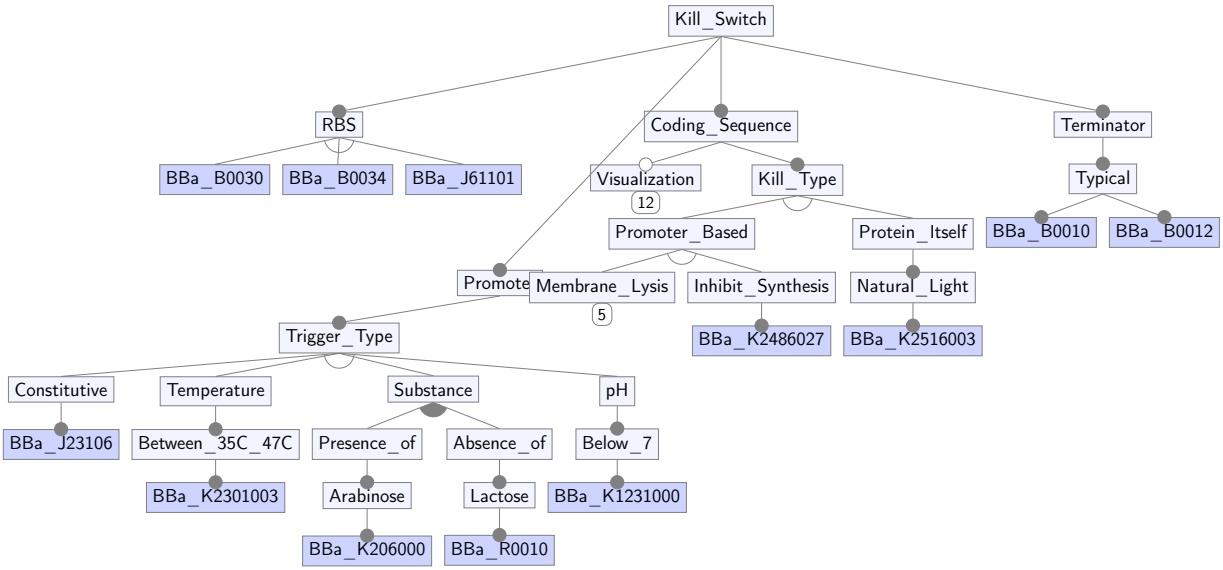

Figure 5.8: A feature model for a kill switch. A numeral on a leaf represents how many nodes are in their subtrees (obfuscated).

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 5.4 this model has 23 features and represents 882 kill switches.

5.7.2.3 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 separately. Figure 5.9 has the feature model for the gene of interest (GOI) and Figure 5.10 has the feature model for the capsid.

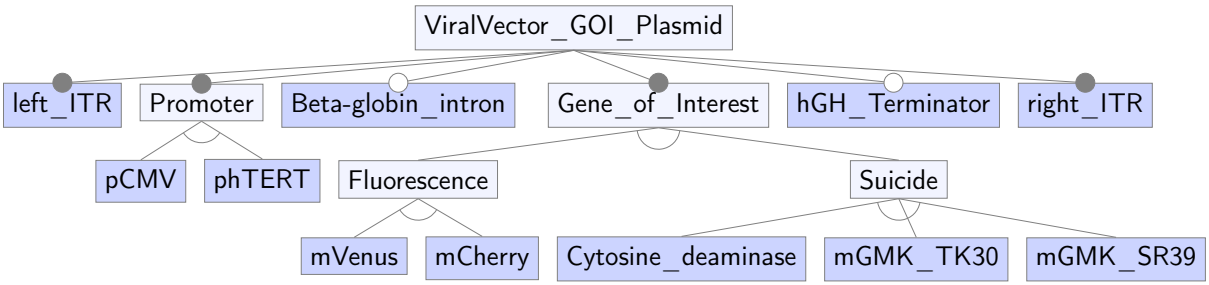

Figure 5.9: Viral Vector Gene of Interest (GOI) feature model.

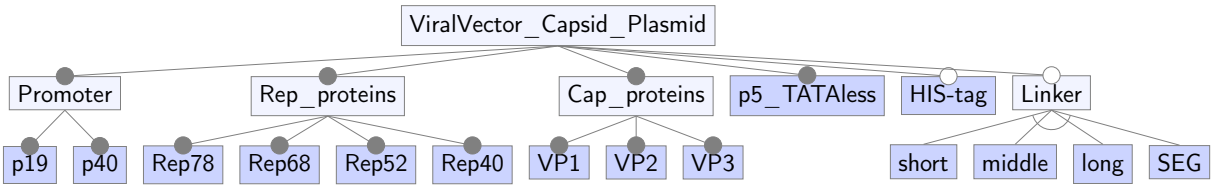

Figure 5.10: Viral Vector Capsid feature model.

The GOI model has 11 features and represents 40 products (Table 5.4). 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 5.4). 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.

5.7.2.4 Summary of RQ2.

We successfully built four feature models for distinct biological functions with variable and common features. Table 5.4 show 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.

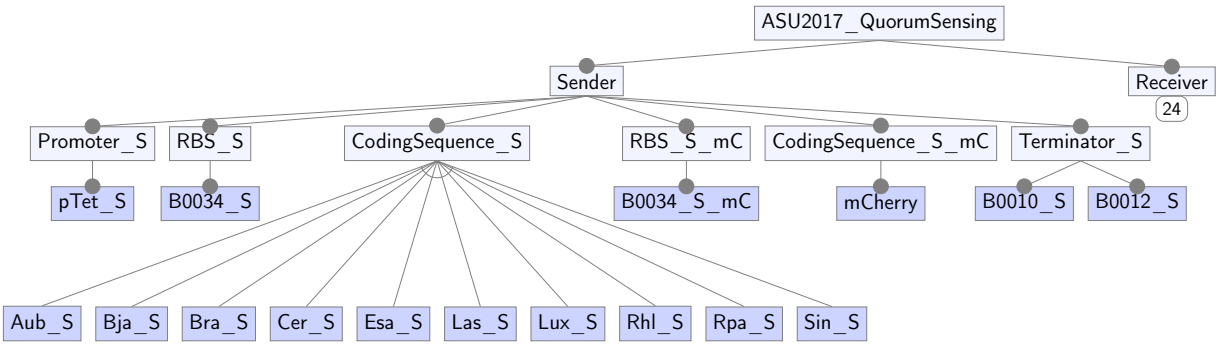

Figure 5.11: The sender slice from a feature model for the 2017 Arizona State University's iGEM project (ASU Model).

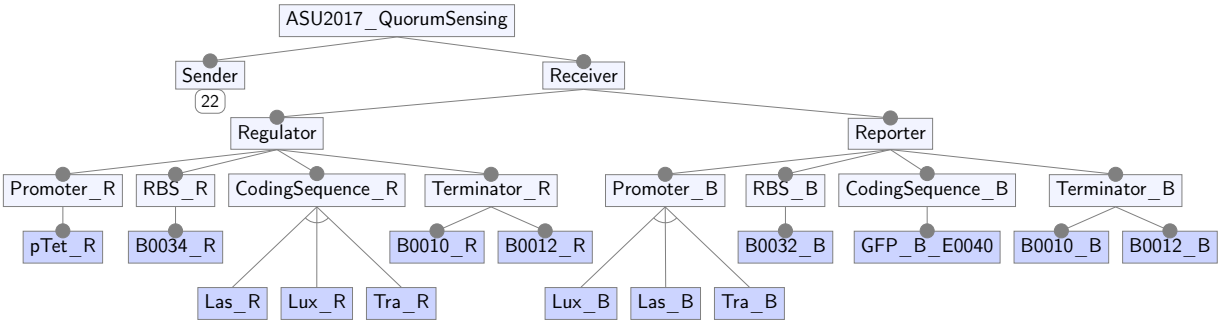

Figure 5.12: The receiver slice from a feature model for the 2017 Arizona State University's iGEM project (ASU Model).

5.7.3 RQ3: What type of end-to-end analysis can we provide to developers of organic programs?

For this research question we ask questions related to application engineering. We use the 2017 iGEM team project from Arizona State University [129, 144] introduced in our motivating example. Because this team built a quorum sensing network, which is a type of cell-to-cell signaling, it helps us validate the results from RQ1 from an application engineering perspective.

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 [129]:

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

The team chose ten different proteins for the sender (Aub, Bja, Bra, Cer, Esa, Las, Lux, Rhl, Rpa, Sin), three different proteins for the receiver (Las, Lux, Tra), and three different proteins for the reporter (Las, Lux, Tra).

5.7.3.1 Providing a Broader View of the Product Space

Though perhaps unintentionally, the ASU team's project represents a feature model. We formalize it by manually reverse-engineering their project feature model from their web page. Figures 5.11 and 5.12 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 experiment model*.

If we compare this ASU model to the reverse engineered model presented in Section 5.7.2.1 (we call this the *cell-to-cell signaling model*), the ASU model is not a direct subset of the cell-to-cell signaling model. In practice it should be. We would have expected the products in the models to overlap, as seen in Figure . However the actual overlap can be seen in Figure . 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.

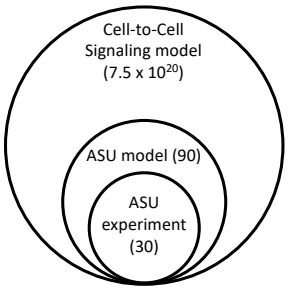

Figure 5.14 Expected overlap

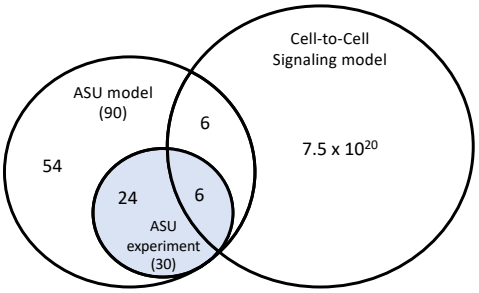

Figure 5.15 Actual overlap

Figure 5.16: The overlap of the feature models and ASU experimentation. The sum of each circle is in parentheses.

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 [151] to allow

users to merge domain knowledge and reconcile multiple models. We further discuss this in Section 5.8.

5.7.3.2 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 Figure 5.17 (called *protein slice*) which 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).

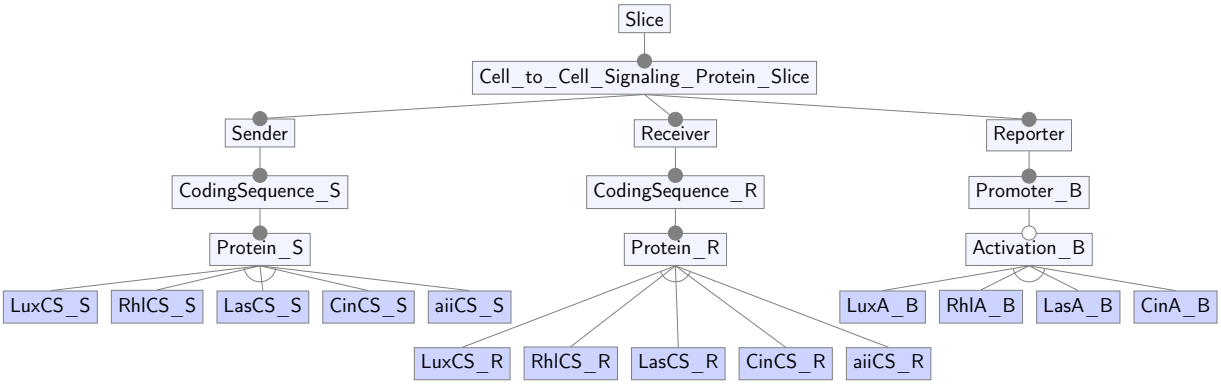

Figure 5.17: A slice of the complete cell-to-cell signaling feature model focuses on protein combinations to mimic the ASU team's experiment (protein slice).

We could also employ common sampling methods such as combinatorial interaction testing (CIT) which samples broadly across a set of features [43]. Using sampling allows us to test a larger space of combinations. We used CASA [146] 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 Figure 5.18. 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.

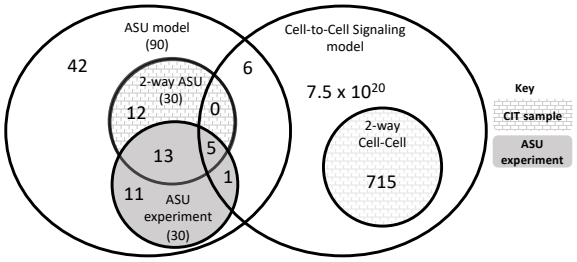

Figure 5.18: Product overlap between each of the feature models and possible samples.

5.7.3.3 Summary of RQ3.

We showed we can provide end-to-end analysis for domain engineers by building a feature model representing a real team's project and demonstrating the use of different analysis methods. In comparing this model to the full cell-to-cell signaling model from RQ2 we see there is discrepancy in domain knowledge between the entire repository and the iGEM team. This signals the potential use 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.

5.7.4 RQ4: How effective is automatically reverse engineering feature models in this domain?

Projects that utilize SPL engineering often do not begin with an existing feature model. An initial feature model can either be manually developed (which may require extensive domain knowledge), or they can be automatically created using existing tooling. Furthermore, given that synthetic biology is an end-user software engineering domain, developers of organic programs may not be well versed in software development methods. Therefore, building a feature model from scratch may be a roadblock. We have observed this in other recent studies that have applied software engineering concepts to synthetic biology [133]. Thus, we want to understand whether it would be possible to use automated reverse engineering tools to obtain an initial feature model.

We reverse engineer four models: *ASU experimental*, *kill switch*, *viral vector GOI*, and *viral vector capsid*. We perform two types of reverse engineering: (1) from a known oracle, and (2) from an incomplete set of products. Recall from Section 5.5.3.4 that the first use case can apply when there is a domain expert who can generate a set of products representing the entire product line to use as a predefined oracle. The second case applies when a complete set of products is not possible to ascertain. In that case the user builds the model using any available resources. Note this is likely to create an *incomplete feature model*. This initial incomplete feature model serves as a starting point for engineers and can be refined over time and in collaboration with other engineers.

5.7.4.1 Reverse engineering from a known oracle

To use as our known oracle, we take the ASU and kill switch models feature models from RQ3 and RQ2 respectively, generate the set of products using FAMA [149], and use those products as inputs to SPLRevO [147, 148].

ASU Experimental: 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

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

Table 5.5: Average and standard deviations over all runs of reverse engineering using SPLRevO on all four subject models. The top two subjects were reverse engineered with a known oracle. The bottom two subjects were reverse engineered from an incomplete set of products.

Model	**Runs**	**Features**	**Products**		**Validity %**		**Runtime (min)**	
			AVG	**SD**	**AVG**	**SD**	**AVG**	**SD**
ASU	40	26	49.58	24.86	94.37	5.34	17.15 hrs	3.46 hrs
KillSwitch	100	16	114.48	36.57	98.16	3.50	18.53	2.70
GOI	100	11	19.31	11.04	90.44	2.86	1.06	0.29
Capsid	100	18	113.18	51.47	83.34	3.18	15.84	3.89

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. Table 5.5 contains a summary of results. We see 17 of the 40 models have 100% validity and the average number of output products was 49.58 with a standard deviation of 24.86. 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.

Out of the 17 models that have 100% validity, 10 have the fewest number of cross-tree constraints (1), a randomly chosen representative can be seen in Figure 5.19. 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.

