BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

AND

COLLEGE OF ENGINEERING

Dissertation

A DESIGN-BUILD-TEST-LEARN TOOL FOR SYNTHETIC BIOLOGY

by

EVAN APPLETON

B.S. Biomedical Engineering, Boston University, 2010

M.S. Bioinformatics, Boston University, 2012

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2016

Approved by

First Reader

Douglas Densmore, Ph.D.
Associate Professor of Computer Engineering

Second Reader

Kenneth Oye, Ph.D.
Associate Professor of Political Science and Engineering Systems
Massachusetts Institute of Technology, Cambridge, MA

A DESIGN-BUILD-TEST-LEARN TOOL FOR SYNTHETIC BIOLOGY

(Order No.)

EVAN APPLETON

Boston University Graduate School of Arts and Sciences

and

College of Engineering 2016

Major Professor: Douglas Densmore, Ph.D., Associate Professor of Computer Engineering

ABSTRACT

Modern synthetic gene regulatory networks emerge from iterative design-build-test cycles that encompass the decisions and actions necessary to design, build, and test target genetic systems. Historically, such cycles have been performed manually, with limited formal problem-definition and progress-tracking. In recent years, researchers have devoted substantial effort to define and automate many sub-problems of these cycles and create systems for data management and documentation that result in useful tools for solving portions of certain workflows. However, biologists generally must still manually transfer information between tools, a process that frequently results in information loss. Furthermore, since each tool applies to a different workflow, tools often will not fit together in a closed-loop and, typically, additional outstanding sub-problems still require manual solutions. This thesis describes an attempt to create a tool that harnesses many smaller tools to automate a fully closed-loop decision-making process to design, build, and test synthetic biology networks and use the outcomes to inform redesigns. This tool, called *Phoenix*, inputs a performance-constrained signal-temporal-logic (STL) equation and an abstract genetic-element structural description to specify a design and then returns iterative sets of building and testing instructions. The user executes the instructions and returns the data to *Phoenix*, which then processes it and uses it to parameterize models for simulation of the behavior of compositional designs. A model-checking algorithm then evaluates these simulations, and returns to the user a new set of instructions for building and testing the next set of constructs. In cases where experimental results disagree with simulations, *Phoenix* uses grammars to determine where likely points of design failure might have occurred and instructs the building and testing of an intermediate composition to test where failures occurred. A design tree represents the design hierarchy displayed in the user interface where progress can be tracked and electronic datasheets generated to review results. Users can validate the computations performed by *Phoenix* by using them to create sets of classic and novel temporal synthetic genetic regulatory functions in *E. coli*.

Acknowledgments

First and foremost, I would like to give special thank you to my adviser, Dr. Douglas Densmore, for his support, mentoring, and, advising over the course of this dissertation work. I would also like to give a big thank you to my other thesis committee members: Dr. Gary Benson, Dr. Ahmad Khalil, Dr. Scott Mohr, Dr. Kenneth Oye, and Dr. Wilson Wong. Their guidance and advice was invaluable in the completion of this work.

Next, I would like to acknowledge the funding sources of this work: National Science Foundation (NSF) and The Office of Naval Research (ONR). Without this funding, this work would not be possible. Additionally, I would like to thank the Boston University Graduate Program in Bioinformatics, Boston University, and the Synthetic Biology Engineering Research Center (Synberc) for various forms of support, including travel funding. I would also like to thank Dr. Christopher Voigt, Dr. Timothy Lu, Dr. Roger Tsien, Dr. Benjamin Glick, Dr. Oliver Griesbeck, and Addgene for generously sharing samples for the experimental work presented in this thesis.

Finally, I would like to thank my primary collaborators and mentors. Namely, I would like to give special thanks to Jenhan Tao, Prashant Vaidyanathan, Dr. Sonya Iverson, Diego Cuerda, Kevin Costa, Dr. Ernst Oberortner, and Dr. Traci Haddock. Professional interactions with these people greatly contributed to the quality of this work and my broader education.