Kill Switch Model: 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. Typically, an iGEM team will add a feature to a kill switch that causes the host bacteria to fluoresce visibly to the naked eye as a simple way to gauge the effectiveness of the kill

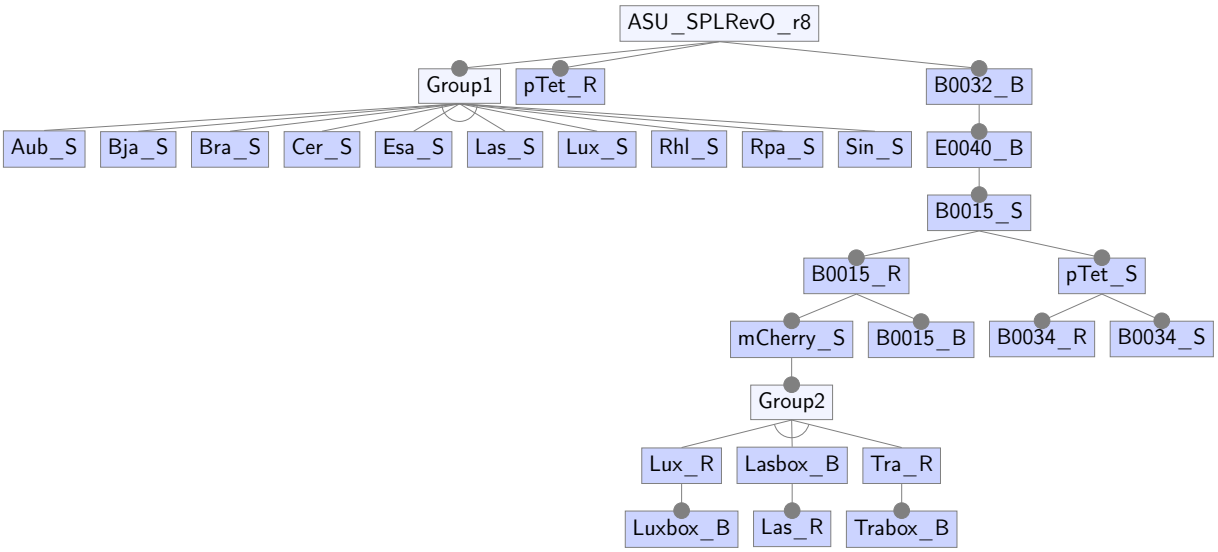

Figure 5.19: ASU reverse-engineered feature model using SPLRevO. We note that the one cross-tree constraint created two false optional features so the displayed feature model is simplified.

switch. That is, if the kill switch is triggered, the host will also produce some fluorescent protein before the membrane is lysed. That reduced the model to 12 features and 105 products to use as inputs to SPLRevO. We ran 100 runs of SPLRevO at 200 generations. As seen in Table 5.5, the average number of output products was 114.48 with a standard deviation of 36.57. 73 of the runs finished with 100% validity, 56 of those had zero cross-tree constraints. We randomly chose one to display in Figure 5.20. If we compare this model to the manually reverse engineered model from RQ2, 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 Figure 5.8.

5.7.4.2 Reverse engineering from an incomplete set of products

We now examine what happens when we do not have a complete set of products for an SPL. We ask whether reverse engineering can provide an initial abstraction that can be used to develop a more complete feature model.

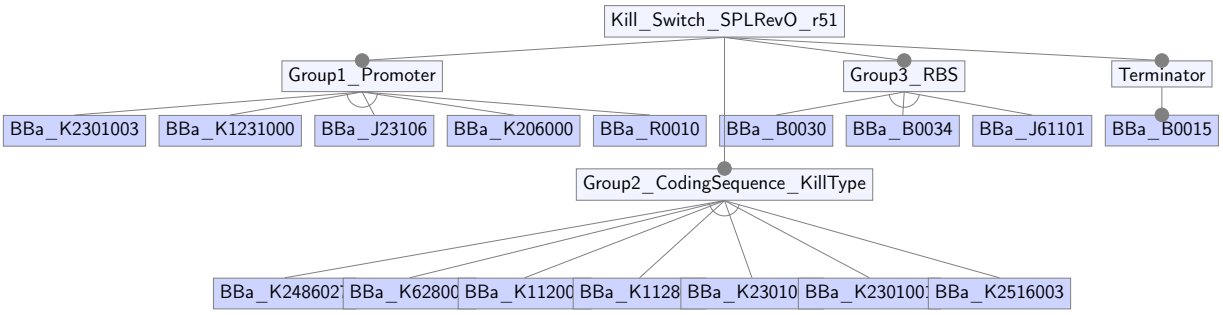

Figure 5.20: Kill Switch reverse-engineered feature model using SPLRevO.

As seen in Table 5.5, the GOI models had on average 19.31 output products with a standard deviation of 11.04. We can immediately notice this is more than the 11 products we used as input. Out of the 100 runs of the GOI model, one has 100% validity representing 11 products which we present in Figure 5.21. 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 capsid model had on average 113.18 output products (Table 5.5). Again this is more than the 18 products used as input. The best model out of 100 runs performed with 88.32% validity representing 60 products and can be seen in Figure 5.22. 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. We discuss these shortly.

5.7.4.3 Runtime

A summary of the runtimes can be seen in Table 5.5. The ASU model took the longest at an average of 17.1 hours. This model took longer due to having more features and thus a larger search

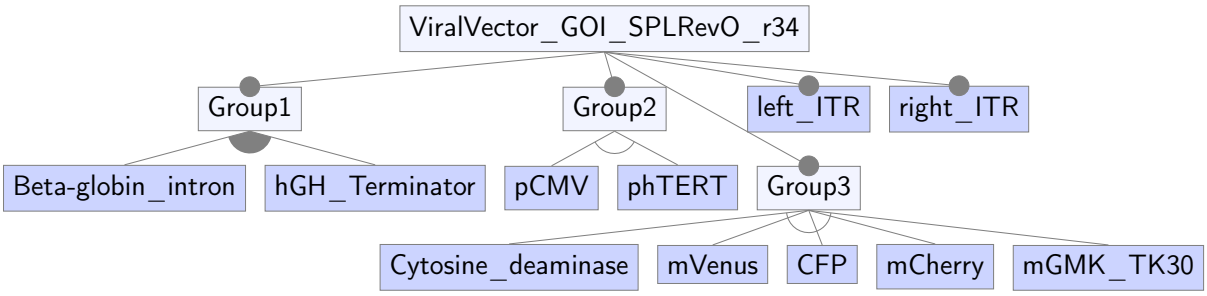

Figure 5.21: GOI reverse engineered feature model using SPLRevO.

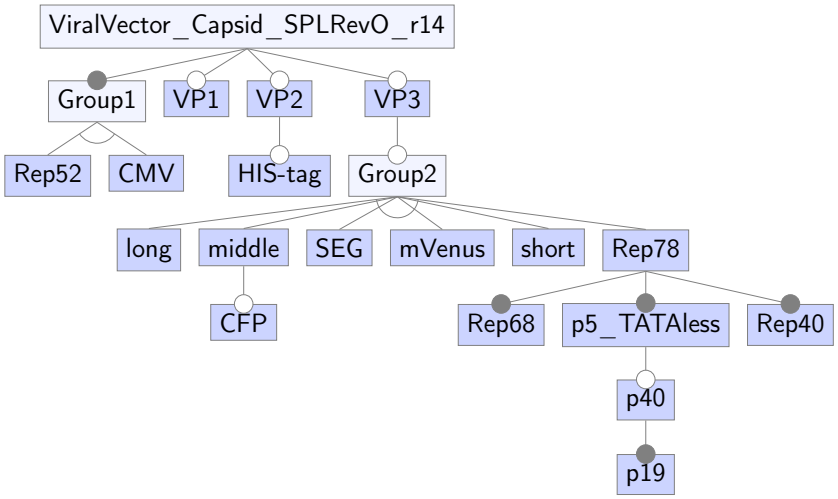

Figure 5.22: Capsid reverse engineered feature model using SPLRevO.

space, but achieved an average of 94.37% validity within 400 generations. The other three models took less than 20 minutes, with the GOI model needing just over one minute. We see that the runtime grows exponentially relative to the number of features (R^2 of 0.9765).

5.7.4.4 Discussion

Figure 5.28 displays the quality of all four feature models measured by validity, F-measure, precision, and recall. We first look at the subjects from a known oracle (ASU and kill switch). Both models achieved high validity with an average of 98.16% for the kill switch and 94.37% for ASU. Recall is also high for both models (up to 99.8%). This demonstrates that we can achieve

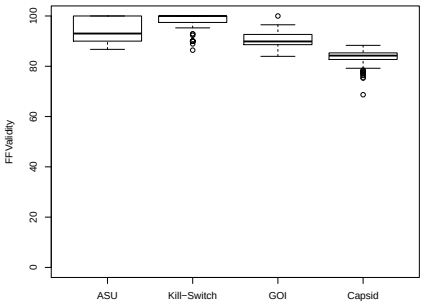

Figure 5.24 Validity

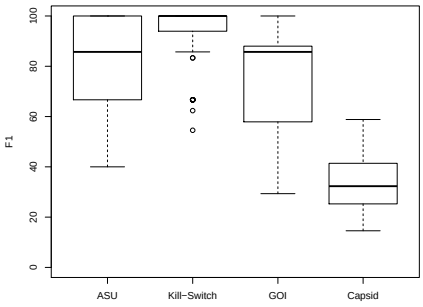

Figure 5.25 F-measure

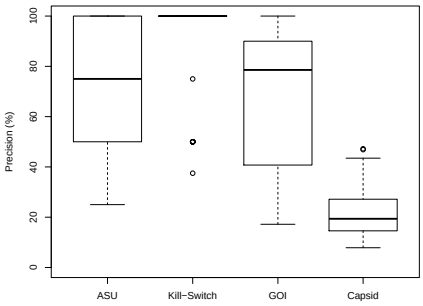

Figure 5.26 Precision (%)

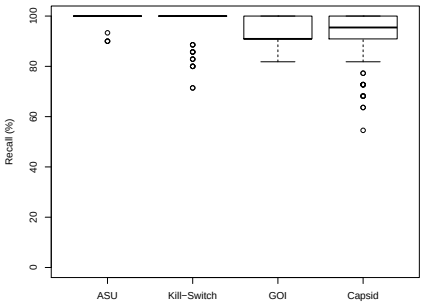

Figure 5.27 Recall (%)

Figure 5.28: Metrics on all four reverse engineered models over all runs.

high coverage of the input products when reverse engineering. However, looking at precision we see more variation. The kill switch has the highest average precision at 94.13% while the ASU model precision drops to an average of 72.7%. A low precision means the tool is generating excessive products (false positives). While this is undesired if we have a complete product set, 17 of the 40 runs do achieve 100% precision.

In the case of an incomplete set of products (GOI and capsid), the average validity is 90.4% for GOI and 83.3% for capsid. While the recall is high for both (average of 93.4% for GOI and 92.3% for capsid) the precision is relatively low (average of 66.7% for GOI and 21.5% for capsid). Again this is due to some relaxation of the reverse engineered model that allows extra products. Since we are knowingly building a model from an incomplete set of products, this can be a desired quality so the results do not overfit the input set of products. Additional products in the engineered feature model can open up new related design decisions for the biologist.

We note that the viral vector models do not perform with as high validity as the ASU or kill switch. This result was in part expected due to the different methods of reverse engineering. We removed any basic parts in our set of products as described in Section 5.5.3.4 to only include composite parts. However, the composite parts may or may not form a complete product. The catalog can provide composite parts that are missing a component (for example the terminator is left out). This is done to allow teams to order partial parts and fill them in with their own choices. These parts are useful in the repository to allow for modularity, but can have a negative effect if used as input to reverse engineering tools. Furthermore, in many cases not all possible products will be in the repository. Thus, we should not expect or desire 100% validity. 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*.

Looking towards tool development, we may also want to add additional features (from the BioBricks repository) and show the potentially larger model. Ultimately, if we can represent BioBricks features using a constraint language, there is an opportunity for direct reverse-engineering and greater scalability. SPLRevO has been evaluated on reverse-engineering a feature model up to 100 features when using input constraints [148]. Furthermore the ASU model timed out on 400 generations highlighting the need for more advanced reverse engineering algorithms. We discuss this further in Section 5.8.

5.7.4.5 Summary of RQ4.

We performed reverse engineering for two use cases, in the presence of a known oracle and in the absence of domain knowledge. 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 minutes with the exception of the ASU model which took on average 17.15 hours due to its large size.

We have demonstrated that we can accurately (by achieving 100% validity in three of four subjects) automatically reverse engineer feature models of organic programs. We show how even with an incomplete set of products we can create an initial feature model for engineers. As we discuss in our future work, it is these initial feature models that could be shared through collaborative feature model tooling to assist synthetic engineers in their design stages.

5.8 Implications: The Future of OSPL Engineering

In our evaluation of OSPLs, we discovered several challenges that we believe provide a roadmap to extend SPL engineering research. In the following discussion, we highlight the new challenges that have emerged. We also discuss how this might extend to other emerging software product lines such as other open-source applications, IoT, robotics, and more.

5.8.1 Need for Tools Supporting SPL Evolution

Most software systems have a cycle of evolution and maintenance. Therefore, their software product lines can be dynamic and require modifications over time. This is especially evident in open-source product lines. For example, in the model that Montalvillo and Díaz present [65], the feature model can be updated as commits are made which include major alterations to current features, removing features, or adding in new features.

We see in Section 5.7.1 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. On one hand, mandatory features such as terminators, may be added. While the structure of the feature models will not change, there will be more options added to OR and Alternative groups. As seen by Figure 5.5, parts of this nature are added less often.

On the other hand, new abstract features or coding parts may be added which will change the architecture of the feature model.

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. Recently there has been an increase in tool development such as by Kuiter et al. [151], where users can view and modify feature models dynamically. In the domain of OSPL, we hypothesize each year's teams will utilize the models for their projects, and add both new parts and any experimental results in the form of constraints or annotations. We see this use extending to other evolving SPLs such as IoT.

5.8.2 Need for SPL Constructs that Support Duplicate Features

We observed some parts can be duplicated across the model. For example, in the ASU model seen in Figure 5.12, the same proteins (Las, Lux, and Tra) can be used in both the receiver and the reporter. Feature models do not typically allow for the same feature to be repeated. This issue can be resolved by adding a tag to the features, however this eliminates potentially useful domain knowledge. Consider a function in a traditional software product line being modeled as a feature. This function is defined once, but can be called multiple times. It may be more cost-effective to reuse this feature in multiple locations than to define a new function. In the domain of IoT, the same device or sensor may appear in multiple locations. This type of information would be important to note and record.

Currently, this type of design cannot be accounted for in feature modeling languages. We see the need for future work in alternative model abstractions to allows designs that may have duplicated features or other architectural elements.

5.8.3 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 [148] 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.

5.8.4 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.

5.8.5 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.

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 (Figure 5.1) as `ifdefs`, it might look like the pseudocode in Listing 5.1. 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 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 [88]. There has also been recent work on automation of genetic designs [152]. 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.

5.9 Conclusions and Future Work

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

Listing 5.1: Pseudocode defining the variability in a cell-to-cell signaling system.

```
//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
```

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.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

In this dissertation we have brought light to the challenges surrounding users of configurable software in the biosciences and propose methods to increase their interpretability. After motivating the need for configurability awareness we developed methods to increase interpretability for users of bioinformatics tools as well as synthetic biologists.

We have shown via a case study that manipulating configuration options in three popular bioinformatics programs can lead to variability in functional output, performance, and in finding errors. We find the functionality of these systems are dependent on the chosen configuration options. We also find that understanding the configuration models and underlying constraints is non-trivial and that developers can benefit from configuration-aware testing tools and existing sampling techniques. We have provided a set of actions developers can take to improve the repeatability and dependability of these systems.

Second, we developed the ICO framework for interpretable configuration options to assist users in learning the effects of configuration options in their tools. ICO can identify configurations that improve over the default, and reports the configuration options causing that effect in a simplistic format.

Last, we presented the notion of organic software product lines. We used the largest open source DNA repository to show 1) there are assets which are reused and products that share common and variable elements, 2) we can build feature models to represent the products in this repository, and 3) how common SPL techniques can be used to benefit product line development in this domain.

In future work on bioinformatics tools we would like to find a way to communicate our findings effectively to the bioinformatics and biology communities. We would also like to further explore the use of sampling methods to see how they compare on bioinformatics tools; do we continue to see higher-order interactions. As future work we would like to perform a broader analysis on the ICO

framework. Our vision is to have interpretable configuration options for the end user, thus a user study is warranted. Related to OSPLs, we would like to integrate software product line engineering into existing synthetic biology practices. We leave as future work to develop methods to integrate feature modeling with DNA repositories allowing synthetic biologists to utilize software product line engineering methods.

BIBLIOGRAPHY

[1] Van Noorden, Richard, Brendan Maher, and Regina Nuzzo. The top 100 papers. *Nature*, 514 (7524):550, 2014. ISSN 0028-0836.

[2] Altschul, Stephen F., Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[3] Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[4] Levin, Clément, Emeric Dynomant, Bruno J. Gonzalez, Laurent Mouchard, David Landsman, Eivind Hovig, and Kristian Vlahovicek. A data-supported history of bioinformatics tools. *CoRR*, abs/1807.06808, 2018.

[5] Perrin, Helene, Marion Denorme, Julien Grosjean, OMICtools community, Emeric Dynomant, Vincent J. Henry, Fabien Pichon, Stéfan Jacques Darmoni, Arnaud Desfeux, and Bruno J. Gonzalez. Omictools: a community-driven search engine for biological data analysis. *CoRR*, abs/1707.03659, 2017.

[6] Henry, Vincent, Anita Bandrowski, Anne-Sophie Pepin, Bruno Gonzalez, and Arnaud Desfeux. Omictools: An informative directory for multi-omic data analysis. *Database*, 2014, 2014.

[7] Medeiros, Flávio, Iran Rodrigues, Márcio Ribeiro, Leopoldo Teixeira, and Rohit Gheyi. An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pages 35–44. ACM, 2015.

[8] Cohen, Myra B., Joshua Snyder, and Gregg Rothermel. Testing across configurations: Implications for combinatorial testing. *SIGSOFT Softw. Eng. Notes*, 31(6):1–9, November 2006. ISSN 0163-5948.

[9] Robinson, Brian, Mithun Acharya, and Xiao Qu. Configuration selection using code change impact analysis for regression testing. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM, pages 129–138. IEEE Computer Society, 2012.

[10] Nguyen, ThanhVu, Ugur Koc, Javran Cheng, Jeffrey S. Foster, and Adam A. Porter. iGen: Dynamic interaction inference for configurable software. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 655–665. ACM, 2016.

[11] Reisner, Elnatan, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *International Conference on Software Engineering*, pages 445–454. IEEE, May 2010.

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

[13] Miller, Greg. Scientific publishing. a scientist's nightmare: software problem leads to five retractions. *Science (New York, N.Y.)*, 314(5807):1856, 2006. ISSN 0036-8075.

[14] Ichikawa, Takanari, Yoshihito Suzuki, Inge Czaja, Carla Schommer, Angela Leßnick, Jeff Schell, and Richard Walden. Retraction note to: Identification and role of adenylyl cyclase in auxin signalling in higher plants. *Nature*, 396, 11 1998.

[15] Morrison-Smith, Sarah, Christina Boucher, Andrea Bunt, and Jaime Ruiz. Elucidating the role and use of bioinformatics software in life science research. In *Proceedings of the British HCI Conference*, pages 230–238. ACM, July 2015.

[16] Qu, Xiao, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis*, ISSTA, pages 75–86. ACM, 2008.

[17] Yilmaz, Cemal, Myra B. Cohen, and Adam Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31 (1):20–34, 2006.

[18] Cohen, Myra B., Matthew B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[19] Medeiros, Flávio, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 643–654. ACM, 5 2016.

[20] Nita, Marius and David Notkin. White-box approaches for improved testing and analysis of configurable software systems. In *International Conference on Software Engineering (ICSE)*, pages 307–310. IEEE, May 2009.

[21] Meinicke, Jens, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 483–494. ACM, 9 2016.

[22] Kamali, Amir Hossein, Eleni Giannoulatou, Tsong Yueh Chen, Michael A. Charleston, Alistair L. McEwan, and Joshua W.K. Ho. How to test bioinformatics software? In *Biophysical Reviews*, volume 7, pages 343–352. Springer Berlin Heidelberg, 2015.

[23] Weyuker, Elaine J. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[24] Vogel, Thomas, Stephan Druskat, Markus Scheidgen, Caludia Draxl, and Lars Grunske. Challenges for verifying and validating scientific software in computational materials science. In *International Workshop on Software Engineering for Science*, SE4Science '19, pages 25–32. IEEE, 2019.

[25] Hannay, J. E., C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, 2009.

[26] Segal, Judith. Scientists and software engineers: A tale of two cultures. In *Proceedings of the 20th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2008, Lancaster, UK, September 10-12, 2008*, page 6. Psychology of Programming Interest Group, 2008.

[27] Segal, Judith. When software engineers met research scientists: A case study. *Empirical Software Engineering*, 10(4):517–536, 2005. ISSN 1382-3256.

[28] iGEM Registry. Registry of standard biological parts. iGEM Foundation. https://parts.igem.org, 2018.

[29] Sayagh, M., N. Kerzazi, B. Adams, and F. Petrillo. Software configuration engineering in practice: Interviews, survey, and systematic literature review. *IEEE Transactions on Software Engineering*, 2018.

[30] Kang, Kyo, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[31] Passos, Leonardo, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, 21(4):1744–1793, August 2016. ISSN 1573-7616.

[32] Siegmund, Norbert, Alexander Grebhahn, Christian Kästner, and Sven Apel. Performance-influence models for highly configurable systems. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 284–294. ACM Press, 8 2015.

[33] Siegmund, Norbert, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 167–177. IEEE Press, 2012. ISBN 9781467310673.

[34] Grebhahn, Alexander, Sebastian Kuckuk, Christian Schmitt, Harald Köstler, Norbert Siegmund, Sven Apel, Frank Hannig, and Jürgen Teich. Experiments on optimizing the performance of stencil codes with SPL Conqueror. In *Parallel Processing Letters*, volume 24, September 2014.

[35] Grebhahn, Alexander, Carmen Rodrigo, Norbert Siegmund, Francisco J. Gaspar, and Sven Apel. Performance-influence models of multigrid methods: A case study on triangular grids. In *Concurrency and Computation: Practice and Experience*, volume 29. John Wiley & Sons, Ltd., January 2017.

[36] Alves Pereira, Juliana, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. Sampling effect on performance prediction of configurable systems: A case study. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 277–288, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450369916.

[37] Fisher, Ronald Aylmer et al. The design of experiments. *The design of experiments.*, (7th Ed), 1960.

[38] Grebhahn, Alexander, Norbert Siegmund, and Sven Apel. Predicting performance of software configurations: There is no silver bullet, 2019.

[39] Petke, Justyna, Myra B. Cohen, Mark Harman, and Shin Yoo. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 26–36. ACM, August 2013.

[40] Cohen, David M., Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.

[41] A.S. Hedayat, N.J.A. Sloane, John Stufken. *Orthogonal arrays : theory and applications.* Springer series in statistics. Springer, New York, 1999. ISBN 0387987665.

[42] Kuhn, D. Richard, Dolores R Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

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

[44] Lei, Yu, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence. Ipog: A general strategy for t-way software testing. In *ECBS '07. 14th Annual IEEE International Conference and Workshops on the*, pages 549–556, March 2007.

[45] Yu, Linbin, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. Acts: A combinatorial test generation tool. In *International Conference on Software Testing, Verification and Validation*, pages 370–375, March 2013.

[46] Cohen, Myra B., Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, page 38–48, USA, 2003. IEEE Computer Society. ISBN 076951877X.

[47] Garvin, Brady J., Myra B. Cohen, and Matthew B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. In *Empirical Software Engineering (EMSE)*, volume 16, pages 61–102. Springer US, February 2010.

[48] Song, Charles, Adam Porter, and Jeffrey S. Foster. itree: Efficiently discovering high-coverage configurations using interaction trees. *IEEE Transactions on Software Engineering*, 40(3): 251–265, March 2014.

[49] Kim, Chang Hwan Peter, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. SPLat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 257–267. ACM, 2013.

[50] Jamshidi, Pooyan, Miguel Velez, Christian Kästner, Norbert , and Prasad Kawthekar. Transfer learning for improving model predictions in highly configurable software. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, pages 31–41. IEEE, 2017.

[51] Jamshidi, Pooyan, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 497–508. IEEE, 11 2017.

[52] Grebhahn, Alexander, Christian Engwer, Matthias Bolten, and Sven Apel. Variability of stencil computations for porous media. In *Concurrency and Computation: Practice and Experience*, volume 29. John Wiley & Sons, Ltd., March 2017.

[53] Schmitt, Christian, Sebastian Kuckuk, Frank Hannig, Jürgen Teich, Harald Köstler, Ulrich Rüde, and Christian Lengauer. Systems of partial differential equations in exaslang. In *Software for Exascale Computing - SPPEXA*, volume 113 of *Lecture Notes in Computational Science and Engineering*, pages 47–67. Springer, 2016.

[54] Lund, Henrik Hautop. Lessons learned in designing user-configurable modular robotics. In *Robot Intelligence Technology and Applications (RiTA)*, volume 274, pages 279–286. Springer, 2013.

[55] Franco, Anthony Di, Hui Guo, and Cindy Rubio-González. A comprehensive study of real-world numerical bug characteristics. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, October 2017.

[56] Lundgren, Anders and Upulee Kanewala. Experiences of testing bioinformatics programs for detecting subtle faults. In *Proceedings of the International Workshop on Software Engineering for Science - SE4Science*, pages 16–22. IEEE, 2016.

[57] Clements, Paul and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2002. ISBN 0201703327.

[58] Benavides, David, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6):615–636, 2010.

[59] Thüm, Thomas, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, 47(1):1–45, 2014.

[60] Swanson, Jacob, Myra B. Cohen, Matthew B. Dwyer, Brady J. Garvin, and Justin Firestone. Beyond the rainbow: Self-adaptive failure avoidance in configurable systems. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 377–388, 2014.

[61] Lotufo, Rafael, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the linux kernel variability model. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC, pages 136–150, September 2010.

[62] Cleland-Huang, Jane, Michael Vierhauser, and Sean Bayley. Dronology: An incubator for cyber-physical systems research. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE, pages 109–112, August 2018.

[63] Lutz, Robyn R., Jack H. Lutz, James I. Lathrop, Titus H. Klinge, Divita Mathur, Donald M. Stull, Taylor G. Bergquist, and Eric R. Henderson. Requirements analysis for a product family of DNA nanodevices. In *Proceedings of the 20th IEEE International Requirements Engineering Conference*, RE, pages 211–220, September 2012.

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

[65] Montalvillo, Leticia and Oscar Díaz. Tuning github for spl development: branching models & repository operations for product engineers. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC, pages 111–120, Jul 2015.

[66] Plakidas, Konstantinos, Srdjan Stevanetic, Daniel Schall, Tudor B. Ionescu, and Uwe Zdun. 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, pages 89–98, September 2016.

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

[68] Nadi, Sarah, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*, pages 140–151, 2014.

[69] Lopez-Herrejon, Roberto Erick, José A. Galindo, David Benavides, Sergio Segura, and Alexander Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In Fraser, Gordon and Jerffeson Teixeira de Souza, editors, *Search Based Software Engineering*, pages 168–182. Springer Berlin Heidelberg, 2012.

[70] Galindo, José A, David Benavides, and Sergio Segura. Debian packages repositories as software product line models. towards automated analysis. In *ACoTA*, pages 29–34, 2010.

[71] She, Steven, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE, pages 461–470. ACM, 2011.

[72] Polanski, Andrzej. *Bioinformatics by Andrzej Polanski, Marek Kimmel.* Springer, Berlin ; New York, 1 edition, 2007. ISBN 3-540-69022-0.

[73] Patti, Gary J., Oscar Yanes, and Gary Siuzdak. Metabolomics: the apogee of the omics trilogy. *Nature reviews Molecular cell biology*, 13(4):263–269, 2012.

[74] Schneider, Maria V. and Sandra Orchard. Omics technologies, data and bioinformatics principles. In *Bioinformatics for omics Data*, pages 3–30. Springer, 2011.

[75] Altschul, Stephen F., Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215, 2018. https://blast.ncbi.nlm.nih.gov/.

[76] Durbin, Richard, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[77] Kitano, Hiroaki. Systems biology: a brief overview. *science*, 295(5560):1662–1664, 2002.

[78] Westerhoff, Hans V. and Bernhard O. Palsson. The evolution of molecular biology into systems biology. *Nature biotechnology*, 22(10):1249–1252, 2004.

[79] Henry, Christopher S., Matthew DeJongh, Aaron A. Best, Paul M. Frybarger, Ben Linsay, and Rick L. Sevens. High-throughput generation, optimization and analysis of genome-scale metabolic models. *Nature Biotechnology*, 28(9):977–982, 2010.

[80] Arkin, Adam P, Robert W Cottingham, Christopher S Henry, Nomi L Harris, Rick L Stevens, Sergei Maslov, et al. KBase: The United States Department of Energy Systems Biology Knowledgebase. *Nature Biotechnology*, 36:566–569, 2018.

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

[82] Whitaker, William B., Nicholas R. Sandoval, Robert K. Bennett, Alan G. Fast, and Eleftherios T. Papoutsakis. Synthetic methylotrophy: engineering the production of biofuels and chemicals based on the biology of aerobic methanol utilization. *Current Opinion in Biotechnology*, 33:165–175, 2015. URL https://www.sciencedirect.com/science/article/pii/S0958166915000166.

[83] Rossello, Ricardo A. and David H. Kohn. Cell communication and tissue engineering. *Communicative & Integrative Biology*, 3(1):53–56, 2010.

[84] Weber, Wilfried and Martin Fussenegger. Emerging biomedical applications of synthetic biology. *Nature Reviews Genetics*, 13(1):21–35, 2012.

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

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

[87] SBOL. Synthetic biology open language. SBOL Research Group. https://sbolstandard.org, 2019.

[88] Cai, Yizhi, Mandy L. Wilson, and Jean Peccoud. Genocad for iGEM: a grammatical approach to the design of standard-compliant constructs. *Nucleic Acids Research*, 38(8):2637–2644, 2010.

[89] Nielsen, Alec A.K., Bryan S. Der, Jonghyeon Shin, Prashant Vaidyanathan, Vanya Paralanov, Elizabeth A. Strychalski, David Ross, Douglas Densmore, and Christopher A. Voigt. Genetic circuit design automation. *Science*, 352(6281), 2016.