Contents

List of Tables

List of Figures

Abbreviations

AGRN Abstract Genetic Regulatory Network.

DURC Dual Use Research of Concern.

EPA U.S. Environmental Protection Agency.

FDA Food and Drug Administration.

iGEM International Genetically Engineered Machine Competition.

MEFL Molecules of Equivalent Fluorescein.

PoET Program on Emerging Technologies.

RBS Ribosome Binding Site.

SGI Synthetic Genomics, Inc..

STL Signal Temporal Logic.

1 Introduction

1.1 Synthetic Biology

The era of modern genetics began when Watson and Crick discovered the structure of DNA in 1953[98], which laid the foundation for humans to understand heritable biological information. After an additional twenty years of research focused on understanding the mechanisms of nucleic acids, the first instance of *genetic engineering* was first reported[32] in the literature. This breakthrough was enabled by development of techniques for cleaving specific DNA strands with restriction enzymes[93] and pasting cleaved strands back together with DNA ligase[100]. This process of 'DNA cut and paste' paired with the technique of DNA amplification via polymerase chain reactions (PCR)[87] became the basis of modern *molecular cloning*. The capabilities of molecular cloning technology have since developed considerably to enable high-throughput DNA synthesis and custom DNA sequences for modification.

At first, these molecular cloning techniques were largely used to clone single genes or manipulate single genes of interest in a pre-existing natural genetic system. However, around the turn of the millennium, multiple groups started exploring the idea of engineering completely synthetic genetic systems from synthesized DNA. This materialized in the first instances of synthetic transcriptional regulatory networks in *E. coli*[40,47] and, consequently, to the establishment of the field of *synthetic biology*. These first synthetic genetic regulatory networks galvanized many groups in the biology and biotechnology community and inspired them to the create the first combinatorially-synthesized genetic networks[49], complex transcriptional-regulatory networks[17,35,95,96] and RNA regulatory networks[19,54]. Around this time, some of the first synthetic biology applications were also reported[11,58,83] and the biotechnology industry began to build around novel applications enabled by synthetic biolog.

Towards the end of the 2000's, as the techniques for engineering DNA improved and the demand for DNA synthesis grew, an emphasis was placed on developing methodologies for increasing the speed and scale of DNA synthesis[42,48,62,81,90,99]. These newer methodologies could be used to build larger and more complicated constructs at a higher throughput. Concurrently, the community's interest grew in applying ideas from other fields of engineering (specifically electrical engineering) to build these systems, and some synthetic biology groups started building synthetic biology 'logic gates'[12,97]. The interest in cellular logic has since grown[26,69] and recently, DNA recombinases have also been adopted in many of these systems as a means of creating digital logic and storing memory[27,44,91].

1.2 Bio-Design Automation

In parallel with the developments in biochemical technologies, engineers and computer scientists from other fields garnered interest in synthetic biology and began developing computational tools for engineering these genetic systems. One of the first realizations of scientists and engineers from this background was that there were few standards and little documentation for engineering these systems. In response, multiple efforts began working towards community standards[15,29,41,46,57,94] for data models and data exchange. This effort has recently gained more momentum[45] and seen support and involvement from larger government organizations[50]. Currently, some of the core requirements for representing sequence information has been discussed at length and interest has shifted into establishing standards for additional related sub-problems.

One of the foundations of other engineering fields is the definition and solving of sub-problems in a large and complicated process. This 'divide and conquer' approach allows for the solving of small parts of a larger problem one piece at a time and allows for specialists to tackle specific narrow problems of which they have considerable expertise. After all necessary pieces are defined and solved, solutions for sub-problem can be automated and connected to solve a larger problem. The central conjecture of this approach to synthetic biology is that the application of these core engineering concepts will accelerate synthetic biology workflows and increase the scale of systems that can be successfully built. This pursuit has recently been coined *bio-design automation*[36].