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

[91] Tavella, Federico, Alberto Giaretta, Triona Marie Dooley-Cullinane, Mauro Conti, Lee Coffey, and Sasitharan Balasubramaniam. DNA molecular storage system: Transferring digitally encoded information through bacterial nanonetworks. *IEEE Transactions on Emerging Topics in Computing*, 2018.

[92] Elowitz, Michael B. and Stanislas Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403:335–338, 2000.

[93] Gardner, Timothy S., Charles R. Cantor, and James J. Collins. Construction of a genetic toggle switch in escherichia coli. *Nature*, 402:339–342, 2000.

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

[95] iGEM Competition. International genetically engineered machine competition. iGEM Foundation. https://igem.org, 2018.

[96] Cashman, Mikaela, Myra B. Cohen, Priya Ranjan, and Robert W. Cottingham. Navigating the maze: The impact of configurability in bioinformatics software. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 757–767, New York, NY, USA, 2018. ACM.

[97] Jin, Dongpu, Xiao Qu, Myra B. Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *International Conference on Software Engineering, Software in Practice Track*, ICSE, pages 215–225. ACM, 2014.

[98] kbase-assembly. Microbial genome assembly and annotation tutorial. https://narrative.kbase.us/narrative/notebooks/ws.18188.obj.6, March 2017.

[99] Li, Dinghua, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph. *Bioinformatics*, 31(10), 2015.

[100] Danial, Al. CLOC - Count Lines of Code. http://cloc.sourceforge.net/, 2015.

[101] of Medicine, U.S. National Library. National center for biotechnology information. https://www.ncbi.nlm.nih.gov/.

[102] HCC. Using BLAST on HCC. https://github.com/unlhcc/job-examples, April 2017.

[103] kb-megahit. KBase MEGAHIT SDK Repository. https://github.com/kbaseapps/kb_megahit, November 2017.

[104] Henry, Christopher S. MFAToolkit GitHub Repository. https://github.com/cshenry/fba_tools/tree/master/MFAToolkit, December 2017.

[105] Henard, Christopher, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. *IEEE/ACM International Conference on Software Engineering (ICSE)*, 1:517–528, 2015.

[106] Nadi, Sarah, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Where do configuration constraints stem from? an extraction approach and an empirical study. *IEEE Transactions on Software Engineering (TSE)*, 41:820–841, March 2015.

[107] Roscher, Ribana, Bastian Bohn, Marco F. Duarte, and Jochen Garcke. Explainable machine learning for scientific insights and discoveries. *IEEE Access*, 8:42200–42216, 2020.

[108] Adadi, Amina and Mohammed Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (XAI). *IEEE Access*, 6:52138–52160, 2018.

[109] Gunning, David. Explainable artificial intelligence (XAI). *Defense Advanced Research Projects Agency (DARPA), nd Web*, 2, 2017.

[110] Carvalho, Diogo V., Eduardo M. Pereira, and Jaime S. Cardoso. Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8):832, 2019.

[111] Du, Mengnan, Ninghao Liu, and Xia Hu. Techniques for interpretable machine learning. *Communications of the ACM*, 63(1):68–77, 2019.

[112] Molnar, Christoph. *Interpretable machine learning*. Lulu.com, 2019.

[113] Rüping, Stefan. *Learning interpretable models*. PhD thesis, Dortmund Technical University, 2006.

[114] Ladd, Scott Robert. Analysis of compiler options via evolutionary algorithm. https://github.com/Acovea/libacovea, 2003.

[115] Zeljko Juric, Sebastian Reichelt, Kevin Kofler. GCC command-line options. TIGCC Team. http://tigcc.ticalc.org/doc/comopts.html, 2020.

[116] Xu, Tianyin, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on foundations of software engineering*, ESEC/FSE 2015, pages 307–319. ACM, 2015. ISBN 9781450336758.

[117] Zhang, Sai and Michael D. Ernst. Which configuration option should I change? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 152–163, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565.

[118] Ko, Andrew J. and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 301–310, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791.

[119] Ko, Andrew J. and Brad A Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, 2004.

[120] Ko, Andrew J. and Brad A. Myers. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1569–1578, 2009.

[121] Sayagh, Mohammed, Noureddine Kerzazi, Fabio Petrillo, Khalil Bennani, and Bram Adams. What should your run-time configuration framework do to help developers? *Empirical Software Engineering*, 2020. URL https://doi.org/10.1007/s10664-019-09790-x.

[122] Hamming, R. W. (Richard Wesley). *Coding and information theory / R. W. Hamming.* Prentice-Hall, Englewood Cliffs, N.J., 1980. ISBN 0131391399.

[123] Iowa State University. Speedy5-8 hardware. Iowa State University LAS Research IT. https://researchit.las.iastate.edu/speedy5, 2020.

[124] Ting, Kai Ming. *Confusion Matrix.* Springer US, Boston, MA, 2017. ISBN 978-1-4899-7687-1.

[125] Cashman, Mikaela, Justin Firestone, Myra B. Cohen, Thammasak Thianniwet, and Wei Niu. DNA as features: Organic software product lines. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, SPLC '19, pages 108–118, New York, NY, USA, 2019. ACM.

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

[127] Cetina, C., P. Giner, J. Fons, and V. Pelechano. Using feature models for developing self-configuring smart homes. In *5th International Conference on Autonomic and Autonomous Systems*, pages 179–188, April 2009.

[128] Tzeremes, Vasilios and Hassan Gomaa. A software product line approach to designing end user applications for the internet of things. In *ICSOFT*, 2018.

[129] Arizona State University. ASU iGEM 2017: Engineering variable regulators for a quorum sensing toolbox. https://2017.igem.org/Team:Arizona_State, 2017.

[130] Feinberg, Martin. Chemical reaction network structure and the stability of complex isothermal reactors–I. the deficiency zero and deficiency one theorems. *Chemical Engineering Science*, 42:2229–2268, 1987.

[131] Fages, François, Guillaume Le Guludec, Olivier Bournez, and Amaury Pouly. Strong turing completeness of continuous chemical reaction networks and compilation of mixed analog-digital programs. In *Proceedings of the 15th International Conference on Computational Methods in Systems Biology*, CMSB, pages 108–127, September 2017.

[132] Valverde, Sergi, Manuel Porcar, Juli Peretó, and Ricard V. Solé. The software crisis of synthetic biology. *bioRxiv*, 2016. URL https://www.biorxiv.org/content/early/2016/02/29/041640.

[133] Firestone, Justin and Myra B. Cohen. The assurance recipe: facilitating assurance patterns. In *Proceedings of the International Conference on Computer Safety, Reliability, and Security (SAFECOMP), ASSURE Workshop*, pages 22–30, September 2018.

[134] Quan, Jiayuan and Jingdong Tian. Circular polymerase extension cloning of complex gene libraries and pathways. *PloS one*, 4(7):1–6, 2009.

[135] Miller, Melissa B. and Bonnie L. Bassler. Quorum sensing in bacteria. *Annual Review of Microbiology*, 55(1):165–199, 2001. PMID: 11544353.

[136] Stirling, Finn, Lisa Bitzan, Samuel O'Keefe, Elizabeth Redfield, John W.K. Oliver, Jeffrey Way, and Pamela A. Silver. Rational design of evolutionarily stable microbial kill switches. *Molecular Cell*, 68(4):686 – 697, 2017. ISSN 1097-2765.

[137] Stirling, Finn, Alexander Naydich, Juliet Bramante, Rachel Barocio, Michael Certo, Hannah Wellington, Elizabeth Redfield, Samuel O'Keefe, Sherry Gao, Adam Cusolito, Jeffrey Way, and Pamela Silver. Synthetic cassettes for pH-mediated sensing, counting and containment. *bioRxiv*, 2019.

[138] Whitford, Christopher M., Saskia Dymek, Denise Kerkhoff, Camilla März, Olga Schmidt, Maximilian Edich, Julian Droste, Boas Pucker, Christian Rückert, and Jörn Kalinowski. Auxotrophy to Xeno-DNA: an exploration of combinatorial mechanisms for a high-fidelity biosafety system for synthetic biology applications. *Journal of Biological Engineering*, 12(1), August 2018.

[139] Naso, Michael F., Brian Tomkowicz, William L. Perry 3rd, and William R. Strohl. Adeno-associated virus (AAV) as a vector for gene therapy. *BioDrugs*, 31:317 – 334, 2017.

[140] Levine, Fred and Gil Leibowitz. Towards gene therapy of diabetes mellitus. *Molecular Medicine Today*, 5(4):165 – 171, 1999. ISSN 1357-4310.

[141] Aponte-Ubillus, Juan Jose, Daniel Barajas, Joseph Peltier, Cameron Bardliving, Parviz Shamlou, and Daniel Gold. Molecular design for recombinant adeno-associated virus (rAAV) vector production. *Appl Microbiol Biotechnol*, 102:1045–1054, 2018.

[142] iGEM API. Registry of standard biological parts API. iGEM Foundation. https://parts.igem.org/Registry_API, 2018.

[143] Freiburg Bioware. Freiburg bioware iGEM 2010: Virus construction kit for therapy. http://2010.igem.org/Team:Freiburg_Bioware, 2010. Last Accessed: November 6, 2019.

[144] Tekel, Stefan J., Christina L. Smith, Briana Lopez, Amber Mani, Christopher Connot, Xylaan Livingstone, and Karmella Ann Haynes. Engineered orthogonal quorum sensing systems for synthetic gene regulation in escherichia coli. *Frontiers in Bioengineering and Biotechnology*, 7(80), 2019.

[145] Thüm, Thomas, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70 – 85, 2014. ISSN 0167-6423. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[146] Garvin, Brady J., Myra B. Cohen, and Matthew B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16 (1):61–102, 2011.

[147] Thianniwet, Thammasak and Myra. B. Cohen. 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, pages 1–16, February 2015.

[148] Thianniwet, Thammasak and Myra. B. Cohen. Scaling up the fitness function for reverse engineering feature models. In *Symposium on Search-Based Software Engineering*, SSBSE, pages 128–142, October 2016.

[149] Benavides, David, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-cortés. 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, pages 129–134, January 2007.

[150] Zhu, Jiaxin, Minghui Zhou, and Audris Mockus. 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*, pages 1–4. ACM, September 2014.

[151] Kuiter, Elias, Sebastian Krieter, Jacob Krüger, Thomas Leich, and Gunter Saake. Foundations of collaborative, real-time feature modeling. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, SPLC, pages 257–264. ACM, 2019.

[152] Storch, Marko, Matthew C. Haines, and Geoff S. Baldwin. DNA-BOT: A low-cost, automated DNA assembly platform for synthetic biology. *bioRxiv*, 2019.

APPENDIX A. CONFIGURATION MODELS

We display the configuration models for all three of our subject programs from Chapter 3. For additional model information such as which options were removed please see the supplementary website associated with the relevant publication at https://github.com/mikacashman/ASE18SupResources.

Table A.1: BLAST configuration model for Chapter 3.

Parameter Name	Category	Type	Range Tested
dust	Query filtering options	String	yes, "20 64 1", no
soft_masking	Query filtering options	Boolean	TRUE, FALSE
lcase_masking	Query filtering options	Flag	TRUE, FALSE
xdrop_ungap	Extension options	Real	0, 0.1, 0.5, 20, 100
xdrop_gap	Extension options	Real	0, 0.1, 0.5, 30, 100
xdrop_gap_final	Extension options	Real	0, 0.1, 0.5, 10, 100
ungapped	Extension options	Flag	TRUE, FALSE

Table A.2: MEGAHIT configuration model for Chapter 3.

Parameter Name	Type	Range Allowed	Range Tested
min-count	Numeric	[2,10]	2, 4, 6, 8, 10
k-min	Numeric	odd in range [1,127]	15, 21, 59, 99, 141
k-max	Numeric	odd in range [1,255]	15, 21, 59, 99, 141
k-step	Numeric	even in range [2,28]	2, 6, 12, 20, 28

Table A.3: FBA-GUI configuration model for Chapter 3.

Parameter Name	Type	Range Tested
flux variability analysis	Boolean	TRUE, FALSE
minimize flux	Boolean	TRUE, FALSE
simulate all single KOs	Boolean	TRUE, FALSE
max Carbon uptake	Float	0, 25, 50, 75, 100
max Nitrogen uptake	Float	0, 25, 50, 75, 100
max Phosphate uptake	Float	0, 25, 50, 75, 100
max Sulfur uptake	Float	0, 25, 50, 75, 100
max Oxygen uptake	Float	0, 25, 50, 75, 100

Table A.4: FBA-MFA configuration model for Chapter 3.

Parameter Name	Type	Range Tested
minimize flux	Boolean	TRUE, FALSE
simulate all single Kos	Boolean	TRUE, FALSE
find_min_media	Boolean	TRUE, FALSE
allRevMFA	Boolean	TRUE, FALSE
useVarDrainFlux	Boolean	TRUE, FALSE
rxnKOSen	Boolean	TRUE, FALSE
decompDrain	Boolean	TRUE, FALSE
decompRxn	Boolean	TRUE, FALSE
tightBounds	Boolean	TRUE, FALSE
phenoFit	Boolean	TRUE, FALSE
forceUse	Boolean	TRUE, FALSE
maxActiveRxn	Boolean	TRUE, FALSE
newPipeline	Boolean	TRUE, FALSE
prefMFA	Boolean	TRUE, FALSE
rxnUseVar	Boolean	TRUE, FALSE
recMILP	Boolean	TRUE, FALSE
useDataFields	Boolean	TRUE, FALSE
simpVar	Boolean	TRUE, FALSE
LPFile	Boolean	TRUE, FALSE
varKey	Boolean	TRUE, FALSE
optimMetabo	Boolean	TRUE, FALSE
maxDrain	Float	0, 250, 500 ,750, 1000
minDrain	Float	0, -250, -500 , -750, -1000
deltaGSlack	Float	0, 5, 10, 15, 20
maxDeltaG	Float	0, 2500, 5000, 7500, 10000
maxFlux	Float	0, 250, 500 ,750, 1000
minFluxMulti	Float	0, 1, 2, 3, 4
minFlux	Float	0, -250, -500 , -750, -1000
conObjec	Float	0, 0.05, 0.1, 0.15, 0.2
minTargetFlux	Float	0, .005, .01, .015, .02
uptake limits	float	permutations of 5 [0,25,50,75,100]

We display the configuration models for all three of our subject programs from Chapter 4. We show each model in full, and if the model was refined in any step we show the result and reasoning behind the refinement.

Table A.5: The complete configuration model for BLAST used in the evaluation of ICO. The *category* is defined from the command line help output.

Parameter Name	Category	Type	Range Tested
dust	Query filtering	String	yes, “20 64 1”, no
soft_masking	Query filtering	Boolean	TRUE, FALSE
lcase_masking	Query filtering	Flag	TRUE, FALSE
xdrop_ungap	Extension	Real	0, 0.1, 0.5, 20, 100
xdrop_gap	Extension	Real	0, 0.1, 0.5, 30, 100
xdrop_gap_final	Extension	Real	0, 0.1, 0.5, 10, 100
ungapped	Extension	Flag	TRUE, FALSE
word_size	General search	Integer	4, 28, 40, 100, 1000
min_raw_gapped_score	Extension	Integer	NULL, 0, 10, 100, 1000
strand	Input query	String	“both”,“minus”,“plus”
use_index	General search	Boolean	TRUE, FALSE
show_gis	Formatting	Flag	TRUE, FALSE
num_descriptions	Formatting	Integer	0, 10, 100, 500, 1000
num_alignments	Formatting	Integer	0, 10, 100, 250, 1000
line_length	Formatting	Integer	1, 5, 20, 60, 100
html	Formatting	Flag	TRUE, FALSE
max_target_seqs	Restrict search or results	Integer	1, 10, 100, 500, 1000
no_greedy	Extension	Flag	TRUE, FALSE
sum_stats	Statistical	Boolean	TRUE, FALSE
parse_deflines	Miscellaneous	Flag	TRUE, FALSE
num_threads	Miscellaneous	Integer	1, 2, 5, 10, 100
remote	Miscellaneous	Flag	TRUE, FALSE

Table A.6: All three configuration models for the BLAST subject for ICO. The initial configuration is in the middle and has 22 configuration options and 76 values in total. The left shows the model for the Distance 2 experiments. We removed any options that caused an error, a warning signaling the option had no effect, or a constraint with the default configuration. This Distance 2 model has 17 configuration options and 58 values. The right shows the model for the covering arrays and the random experiments. We removed parameters that caused an error, timed out, or had a constraint with the default configuration. We also added in three values in order to prevent a constraint marked with the asterisk (note we did not need these in the Distance 2 experiments and they were removed for a warning). This covering array and random model has 19 configuration options and 71 values.

Distance 2 Model	
Reason	**Values**
	T,F
Error	-
	T,F
	T,F
	T,F
	T,F
	T,F
Constraint	-
	T,F
Error	-
	1,10,100,500,1000
Invalid Value	1,2,5,10
	NULL,0,10,100,1000
No Effect	-
No Effect	10,100,250,1000
No Effect	-
	4,28,40,100,1000
	0,0.1,0.5,20,100
	0,0.1,0.5,30,100
	0,0.1,0.5,10,100
	yes,“20 64 1”,no
	“both”,“minus”,“plus”

⟸

Initial Configuration Model	
Option Name	**Values**
soft_masking	T,F
use_index	T,F
sum_stats	T,F
lcase_masking	T,F
ungapped	T,F
show_gis	T,F
html	T,F
no_greedy	T,F
parse_deflines	T,F
remote	T,F
max_target_seqs	1,10,100,500,1000
num_threads	1,2,5,10,100
min_raw_gapped_score	NULL,0,10,100,1000
num_descriptions	0,10,100,500,1000
num_alignments	0,10,100,250,1000
line_length	1,5,20,60,100
word_size	4,28,40,100,1000
xdrop_ungap	0,0.1,0.5,20,100
xdrop_gap	0,0.1,0.5,30,100
xdrop_gap_final	0,0.1,0.5,10,100
dust	yes,“20 64 1”,no
strand	“both”,“minus”,“plus”

⟹

CA/Random Model	
Reason	**Values**
	T,F
Error	-
	T,F
	T,F
	T,F
	T,F
	T,F
Constraint	-
	T,F
Error	-
*Constraint	NULL,1,10,100,500,1000
	1,2,5,10,100
	NULL,0,10,100,1000
*Constraint	NULL,0,10,100,500,1000
*Constraint	NULL,0,10,100,250,1000
	1,5,20,60,100
Timeout	28,40,100,1000
	0,0.1,0.5,20,100
	0,0.1,0.5,30,100
	0,0.1,0.5,10,100
	yes,“20 64 1”,no
	“both”,“minus”,“plus”

Table A.7: MFA Configuration Model. Note since variable types are not explicit we combine Integer and Float to Numeric, for the value of values we stayed consistent with the default. Only the configuration option `rxnUse` was removed in the D2, D3, Random, and covering array experiments due to timeout.

Name in File	Option Name	Type	Values	Default
Add use variables for any drain fluxes	useVarDrainFlux	Boolean	TRUE, FALSE	0
Decompose reversible drain fluxes	decompDrain	Boolean	TRUE, FALSE	0
Decompose reversible reactions	decompRxn	Boolean	TRUE, FALSE	0
Force use variables for all reactions	varForce	Boolean	TRUE, FALSE	1
Make all reactions reversible in MFA	rxnRev	Boolean	TRUE, FALSE	0
Reactions use variables	rxnUse*	Boolean	TRUE, FALSE	0
Recursive MILP solution limit	recMILP	Boolean	TRUE, FALSE	1
calculate reaction knockout sensitivity	rxnKOSen	Boolean	TRUE, FALSE	0
determine minimal required media	minMedia	Boolean	TRUE, FALSE	0
find tight bounds	tightBounds	Boolean	TRUE, FALSE	1
fit phenotype data	phenoFit	Boolean	TRUE, FALSE	0
flux minimization	fluxMin	Boolean	TRUE, FALSE	1
maximize active reactions	maxActiveRxn	Boolean	TRUE, FALSE	0
optimize metabolite production if objective is zero	optimMetabo	Boolean	TRUE, FALSE	0
perform MFA	prefMFA	Boolean	TRUE, FALSE	1
perform single KO experiments	KO	Boolean	TRUE, FALSE	NULL
use database fields	useDataFields	Boolean	TRUE, FALSE	1
use simple variable and constraint names	simpVar	Boolean	TRUE, FALSE	1
write LP file	LPFile	Boolean	TRUE, FALSE	0
write variable key	varKey	Boolean	TRUE, FALSE	1
new fba pipeline	newPipeline	Boolean	TRUE, FALSE	1
Constrain objective to this fraction of the optimal value	conObj	Numeric	0, 0.05, 0.1, 0.15, 0.2	0.1
Default max drain flux	maxDrain	Numeric	0, 250, 500 ,750, 1000	0
Default min drain flux	minDrain	Numeric	0, -250, -500 , -750, -1000	-1000
Max deltaG	maxDeltaG	Numeric	0, 2500, 5000, 7500, 10000	10000
Max flux	maxFlux	Numeric	0, 250, 500 ,750, 1000	1000
Min flux multiplier	minFluxMult	Numeric	0, 1, 2, 3, 4	1
Min flux	minFlux	Numeric	0, -250, -500 , -750, -1000	-1000
deltagslack	deltaGSlack	Numeric	0, 5, 10, 15, 20	10
minimum_target_flux	minTargetFlux	Numeric	0, 0.01, 0.1, 0.2, 0.5	0.1
MFASolver	solver	Enumeration	NULL, CPLEX, GLPK, SCIP, LINDO	NULL

Table A.8: SPLC Configuration Model. Note since variable types are not explicit we combine Integer and Float to Numeric, for the value of values we stayed consistent with the default.

† Removed from Dune in D2, D3, Random, and covering arrays

§ Removed from Hipacc in D2, D3, Random, and covering arrays

Option Name	Type	Values	Default
parallelization	Boolean	TRUE, FALSE	TRUE
limitFeatureSize	Boolean	TRUE, FALSE	FALSE
quadraticFunctionSupport	Boolean	TRUE, FALSE	TRUE
crossValidation	Boolean	TRUE, FALSE	FALSE
learn-logFunction	Boolean	TRUE, FALSE	FALSE
learn-accumulatedLogFunction	Boolean	TRUE, FALSE	FALSE
learn-asymFunction	Boolean	TRUE, FALSE	FALSE
learn-ratioFunction	Boolean	TRUE, FALSE	FALSE
learn-mirrowedFunction	Boolean	TRUE, FALSE	FALSE
withHierarchy§	Boolean	TRUE, FALSE	FALSE
bruteForceCandidates	Boolean	TRUE, FALSE	FALSE
ignoreBadFeatures	Boolean	TRUE, FALSE	FALSE
stopOnLongRound	Boolean	TRUE, FALSE	TRUE
candidateSizePenalty	Boolean	TRUE, FALSE	TRUE
pythonInfluenceAnalysis	Boolean	TRUE, FALSE	FALSE
considerEpsilonTube	Boolean	TRUE, FALSE	FALSE
epsilon	Numeric	0, 0.01, 0.5	0
baggingNumbers	Numeric	0, 3, 100	100
baggingTestDataFraction	Numeric	0, 50, 100	50
abortError	Numeric	0.001, 1, 10	1
numberOfRounds	Numeric	0, 30, 70	70
backwardErrorDelta	Numeric	0.01, 1, 5	1
minImprovementPerRound	Numeric	0.01, 0.1, 0.25	0.1
learnTimeLimit	Numeric	0, 00:30:00, 01:00:00	0
crossValidation_k	Numeric	0, 5, 10	5
scoreMeasure†§	Enumeration	RELERROR, INFLUENCE	RELERROR

APPENDIX B. FRAMEWORK OUTPUT

We display the output of ICO for distance 0 through 3 for all four of our subject examples: BLAST, MFA, SPL Conqueror on Dune, and SPL Conqueror on Hipacc.

Table B.1: Distance 1 and Distance 2 effects on subject BLAST. The functional value is the number of hits. *Effect* is the functional value of the row's configuration minus the functional value of the default. *Add. Eff.* is the type of additive effect. We obfuscate expanded rows with "—".

EQ: The effect is equal to the added effects of the Distance 1 configurations
LT: The effect is less than the added effects of the Distance 1 configurations (new interaction)
GT: The effect is equal to the added effects of the Distance 1 configurations (new interaction)

Starting Configuration						
Configuration				**# Hits**		
Default				4516		
Distance 1						
Config. Opt. 1	**Value 1**			**# Hits**	**Effect**	
ungapped	TRUE			6048	1532	
sum_stats	TRUE			5240	724	
xdrop_gap_final	0, 0.1, 0.5, 10			4519	3	
Distance 2						
Config. Opt. 1	**Value 1**	**Config. Opt. 2**	**Value 2**	**# Hits**	**Effect**	**Add.Eff.**
xdrop_gap_final	0, 0.1, 0.5	xdrop_gap	0, 0.1, 0.5	7994	3478	GT
ungapped	TRUE	sum_stats	TRUE	7980	3464	GT
ungapped	TRUE	xdrop_ungap	0.1, 0.5	7625	3109	GT
sum_stats	TRUE	soft_masking	FALSE	6513	1997	GT
ungapped	TRUE	min_raw_gapped_score	10	6073	1557	GT
ungapped	TRUE	xdrop_ungap	20	6063	1547	GT
ungapped	TRUE	xdrop_gap_final	0, 0.1, 0.5, 10	6048	1532	LT
ungapped	TRUE	—	—	6048	1532	EQ
ungapped	TRUE	min_raw_gapped_score	100, 1000	6044	1528	LT
ungapped	TRUE	xdrop_ungap	100	6032	1516	LT
ungapped	TRUE	max_target_seqs 1		6015	1499	LT
ungapped	TRUE	soft_masking	FALSE	5751	1235	LT
sum_stats	TRUE	xdrop_gap_final	0, 0.1, 0.5, 10	5244	728	GT
sum_stats	TRUE	dust	no	5241	725	GT
sum_stats	TRUE	—	—	5240	724	EQ
sum_stats	TRUE	xdrop_gap	0, 0.1, 0.5	5239	723	LT
sum_stats	TRUE	max_target_seqs	1	5234	718	LT
sum_stats	TRUE	word_size	40	5036	520	LT
sum_stats	TRUE	min_raw_gapped_score	100	4682	166	LT
sum_stats	TRUE	word_size	100	4663	147	LT
xdrop_gap_final	10	xdrop_gap	0, 0.1, 0.5	4531	15	GT
xdrop_gap_final	0, 0.1, 0.5, 10	—	—	4519	3	EQ
xdrop_gap_final	0, 0.1, 0.5, 10	xdrop_gap	30	4518	2	LT

Table B.3: Distance 3 effects on subject BLAST. The functional value is the number of hits. *Effect* is the functional value of the row's configuration minus the functional value of the default. *Add. Eff.* is the type of additive effect.

EQ: The effect is equal to the sum of the effects a subset of the configuration
NEQ: The effect is not equal to the sum of the effects of any subset of the configuration (new interaction)

Starting Configuration								
Configuration						**# Hits**		
Default						4516		
Distance 3								
Config.Opt. 1	**Value 1**	**Config.Opt. 2**	**Value 2**	**Config.Opt. 3**	**Value 2**	**# Hits**	**Effect**	**Add.Eff.**
sum_stats	TRUE	xdrop_gap	0, 0.1, 0.5	xdrop_gap_final	0, 0.1, 0.5	10923	6407	NEQ
ungapped	TRUE	xdrop_gap	0, 0.1, 0.5	xdrop_gap_final	0, 0.1, 0.5	7994	3478	EQ
sum_stats	TRUE	soft_masking	FALSE	xdrop_gap_final	0, 0.1, 0.5, 10	7277	2761	NEQ
sum_stats	TRUE	ungapped	TRUE	soft_masking	FALSE	6513	1997	EQ
sum_stats	TRUE	xdrop_gap	0, 0.1, 0.5	xdrop_gap_final	10	5378	862	NEQ
sum_stats	TRUE	dust	no	xdrop_gap_final	0, 0.1, 0.5, 10	5245	729	NEQ
sum_stats	TRUE	ungapped	TRUE	xdrop_gap_final	0, 0.1, 0.5, 10	5244	728	EQ
sum_stats	TRUE	xdrop_gap	30	xdrop_gap_final	0, 0.1, 0.5, 10	5242	726	EQ
sum_stats	TRUE	ungapped	TRUE	dust	no	5241	725	EQ
sum_stats	TRUE	ungapped	TRUE	xdrop_ungap	0.1, 0.5, 20, 100	5240	724	EQ
sum_stats	TRUE	ungapped	TRUE	min_raw_gapped_score	10	5240	724	EQ
sum_stats	TRUE	ungapped	TRUE	xdrop_gap	0, 0.1, 0.5	5239	723	EQ
sum_stats	TRUE	max_target_seqs	1	xdrop_gap_final	0, 0.1, 0.5, 10	5238	722	NEQ
sum_stats	TRUE	ungapped	TRUE	max_target_seqs	1	5234	718	EQ
sum_stats	TRUE	word_size	40	xdrop_gap_final	0, 0.1, 0.5, 10	5040	524	NEQ
sum_stats	TRUE	ungapped	TRUE	word_size	40	5036	520	EQ
sum_stats	TRUE	min_raw_gapped_score	100	xdrop_gap_final	0, 0.1, 0.5, 10	4686	170	NEQ
sum_stats	TRUE	ungapped	TRUE	min_raw_gapped_score	100	4682	166	EQ
sum_stats	TRUE	word_size	100	xdrop_gap_final	0, 0.1, 0.5, 10	4667	151	NEQ
sum_stats	TRUE	ungapped	TRUE	word_size	100	4663	147	EQ
ungapped	TRUE	xdrop_gap	0, 0.1, 0.5	xdrop_gap_final	10	4531	15	EQ
ungapped	TRUE	min_raw_gapped_score	10, 100	xdrop_gap_final	0, 0.1, 0.5, 10	4519	3	EQ
ungapped	TRUE	xdrop_ungap	0.1, 0.5, 20, 100	xdrop_gap_final	0, 0.1, 0.5,10	4519	3	EQ
ungapped	TRUE	max_target_seqs	1	xdrop_gap_final	0, 0.1, 0.5, 10	4519	3	EQ
ungapped	TRUE	xdrop_gap	30	xdrop_gap_final	0, 0.1, 0.5, 10	4518	2	EQ

Table B.5: ICO output on MFA

Table B.7 Distance 1 and Distance 2 effects on subject MFA. The functional value is the measurement of growth known as the Objective Value (OV). Effect is the functional value of the row's configuration minus the functional value of the default. *Add. Eff.* is the type of additive effect. We obfuscate expanded rows with "—".

EQ: The effect is equal to the added effects of the Distance 1 configuration
LT: The effect is less than the added effects of the Distance 1 configuration
GT: The effect is equal to the added effects of the Distance 1 configuration

Starting Configuration						
Configuration				**OV**		
Default				0.674061		
Distance 1						
Config. Opt. 1	**Value 1**			**OV**	**Effect**	
maxDrain	250,500,750,1000			0.686	0.012	
rxnRev	TRUE			0.676	0.002	
Distance 2						
Config. Opt. 1	**Value 1**	**Config. Opt. 2**	**Value 2**	**OV**	**Effect**	**Add. Eff.**
rxnRev	TRUE	maxDrain	500,750,1000	34.069	33.395	GT
rxnRev	TRUE	maxDrain	250	23.271	22.597	GT
maxDrain	250,500,750,1000	—	—	0.686	0.012	EQ
rxnRev	TRUE	—	—	0.676	0.002	EQ
rxnRev	TRUE	minMedia	TRUE	0.676	0.002	GT

Table B.8 Distance 3 effects on subject MFA. The functional value is the number of hits. Effect is the functional value of the row's configuration minus the functional value of the default. *Add. Eff.* is the type of additive effect.

EQ: The effect is equal to the added effects of the Distance 1 configuration
LT: The effect is less than the added effects of the Distance 1 configuration
GT: The effect is equal to the added effects of the Distance 1 configuration

Distance 3								
Config. Opt. 1	**Value 1**	**Config. Opt. 2**	**Value 2**	**Config. Opt. 3**	**Value 3**	**OV**	**Effect**	**Add. Eff.**
maxDrain	500,750, 1000	rxnRev	TRUE	minMedia	TRUE	27.413	26.739	NEQ
maxDrain	250	rxnRev	TRUE	minMedia	TRUE	20.671	19.997	NEQ

Table B.9: Distance 1 and Distance 2 effects on subject Dune in SPLConqueror. The functional value is the error rate of the regression prediction equation. Effect is the functional value of the row's configuration minus the functional value of the default.

Starting Configuration					
Configuration				**OV**	
Default				0.674061	
Distance 1					
Config. Opt. 1	**Value 1**			**Error**	**Effect**
withHierarchy	TRUE			9.221	-1.747
learn-mirrowedFunction	TRUE			10.269	-0.699
minImprovementPerRound	0.01			10.313	-0.654
ignoreBadFeatures	TRUE			10.960	-0.008
learn-asymFunction	TRUE			10.943	-0.025
learn-logFunction	TRUE			10.955	-0.013
candidateSizePenalty	FALSE			10.910	-0.057
Distance 2					
Config. Opt. 1	**Value 1**	**Config. Opt. 2**	**Value 2**	**Error**	**Effect**
learn-asymFunction	TRUE	withHierarchy	TRUE	9.221	-1.747
quadraticFunctionSupport	FALSE	withHierarchy	TRUE	9.344	-1.624
learn-mirrowedFunction	TRUE	withHierarchy	TRUE	9.446	-1.522
withHierarchy	TRUE	minImprovementPerRound	0.01	9.506	-1.462
withHierarchy	TRUE	minImprovementPerRound	.25	9.570	-1.398
withHierarchy	TRUE	ignoreBadFeatures	TRUE	9.672	-1.295
learn-mirrowedFunction	TRUE	minImprovementPerRound	0.01	9.692	-1.276
learn-logFunction	TRUE	withHierarchy	TRUE	9.762	-1.206
withHierarchy	TRUE	learnTimeLimit	01:00:00	9.802	-1.166
limitFeatureSize	TRUE	withHierarchy	TRUE	9.821	-1.147
withHierarchy	TRUE	candidateSizePenalty	FALSE	9.990	-0.978
withHierarchy	TRUE	abortError	10	9.992	-0.976
withHierarchy	TRUE	learnTimeLimit	00:30:00	10.011	-0.956
withHierarchy	TRUE	numberOfRounds	30	10.119	-0.848
candidateSizePenalty	FALSE	minImprovementPerRound	0.01	10.197	-0.771
learn-mirrowedFunction	TRUE	candidateSizePenalty	FALSE	10.219	-0.749
ignoreBadFeatures	TRUE	minImprovementPerRound	0.01	10.278	-0.690
learn-asymFunction	TRUE	minImprovementPerRound	0.01	10.287	-0.681
learn-logFunction	TRUE	minImprovementPerRound	0.01	10.306	-0.661
limitFeatureSize	TRUE	minImprovementPerRound	0.01	10.385	-0.583
learn-asymFunction	TRUE	learn-mirrowedFunction	TRUE	10.525	-0.442
learn-mirrowedFunction	TRUE	ignoreBadFeatures	TRUE	10.708	-0.260
learn-asymFunction	TRUE	candidateSizePenalty	FALSE	10.878	-0.089
learn-logFunction	TRUE	candidateSizePenalty	FALSE	10.892	-0.076
learn-logFunction	TRUE	learn-asymFunction	TRUE	10.943	-0.025
learn-asymFunction	TRUE	ignoreBadFeatures	TRUE	10.943	-0.025
learn-logFunction	TRUE	learn-mirrowedFunction	TRUE	10.955	-0.0126
learn-logFunction	TRUE	ignoreBadFeatures	TRUE	10.955	-0.0126

Table B.10: Distance 3 effects on subject Dune in SPLConqueror. The functional value is the error rate on the regression question. *Effect* is the functional value of the row's configuration minus the functional value of the default. Due to size of table configuration option names were shortened.

Starting Configuration							
Configuration						**Err.Rate**	
Default						10.977507	
Distance 3							
Config. Opt. 1	**Val. 1**	**Config. Opt. 2**	**Val. 2**	**Config. Opt. 3**	**Val. 3**	**Err.Rate**	**Effect**
quadFuncSup	F	learn-mirrowFunc	T	withHier	T	8.934399	-2.033099
withHier	T	learn-mirrowFunc	T	minImprPerRound	0.01	8.979476	-1.988022
learn-logFunc	T	learn-mirrowFunc	T	withHierar	T	9.143074	-1.824424
withHier	T	ignoreBadFeat	T	minImprPerRound	0.01	9.197190	-1.770308
quadFuncSup	F	withHierar	T	ignoreBadFeat	T	9.204249	-1.763249
.....							
learn-asymFunc	T	ignoreBadFeat	T	learn-mirrowFunc	T	10.94296	-0.024538
learn-asymFunc	T	ignoreBadFeat	T	learn-logFunc	T	10.94296	-0.024538
learn-asymFunc	T	ignoreBadFeat	T	candSizePenalty	F	10.965887	-0.001611

Table B.11: Distance 1 and Distance 2 effects on subject Hipacc in SPLConqueror. The functional value is the error rate of the regression prediction equation. Effect is the functional value of the row's configuration minus the functional value of the default. *Add. Eff.* is the type of additive effect. We obfuscate expanded rows with "—".

EQ: The effect is equal to the added effects of the Distance 1 configuration
LT: The effect is less than the added effects of the Distance 1 configuration
GT: The effect is equal to the added effects of the Distance 1 configuration

Starting Configuration						
Configuration				**Error**		
Default				10.694		
Distance 1						
Config. Opt. 1	**Value 1**			**Error**	**Effect**	
candidateSizePenalty	FALSE			9.688	-1.006	
learn-asymFunction	TRUE			9.874	-0.821	
learn-mirrowedFunction	TRUE			10.145	-0.550	
ignoreBadFeatures	TRUE			10.318	-0.376	
learn-logFunction	TRUE			10.582	-0.112	
Distance 2						
Config. Opt. 1	**Value 1**	**Config. Opt. 2**	**Value 2**	**Error**	**Effect**	**Add.Eff.**
candidateSizePenalty	FALSE	learn-asymFunction	TRUE	8.495	-2.200	LT
candidateSizePenalty	FALSE	learn-logFunction	TRUE	9.189	-1.505	LT
candidateSizePenalty	FALSE	learn-mirrowedFunction	TRUE	9.221	-1.473	GT
candidateSizePenalty	FALSE	—	—	9.688	-1.006	EQ
learn-asymFunction	TRUE	—	9.874	-0.82	1	EQ
learn-mirrowedFunction	TRUE	learn-ratioFunction	TRUE	9.950	-0.745	LT
learn-asymFunction	TRUE	abortError	10	9.960	-0.734	GT
candidateSizePenalty	FALSE	abortError	10	9.996	-0.698	GT
learn-mirrowedFunction	TRUE	—	—	10.145	-0.550	EQ
learn-mirrowedFunction	TRUE	ignoreBadFeatures	TRUE	10.240	-0.454	GT
ignoreBadFeatures	TRUE	—	—	10.318	-0.376	EQ
learn-asymFunction	TRUE	learnTimeLimit	1:00:00	10.431	-0.263	GT
learn-logFunction	TRUE	—	—	10.582	-0.112	EQ

Table B.13: Distance 3 effects on subject Hipacc in SPLConqueror. The functional value is the error rate on the regression question. *Effect* is the functional value of the row's configuration minus the functional value of the default. Due to size of table configuration option names were shortened.

Starting Configuration							
Configuration						**Err.Rate**	
Default						10.694	
Distance 3							
Config. Opt. 1	**Val. 1**	**Config. Opt. 2**	**Val. 2**	**Config. Opt. 3**	**Val. 3**	**Err.Rate**	**Effect**
candSizePenalty	F	learn-ratioFunc	T	learn-mirrowFunc	T	8.465	-2.230
candSizePenalty	F	learn-asymFunc	T	ignoreBadFeat	T	8.495	-2.200
candSizePenalty	F	learn-asymFunc	T	learn-mirrowFunc	T	8.495	-2.200
candSizePenalty	F	learn-asymFunc	T	learn-logFunc	T	8.495	-2.200
candSizePenalty	F	learn-logFunc	T	ignoreBadFeat	T	9.189	-1.505
candSizePenalty	F	learn-logFunc	T	learn-mirrowFunc	T	9.189	-1.505
candSizePenalty	F	learn-mirrowFunc	T	ignoreBadFeat	T	9.221	-1.473
candSizePenalty	F	learn-asymFunc	T	abortError	10	9.864	-0.830
candSizePenalty	F	learn-mirrowFunc	T	abortError	10	9.864	-0.830
learn-asymFunc	T	learn-mirrowFunc	T	ignoreBadFeat	T	9.874	-0.821
learn-asymFunc	T	learn-ratioFunc	T	learn-mirrowFunc	T	9.874	-0.821
learn-asymFunc	T	learn-logFunc	T	ignoreBadFeat	T	9.874	-0.821
learn-asymFunc	T	learn-logFunc	T	learn-mirrowFunc	T	9.874	-0.821
ignoreBadFeat	T	learn-ratioFunc	T	learn-mirrowFunc	T	9.950	-0.745
ignoreBadFeat	T	learn-asymFunc	T	abortError	10	9.960	-0.734
ignoreBadFeat	T	learn-asymFunc	T	learnTimeLimit	1:00:00	9.960	-0.734
abortError	10	learn-asymFunc	T	learn-mirrowFunc	T	9.960	-0.734
abortError	10	learn-asymFunc	T	learn-logFunc	T	9.960	-0.734
abortError	10	candSizePenalty	F	learn-logFunc	T	9.965	-0.729
abortError	10	candSizePenalty	F	ignoreBadFeat	T	9.996	-0.698
learnTimeLimit	1:00:00	ignoreBadFeat	T	learn-asymFunc	T	10.201	-0.493
learnTimeLimit	1:00:00	ignoreBadFeat	T	learn-mirrowFunc	T	10.425	-0.269
learnTimeLimit	1:00:00	ignoreBadFeat	T	candSizePenalty	F	10.577	-0.117
learn-logFunc	T	ignoreBadFeat	T	learn-mirrowFunc	T	10.582	-0.112
learn-logFunc	T	learn-ratioFunc	T	learn-mirrowFunc	T	10.582	-0.112
learn-asymFunc	T	learnTimeLimit	1:00:00	candSizePenalty	F	10.648	-0.046
learn-asymFunc	T	learnTimeLimit	1:00:00	learn-mirrowFunc	T	10.662	-0.032
learn-asymFunc	T	learnTimeLimit	1:00:00	learn-logFunc	T	10.662	-0.032

APPENDIX C. SPL CONQUEROR TOOL

Run Summaries

SPL Conqueror regression results The RAE (relative absolute error) is reported by SPL Conqueror and evaluated on the training data. Additional analysis was performed outside of SPL Conqueror to evaluate the MSE (mean squared error), MAE (mean absolute error), and MDAE (median absolute error) on three test sets: Random, D1, and D2.

Recall that for BLAST and MFA we have different transformations, we choose the best and report which transformation

Table C.1: SPL Conqueror regression results evaluated against Random (MSE, MAE, MDAE)

Subject	Test	Trans.	RAE	Random			
				MSE	MAE	MAE%	MDAE
BLAST	2CA5	absolute	0.084	1.806×10^{10}	3.65×10^{4}	1098.331	1.764×10^{5}
	3CA1	P1	33.184	7.475×10^{6}	1549.284	46.595	265.689
	Rand	P1	75.005	2.141×10^{7}	3325.739	100.023	3017.000
	DAll	P1	99.234	7.816×10^{23}	1.943×10^{11}	5.845×10^{9}	3.549×10^{3}
MFA	2CA3	P1	0.00	17.355	0.758	102.685	0.674
	3CA5	absolute	1.204	15.15	1.148	155.46	2.133
	Rand	absolute	1.315	16.349	1.315	178.178	0.84
	D0-3	absolute	0.258	16.658	0.977	132.346	0.045
Dune	2CA4	Reg	0.078	65.877	6.284	35.91	15.26
	3CA4	Reg	0.711	35.729	2.888	16.505	60.275
	Rand	Reg	1.152	0.056	0.121	0.69	70.046
	DAll	Reg	4.174	29.731	2.944	16.824	0.177
Hipacc	2CA1	Reg	0.00	575.181	19.09	84.975	5.62
	3CA2	Reg	0.977	10.374	2.61	11.617	3.867
	Rand	Reg	4.26	1.631	0.705	3.138	0.722
	DAll	Reg	1.911	62.798	4.547	20.241	0.045

Table C.2: SPL Conqueror regression results evaluated against D1 (MSE, MAE, MDAE)

Subject	Test	Trans.	RAE	D1			
				MSE	MAE	MAE%	MDAE
BLAST	2CA5	absolute	0.084	2.676×10^{10}	1.519×10^{5}	3488.201	1.764×10^{5}
	3CA1	P1	33.184	2.387×10^{5}	340.115	7.811	265.689
	Rand	P1	75.005	1.971×10^{7}	4355.514	100.023	4517.000
	DAll	P1	99.234	1.971×10^{7}	4355.529	100.023	4517.000
MFA	2CA3	P1	0.00	0.43	0.637	100	0.674
	3CA5	absolute	1.204	4.556	2.107	330.737	2.133
	Rand	absolute	1.315	0.768	0.86	135.028	0.84
	D0-3	absolute	0.258	0.025	0.074	11.566	0.045
Dune	2CA4	Reg	0.078	224.221	14.787	131.336	15.26
	3CA4	Reg	0.711	3920.724	61.026	542.025	60.275
	Rand	Reg	1.152	4975.091	69.232	614.904	70.046
	DAll	Reg	4.174	0.513	0.292	2.589	0.177
Hipacc	2CA1	Reg	0.00	89.168	7.361	65.325	5.62
	3CA2	Reg	0.977	13.482	3.53	31.323	3.867
	Rand	Reg	4.26	0.646	0.763	6.773	0.722
	DAll	Reg	1.911	0.031	0.099	0.881	0.045

Confusion Matrices

We display the confusion matrices for all four subject on four training data samples: random, D0-3, best 2-way covering array, and best 3-way covering array.

Table C.3: SPL Conqueror regression results evaluated against D2 (MSE, MAE, MDAE)

Subject	Test	Trans.	RAE	D2			
				MSE	MAE	MAE%	MDAE
BLAST	2CA5	absolute	0.084	2.194×10^{10}	1.214×10^{5}	2924.73	1.764×10^{5}
	3CA1	P1	33.184	5.985×10^{5}	470.984	11.349	265.689
	Rand	P1	75.005	1.904×10^{7}	4151.019	100.023	4517.000
	DAll	P1	99.234	1.904×10^{7}	4151.050	100.024	4517.000
MFA	2CA3	P1	0.00	1.462	0.632	99.972	0.674
	3CA5	absolute	1.204	5.429	2.106	333.042	2.133
	Rand	absolute	1.315	1.787	0.91	143.911	0.84
	D0-3	absolute	0.258	1.058	0.134	21.263	0.045
Dune	2CA4	Reg	0.078	215.344	14.308	123.68	15.26
	3CA4	Reg	0.711	4222.414	61.747	533.762	60.275
	Rand	Reg	1.152	5128.506	68.6	593.002	70.046
	DAll	Reg	4.174	0.981	0.4	3.461	0.177
Hipacc	2CA1	Reg	0.00	147.724	9.043	76.032	5.62
	3CA2	Reg	0.977	12.059	3.214	27.021	3.867
	Rand	Reg	4.26	0.795	0.805	6.77	0.722
	DAll	Reg	1.911	0.187	0.174	1.459	0.045

Regression Models

This section contains a sample of the regression models. For each of the four subjects (BLAST, MFA, SPL Conqueror on Dune, SPL Conqueror on Hipacc) we display four regression models. The four models are trained on the best 2-way covering array (according the to RAE), the best 3-way covering array (according to the RAE), Random, and D0-3. We report on the number of configuration used to train the model, the complexity of the model, the RAE, MAE, and MAE% of each model, and display the raw output of SPL Conqueror's regression model.

```
7999.44454204033 * root + -7.16554469601431 * word_size +
   -4427.93259309406 * min_raw_gapped_score_1000 + 4.88746592948918 *
   word_size * min_raw_gapped_score_1000 + 2608.03788876937 * xdrop_gap
    + 149.544630399118 * sum_stats + -4193.9018136208 * strand_minus +
   -115.080739237693 * xdrop_gap * xdrop_gap + -2478.00175331137 *
```

Table C.4: SPL-Conqueror Regression Results - BLAST

Test (# configs)	Transformation	Round	Elapsed Seconds	Model Complexity	Learning Error
2CA1 (58)	**absolute**	**49**	**786.983**	**118**	**0.687**
	P1	4	1.619	31	75.862
	T100P1	4	1.526	31	75.862
2CA2 (55)	**absolute**	**50**	**820.031**	**101**	**0.903**
	P1	5	2.66	31	78.182
	T100P1	5	2.365	31	78.182
2CA3 (62)	absolute	21	50.207	46	258.405
	P1	**4**	**1.464**	**12**	**77.411**
	T100P1	4	1.173	12	4454.858
2CA4 (58)	absolute	15	24.169	31	755.301
	P1	**4**	**0.929**	**10**	**109.038**
	T100P1	4	0.939	10	7838.488
2CA5 (55)	**absolute**	**54**	**1027.351**	**103**	**0.084**
	P1	4	0.897	10	191.68
	T100P1	4	1.085	10	16589.278
3CA1 (387)	absolute	14	65.707	45	795.184
	P1	**4**	**2.142**	**12**	**33.184**
	T100P1	4	2.254	12	669.182
3CA2 (390)	absolute	4	2.001	7	992.027
	P1	**4**	**2.271**	**11**	**86.498**
	T100P1	4	2.055	11	5921.15
3CA3 (401)	absolute	4	1.912	7	1088.445
	P1	**4**	**2.263**	**10**	**120.977**
	T100P1	4	2.135	10	9167.901
3CA4 (393)	absolute	49	2384.86	155	632.758
	P1	**4**	**2.216**	**11**	**36.494**
	T100P1	4	2.069	11	1031.73
3CA5 (411)	absolute	19	163.922	48	624.715
	P1	**4**	**2.323**	**10**	**104.071**
	T100P1	4	2.343	10	7547.368
Rand (99995)	absolute	4	956.171	8	1256.329
	P1	**4**	**1632.434**	**31**	**75.005**
	T100P1	4	1632.47	31	75.005
D0-3 (874)	absolute	39	1956.88	96	159.211
	P1	**6**	**9.513**	**41**	**99.234**
	T100P1	5	6.319	30	99.657

Table C.5: SPL-Conqueror Regression Results - MFA

Test (# configs)	Transformation	Round	Elapsed Seconds	Model Complexity	Learning Error
2CA1 (38)	absolute	1	0.048	2	0.031
	P1	2	0.086	5	0.494
	T100P1	**4**	**0.286**	**23**	**0.00**
2CA2 (38)	absolute	1	0.048	3	0.061
	P1	**3**	**0.213**	**13**	**0.013**
	T100P1	2	0.103	5	325.997
2CA3 (38)	absolute	1	0.051	2	0.031
	P1	**3**	**0.158**	**7**	**0.00**
	T100P1	3	0.143	7	0.00
2CA4 (38)	absolute	2	0.08	4	0.723
	P1	9	2.064	21	0.733
	T100P1	**13**	**8.716**	**43**	**0.00**
2CA5 (38)	absolute	3	0.179	6	2.313
	P1	4	0.253	9	109.604
	T100P1	**8**	**1.599**	**24**	**0.00**
3CA1 (264)	**absolute**	**4**	**1.288**	**15**	**5.401**
	P1	9	7.317	42	40.849
	T100P1	10	50.155	514	98.462
3CA2 (267)	**absolute**	**1**	**0.122**	**2**	**1.481**
	P1	15	29.537	71	3.042
	T100P1	47	1128.371	300	613.932
3CA3 (266)	**absolute**	**2**	**0.346**	**4**	**1.281**
	P1	11	13.625	62	3.844
	T100P1	10	36.95	400	97.859
3CA4 (260)	**absolute**	**4**	**1.216**	**11**	**5.382**
	P1	12	13.702	44	48.915
	T100P1	15	25.981	59	4073.641
3CA5 (265)	**absolute**	**2**	**0.373**	**4**	**1.204**
	P1	15	48.902	105	24.142
	T100P1	53	1503.504	239	606.599
Rand (97795)	**absolute**	**1**	**52.102**	**2**	**1.315**
	P1	4	767.174	10	53.426
	T100P1	14	16834.357	62	177.521
D0-3 (1723)	**absolute**	**1**	**0.584**	**2**	**0.258**
	P1	4	12.09	8	10.224
	T100P1	9	88.739	30	97.172

Table C.6: SPL-Conqueror Regression Results - Dune

Test (# configs)	Transformation	Round	Elapsed Seconds	Model Complexity	Learning Error
2CA1 (14)	Reg	9	1.336	15	0.619
	Absolute	4	0.156	7	0.609
	P1	**9**	**1.145**	**15**	**0.57**
	T100P1	9	1.241	15	0.618
2CA2 (13)	Reg	5	0.22	7	0.348
	Absolute	1	0.04	2	0.947
	P1	**5**	**0.223**	**7**	**0.325**
	T100P1	5	0.272	7	0.348
2CA3 (13)	Reg	9	1.023	13	0.612
	Absolute	5	0.258	8	0.773
	P1	**9**	**1.132**	**13**	**0.568**
	T100P1	9	1.229	13	0.611
2CA4 (13)	**Reg**	**6**	**0.373**	**12**	**0.078**
	Absolute	2	0.058	5	0.862
	P1	5	0.269	11	0.959
	T100P1	6	0.419	12	0.078
2CA5 (13)	Reg	4	0.179	10	0.888
	Absolute	**1**	**0.04**	**2**	**0.821**
	P1	4	0.17	10	0.832
	T100P1	4	0.175	10	0.887
3CA1 (52)	Reg	17	17.673	35	1.789
	Absolute	**3**	**0.196**	**9**	**0.872**
	P1	17	16.71	35	1.666
	T100P1	17	16.932	35	1.787
3CA2 (53)	Reg	15	16.285	37	2.467
	Absolute	**4**	**0.394**	**8**	**2.137**
	P1	15	15.668	37	2.31
	T100P1	15	15.202	37	2.465
3CA3 (53)	Reg	4	0.401	19	6.999
	Absolute	**3**	**0.194**	**9**	**0.912**
	P1	4	0.373	19	6.479
	T100P1	4	0.334	14	6.994
3CA4 (53)	**Reg**	**14**	**10.687**	**38**	**0.711**
	Absolute	3	0.214	9	0.886
	P1	13	9.309	34	0.934
	T100P1	14	12.502	38	0.71
3CA5 (53)	Reg	12	6.655	31	3.161
	Absolute	**4**	**0.353**	**19**	**0.996**
	P1	11	5.555	28	3.131
	T100P1	12	7.019	31	3.159
Rand (984)	Reg	16	239.348	59	1.152
	Absolute	4	4.884	14	1.149
	P1	**16**	**238.259**	**59**	**1.042**
	T100P1	16	240.136	59	1.151
D0-3 (668)	**Reg**	**4**	**1.881**	**6**	**4.174**

Table C.7: SPL-Conqueror Regression Results - Hipacc

Test (# configs)	Transformation	Round	Elapsed Seconds	Model Complexity	Learning Error
2CA1 (13)	**Reg**	**12**	**2.708**	**22**	**0.00**
	Absolute	4	0.144	7	0.891
	P1	11	2.012	21	0.279
	T100P1	12	2.666	22	0.00
2CA2 (14)	Reg	8	0.834	12	0.731
	Absolute	5	0.249	8	0.835
	P1	8	0.851	12	0.688
	T100P1	**8**	**0.806**	**12**	**0.731**
2CA3 (14)	Reg	10	1.465	13	0.268
	Absolute	5	0.229	9	0.817
	P1	**10**	**1.598**	**13**	**0.256**
	T100P1	10	1.459	13	0.268
2CA4 (12)	Reg	7	0.537	11	0.975
	Absolute	**5**	**0.233**	**9**	**0.448**
	P1	7	0.553	11	0.902
	T100P1	7	0.575	11	0.974
2CA5 (14)	Reg	9	1.196	16	0.946
	Absolute	5	0.252	8	0.959
	P1	**9**	**1.269**	**16**	**0.889**
	T100P1	9	1.203	16	0.945
3CA1 (50)	Reg	23	38.68	46	1.78
	Absolute	**4**	**0.358**	**14**	**1.658**
	P1	15	13.78	31	4.273
	T100P1	23	39.722	46	1.779
3CA2 (49)	Reg	28	56.704	51	0.977
	Absolute	4	0.339	13	1.931
	P1	**28**	**57.106**	**51**	**0.922**
	T100P1	28	55.735	51	0.977
3CA3 (48)	Reg	11	4.434	28	3.133
	Absolute	**6**	**0.993**	**19**	**1.112**
	P1	11	4.884	28	2.908
	T100P1	11	4.221	28	3.131
3CA4 (49)	Reg	27	54.872	59	2.839
	Absolute	**5**	**0.586**	**31**	**2.033**
	P1	27	56.96	59	2.682
	T100P1	27	53.69	59	2.837
3CA5 (49)	Reg	22	33.589	48	2.42
	Absolute	**4**	**0.32**	**13**	**1.935**
	P1	23	42.675	49	2.165
	T100P1	22	33.918	48	2.418
Rand (991)	Reg	14	168.855	46	4.26
	Absolute	**7**	**20.837**	**22**	**0.957**
	P1	10	64.18	48	5.196
	T100P1	10	61.466	48	5.662
D0-3 (548)	**Reg**	**8**	**13.561**	**11**	**1.911**

Table C.8: Confusion Matrices on Distance 1 configurations. Results displayed evaluated on random, D0-3 (distance 0 through 3), best 2-way covering array, and best 3-way covering array.

Table C.10 BLAST - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	1	1
	EQ	6	58	4
	LT	0	0	0

Table C.11 BLAST - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	1	2
	EQ	6	58	3
	LT	0	0	0

Table C.12 BLAST - 2CA5

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	4	11	2
	EQ	0	36	0
	LT	2	12	3

Table C.13 BLAST - 3CA1

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	6	57	3
	LT	0	2	2

Table C.14 MFA - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	5	80	4
	LT	0	0	1

Table C.15 MFA - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	5	80	4
	LT	0	0	1

Table C.16 MFA - 2CA3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	5	79	5
	LT	0	1	0

Table C.17 MFA - 3CA5

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	5	80	3
	LT	0	0	2

Table C.18 Dune - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	2	0
	EQ	2	44	7
	LT	2	3	0

Table C.19 Dune - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	3	0	0
	EQ	1	49	5
	LT	0	0	2

Table C.20 Dune - 2CA4

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	1	0
	EQ	4	45	7
	LT	0	3	0

Table C.21 Dune - 3CA4

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	2	4	0
	EQ	1	44	6
	LT	1	1	1

Table C.22 Hipacc - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	4	1	0
	EQ	2	46	4
	LT	0	0	1

Table C.23 Hipacc - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	4	1	0
	EQ	2	46	3
	LT	0	0	2

Table C.24 Hipacc - 2CA1

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	2	5	0
	EQ	3	38	5
	LT	1	4	0

Table C.25 Hipacc - 3CA2

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	3	1	2
	EQ	2	39	1
	LT	1	7	2

Table C.26: Confusion Matrices on Distance 2 configurations. Results displayed evaluated on random, D0-3 (distance 0 through 3), best 2-way covering array, and best 3-way covering array.

Table C.28 BLAST - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	6	43	57
	EQ	272	919	198
	LT	0	0	0

Table C.29 BLAST - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	7	43	106
	EQ	271	919	149
	LT	0	0	0

Table C.30 BLAST - 2CA5

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	194	303	134
	EQ	0	287	0
	LT	84	372	121

Table C.31 BLAST - 3CA1

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	265	876	146
	LT	13	86	109

Table C.32 MFA - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	394	2987	336
	LT	0	0	88

Table C.33 MFA - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	394	2987	336
	LT	0	0	88

Table C.34 MFA - 2CA3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	1	0
	EQ	389	2912	420
	LT	5	74	4

Table C.35 MFA - 3CA5

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	0	0	0
	EQ	394	2987	249
	LT	0	0	175

Table C.36 Dune - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	6	89	14
	EQ	98	882	313
	LT	116	129	21

Table C.37 Dune - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	166	0	0
	EQ	52	1100	242
	LT	2	0	106

Table C.38 Dune - 2CA4

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	4	44	7
	EQ	204	924	320
	LT	12	132	21

Table C.39 Dune - 3CA4

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	112	170	28
	EQ	49	882	264
	LT	59	48	56

Table C.40 Hipacc - Random

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	210	43	5
	EQ	91	921	180
	LT	2	0	47

Table C.41 Hipacc - D0-3

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	210	43	3
	EQ	90	921	133
	LT	3	0	96

Table C.42 Hipacc - 2CA1

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	118	196	25
	EQ	120	615	187
	LT	65	153	20

Table C.43 Hipacc - 3CA2

		Actual		
		GT	**EQ**	**LT**
Predicted	**GT**	159	40	88
	EQ	72	651	37
	LT	72	273	107

Table C.44: BLAST Regression Model - Best 2-way CA

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
2CA5	absolute	55	103	0.084	1098.331	2215.5

```
xdrop_gap * strand_both + −2654.77723859989 *
min_raw_gapped_score_1000 * html + 84.0156441510883 * xdrop_gap *
xdrop_gap * strand_both + −0.13487148036566 * strand_minus *
max_target_seqs_100 + −0.696655081454412 * word_size * xdrop_gap +
−508.534510301995 * sum_stats * num_descriptions_1000 +
6175.91538610609 * sum_stats * num_descriptions_1000 *
min_raw_gapped_score_NULL + 1748.83713069842 * strand_minus *
soft_masking + −2281.41094372025 * sum_stats * num_alignments_1000 +
 −1965.3501584382 * strand_minus * min_raw_gapped_score_NULL +
−1986.47809688529 * strand_minus * max_target_seqs_10 +
1238.8134386098 * strand_minus * num_descriptions_1000 +
−1853.02150905254 * max_target_seqs_1 + 0.221695063423922 *
xdrop_gap * xdrop_gap * soft_masking + 4348.02234186714 * sum_stats
* num_alignments_1000 * strand_plus + 0.000678443277598907 *
word_size * xdrop_gap * word_size + −925.843131932729 *
max_target_seqs_1 * dust_no + −286.829679286521 * sum_stats *
max_target_seqs_500 + −3.68420552241089 * xdrop_gap * xdrop_gap *
soft_masking * max_target_seqs_100 + −16.8832383304336 *
strand_minus * num_descriptions_500 + 3.73206378807875 * word_size *
 strand_minus + 146.940168905158 * strand_minus * num_threads +
3949.3182164807 * strand_minus * num_alignments_1000 +
0.878374726876579 * xdrop_gap * max_target_seqs_500 +
0.542831143806765 * xdrop_gap * num_descriptions_10
```

```
+ 643.376477357997 * num_descriptions_100 + 508.174391010448 *
   max_target_seqs_1 * num_threads + -839.741084680756 *
   num_alignments_1000 + 1263.82627752009 * min_raw_gapped_score_1000 *
    max_target_seqs_1000 + 1319.66387878057 * sum_stats * ungapped +
   -59.8169483744221 * xdrop_gap * strand_both * xdrop_gap_final +
   -1454.5918709528 * ungapped + -550.891593751467 *
   num_descriptions_100 * min_raw_gapped_score_NULL + -5061.27620603111
    * sum_stats * max_target_seqs_100 + 0.190072240129932 * word_size *
    max_target_seqs_10 + -2106.9106456038 * strand_plus +
   41.5270033536828 * word_size * max_target_seqs_100 +
   493.601506674129 * strand_plus * num_descriptions_1000 +
   -1479.36391796064 * max_target_seqs_100 + 660.873339042128 *
   max_target_seqs_10 + -872.556228821963 * num_alignments_10 +
   -1702.12494000978 * num_alignments_10 * dust_no + 0.890675434164375
   * xdrop_gap * xdrop_gap * xdrop_gap + -9.86267687854535 *
   num_threads * num_threads + 127.444556353644 * lcase_masking +
   136.015548731033 * num_alignments_250 + -0.00583968563669321 *
   xdrop_ungap * xdrop_ungap
```

Table C.45: BLAST Regression Model - Best 3-way CA

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
3CA1	P1	387	12	33.184	46.595	733.159

```
4254.64597692912 * root + -0.0042533272413316 * word_size * word_size +
    -4253.64597687382 * min_raw_gapped_score_1000 + 0.00425332724122571
    * word_size * word_size * min_raw_gapped_score_1000 +
   2.19382036230749E-16 * min_raw_gapped_score_1000 * word_size *
   word_size * xdrop_gap * xdrop_gap
```

Table C.46: BLAST Regression Model - Random

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
Rand	P1	99995	31	75.005	100.023	3017

```
0 * root + 0 * word_size * word_size + 1.00010259580583E-84 * word_size
    * word_size * word_size * word_size + 1.00000000922686E-72 *
   word_size * word_size * word_size * word_size * word_size *
   word_size * word_size * word_size + 9.99999999279108E-49 * word_size
    * word_size * word_size * word_size * word_size * word_size *
   word_size * word_size * word_size * word_size * word_size *
   word_size * word_size * word_size * word_size * word_size
```

Table C.47: BLAST Regression Model - D0-3

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
D0-3	P1	874	41	99.234	5844638000	3549

```
0 * root + 1.26627416846957E-17 * min_raw_gapped_score_1000 +
  2.64355869785184E-20 * word_size * word_size + 2.07751336514438E-17
  * word_size * word_size * min_raw_gapped_score_1000 * word_size *
  word_size + 4.25655892024368E-12 * word_size * word_size *
  min_raw_gapped_score_1000 * word_size * word_size * word_size *
  word_size * min_raw_gapped_score_1000 * word_size * word_size +
  -2.31081899342302E-12 * word_size * word_size *
  min_raw_gapped_score_1000 * word_size * word_size * word_size *
  word_size * min_raw_gapped_score_1000 * word_size * word_size *
  max_target_seqs_500 + -1.94573992681965E-12 * word_size * word_size
  * min_raw_gapped_score_1000 * word_size * word_size * word_size *
  word_size * min_raw_gapped_score_1000 * word_size * word_size *
  num_alignments_250
```

Table C.48: MFA Regression Model - Best 2-way CA

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
2CA3	P1	38	7	0	102.685	0

```
1 * root + -1.11022302462516E-16 * solver_LINDO + 1.11022302462516E-16
  * solver_LINDO * phenoFit + 0.686021999999999 * solver_LINDO *
  phenoFit * newPipeline
```

Table C.49: MFA Regression Model - Best 3-way CA

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
3CA5	absolute	265	4	1.204	155.46	0

```
−2.68776866041431E−15 * root + 1.17589878893126E−15 * newPipeline +
   2.80687163768117 * newPipeline * prefMFA
```

Table C.50: MFA Regression Model - Random

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
Rand	absolute	97795	2	1.315	178.178	0

```
−1.78568872905004E−13 * root + 1.51433789862822 * prefMFA
```

Table C.51: MFA Regression Model - D0-3

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
D0-3	absolute	1723	2	0.258	132.346	0

```
−1.7858255051172E−14 * root + 0.718846176788936 * prefMFA
```

Table C.52: Dune Regression Model - Best 2-way CA

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
2CA4	Reg	13	12	0.078	35.91	6.68

```
24.0881143005678 * root + -45.0382192878245 * epsilon * epsilon +
   -9.67596564070878 * learnasymFunction + 1.93440381853796 *
   learnasymFunction * backwardErrorDelta + -3.11140147839701 *
   learnasymFunction * learnlogFunction + 1.40844842776291 * epsilon *
   epsilon * abortError + 0.4279539609084 * crossValidation
```

Table C.53: Dune Regression Model - Best 3-way CA

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
3CA4	Reg	53	38	0.711	16.505	0.342

```
16.3509547816622 * root + 8.17867345594968 * crossValidation +
   -0.210452054961813 * crossValidation * epsilon * epsilon +
   -66.9929967832904 * crossValidation * epsilon * epsilon *
   considerEpsilonTube + -1.09096807921687 * numberOfRounds +
   0.0144590956951152 * numberOfRounds * numberOfRounds +
   -0.00760709433245754 * learnasymFunction + 0.0254358483888923 *
   numberOfRounds * bruteForceCandidates + 1.07651617053595 *
   numberOfRounds * crossValidation + -0.0146592356149886 *
   numberOfRounds * numberOfRounds * crossValidation + 7.85436047660849
    * crossValidation * epsilon * epsilon * learnTimeLimit_0 +
   8.31725979953293 * crossValidation * epsilon * epsilon *
   considerEpsilonTube * stopOnLongRound + -0.0223363414678953 *
   numberOfRounds * epsilon + 0.000695092644248127 * numberOfRounds *
```

```
epsilon * baggingNumbers + 0.0007701797522683 48 * numberOfRounds *
numberOfRounds * crossValidation * minImprovementPerRound
```

Table C.54: Dune Regression Model - Random

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
Rand	Reg	884	59	1.152	0.69	0.031

```
16.3447552142791 * root + 8.17095255244418 * crossValidation +
    -0.0120440131561548 * crossValidation * epsilon * epsilon +
    -58.8806264237197 * crossValidation * epsilon * epsilon *
    considerEpsilonTube + -0.285874094178229 * numberOfRounds +
    0.00286179040480351 * numberOfRounds * numberOfRounds +
    0.000978785716900799 * bruteForceCandidates + 0.284675444520781 *
    numberOfRounds * crossValidation + -0.00285036308418422 *
    numberOfRounds * numberOfRounds * crossValidation +
    0.000573501164464196 * bruteForceCandidates * crossValidation +
    0.107289418557839 * numberOfRounds * bruteForceCandidates +
    -0.00110111237315538 * numberOfRounds * numberOfRounds *
    bruteForceCandidates + -0.10749648427177 * numberOfRounds *
    bruteForceCandidates * crossValidation + -8.16205800145555 *
    numberOfRounds * crossValidation * epsilon * epsilon *
    considerEpsilonTube * crossValidation * epsilon * epsilon *
    crossValidation * epsilon * epsilon + 0.00108027887482631 *
    numberOfRounds * numberOfRounds * numberOfRounds * crossValidation *
     epsilon * epsilon * considerEpsilonTube * crossValidation * epsilon
     * epsilon * crossValidation * epsilon * epsilon +
    0.00108907042039402 * numberOfRounds * numberOfRounds *
```

```
bruteForceCandidates * crossValidation + 0.00331360027615267 *
numberOfRounds * bruteForceCandidates * withHierarchy
```

Table C.55: Dune Regression Model - D0-3

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
C0-3	Reg	668	6	4.174	16.824	1.067

```
10.6526955948301 * root + 13.3569309856507 * crossValidation +
1.52729834928707 * bruteForceCandidates + 4.91333645394075 *
minImprovementPerRound + -12.2888021626569 * minImprovementPerRound
* withHierarchy
```

Table C.56: Hipacc Regression Model - Best 2-way CA

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
2CA1	Reg	13	22	0	84.975	16.463

```
37.7151617420574 * root + 15.5637087998535 * considerEpsilonTube +
0.504014604607934 * numberOfRounds + -118.310329481227 * epsilon *
epsilon + 223.947910802904 * minImprovementPerRound *
minImprovementPerRound + -1.66371773043095 * abortError * abortError
+ -4.53431436321219 * parallelization + -7.35284984084749 *
parallelization * bruteForceCandidates + -8.31328295685017 *
bruteForceCandidates + -0.0265707004638089 * abortError * abortError
* epsilon + -0.0126539078944553 * numberOfRounds * numberOfRounds +
-2.11381964593463 * backwardErrorDelta + 0.154641076762567 *
abortError * abortError * abortError
```

```
22.0900863713677 * root + 13.4521003584348 * crossValidation +
11.5347082254957 * crossValidation * epsilon * epsilon +
```

Table C.57: Hipacc Regression Model - Best 3-way CA

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
3CA2	Reg	49	51	0.977	11.617	2.293

```
−94.0594630868332 * crossValidation * epsilon * epsilon *
considerEpsilonTube + −0.474021289392616 * numberOfRounds +
0.00541036361877658 * numberOfRounds * numberOfRounds +
−2.96133020312215 * learnTimeLimit_0 + −0.141825374330856 *
learnmirrowedFunction + −1.53760356395065 * learnasymFunction +
−1.36855074006413 * crossValidation * parallelization +
0.365297984830891 * numberOfRounds * crossValidation +
−0.198215910052391 * bruteForceCandidates + 3.05234430506881 *
learnasymFunction * limitFeatureSize + 2.1395647565276 *
candidateSizePenalty + −0.00415633207142472 * numberOfRounds *
numberOfRounds * crossValidation + −1.56080122905114 *
limitFeatureSize + 0.430219584360117 * minImprovementPerRound +
−2.70684374589516 * crossValidation * learnTimeLimit_30m +
2.1622096421192 * bruteForceCandidates * learnaccumulatedLogFunction
 + 0.775153999281543 * learnasymFunction * quadraticFunctionSupport
+ −2.0601273155768 * learnmirrowedFunction * learnlogFunction +
−2.65108665169328 * learnasymFunction * learnratioFunction +
−0.0348615906869004 * crossValidation_k * crossValidation_k +
0.998942812242666 * learnlogFunction + 0.29838379508381 *
learnasymFunction * crossValidation_k + 0.214886890532533 *
learnmirrowedFunction * crossValidation_k + −1.45765175035339 *
candidateSizePenalty * learnasymFunction + −3.05483366690867 *
```

```
epsilon * epsilon + 0.007917986922758 98 * candidateSizePenalty *
baggingNumbers
```

Table C.58: Hipacc Regression Model - Random

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
Rand	Reg	991	46	4.26	3.138	0.239

```
21.0722220366103 * root + 10.540650340918 * crossValidation +
   −0.426224604666194 * numberOfRounds + 0.00381126380193488 *
   numberOfRounds * numberOfRounds + 0.283788187758465 * numberOfRounds
    * crossValidation + −0.082676681736509 * crossValidation * epsilon
   * epsilon + −64.3143203240148 * crossValidation * epsilon * epsilon
   * considerEpsilonTube + −9.19906506474919E−11 * numberOfRounds *
   crossValidation * numberOfRounds * numberOfRounds * numberOfRounds *
    crossValidation * numberOfRounds * numberOfRounds +
   −1.22693442896886 * crossValidation * epsilon * epsilon *
   considerEpsilonTube * numberOfRounds + 0.00875140705888789 *
   numberOfRounds * bruteForceCandidates + 0.110674777342539 *
   numberOfRounds * minImprovementPerRound + 0.0647909437989701 *
   numberOfRounds * bruteForceCandidates * learnTimeLimit_30m +
   −0.0614717498912005 * numberOfRounds * bruteForceCandidates *
   learnTimeLimit_30m * crossValidation + 0.0122533181226953 *
   crossValidation * epsilon * epsilon * considerEpsilonTube *
   numberOfRounds * numberOfRounds + 0.0104216928584922 *
   numberOfRounds * candidateSizePenalty
```

Table C.59: Hipacc Regression Model - D0-3

Test	Trans.	# Configs	Complexity	RAE	MAE	MAE%
C0-3	Reg	548	11	1.911	20.241	1.918

```
19.1862844191896 * root + 20.5293508772183 * crossValidation +
   -0.475105503375282 * numberOfRounds + 0.0047507041985361 *
   numberOfRounds * numberOfRounds + 60.8680093356515 *
   minImprovementPerRound * minImprovementPerRound + 0.832986519455823
   * candidateSizePenalty + 1.1604040187415 * learnTimeLimit_30m +
   -0.631744124705771 * learnasymFunction + 0.889050268092996 *
   bruteForceCandidates
```