The current bio-design automation landscape can be divided into four main areas: design specification, DNA assembly planning and implementation, strain characterization and analysis, and machine learning. ‘Design specification’ concerns a precise, formal definition of the desired function and design of a target genetic system. ‘DNA assembly planning’ concerns the set of decisions needed to determine a plan for composing DNA constructs from their elements using specific cloning methodologies and ‘DNA assembly implementation’ refers to the processes and decisions involved with physically implementing a DNA assembly plan. ‘Strain characterization and analysis’ concerns the design and physical implementation of experiments for characterizing engineered strains and the accompanying analysis and interpretation of acquired data. ‘Machine learning’ concerns the automated revision of design based upon processed experimental outcomes.

1.2.1 Design Specification

Design specification concerns can be broken down into at least the following six sub--problems: 1. Formal specification; 2. Data model and mathematical model definition; 3. Simulation and verification; 4. DNA library definition; 5. Part assignment; 6. Design hierarchy. In recent years, a number of formal design specification tools have been introduced.

Some tools focus on the structure and organizational aspects of design[31,34,73,74,78], which largely concentrates on part and device design in terms of organization and selection of genetic elements. Other specification tools have put more emphasis on formal mathematically-defined functional specification[20,103], although tools of this type have not yet been explored in great depth.

Since every computational tool relies on a data model, several synthetic biology community data model standards have been developed[46,101] with notable differences. But, given the community emphasis on standardization, converters between many data models have been implemented. The scope of mathematical models for describing and simulating synthetic genetic systems is much larger and extends beyond the field of synthetic biology[66,68,72] to systems biology, in which more standard tools exist for mathematical model specification[56].

Numerous commercial and open source tools exist for building and editing custom DNA sequences such as *VectorNTI*, *Benchling*, *APE*, and *GenomeCompiler*, but they generally require users to upload specific DNA sequences as linear or circular fragments and custom-annotate them with sequence features. The most widely-adopted open-source format for exchanging this data is GenBank, an open source format supported by NIH, although other formats exist and have variable degrees of expressiveness. Since some features have measurable parameters associated with them, they can be ported into simulation tools to create simulated traces with a given mathematical model[30,60]. These traces can then be used by model-checking tools to verify their function[18,104].

The process of selecting features from feature libraries to apply to an abstracted genetic regulatory network design (AGRN) is known as part assignment. Recent tools for part assignment topic have been developed[88,102], but they do not tie in with verification tools. And, with the vast options for AGRN design with numerous tools, there remain many unsolved problems related to determining optimized part arrangements in AGRNs. Furthermore, these selections are heavily based on the assumptions in the underlying mathematical model and experimental design, which is often variable and can make these tools hard to apply.

Finally, although there are some tools that integrate the idea of hierarchical design and grammars to verify the validity of designs[34,78], there aren't any existing tools that decompose a large design into its functional and structural hierarchy and tie these breakdowns to mathematical models. Such a tool could be useful in unifying and organizing many of these concerns, since many of them overlap.

In summary, there has been significant thought and tool development on a variety of design specification problems, but there aren't any tools that unify all of these topics in a cohesive, meaningful way. Although data exchange standards have been developed, many tools were created before many aspects of the current community standard were agreed upon and in many cases, the older tools are no longer maintained, making them unusable under the newer standards.

1.2.2 DNA Assembly

Once a target design specification has been mapped to a set of target DNA constructs, a plan for how to assemble them must be determined. This problem, called the DNA assembly problem, has both design optimization and physical implementation aspects and can be broken into at least these four sub-problems: 1. Cloning and synthesis method selection; 2. Hierarchical assembly planning; 3. Primer design; 4. Physical implementation.

The set of all chemical reactions and flanking DNA sequence designs required to assemble one or more DNA fragments into a replicating biological vector defines a cloning method. Each cloning methods utilizes different enzymes, chemical reaction conditions, and required flanking sequence regions around DNA fragments. In recent years, numerous methods have been developed with variations in each of these categories[42,48,62,81,89,90,99], but all have the goal of scaling and modularizing cloning capabilities. Some methods place a heavy emphasis on modularity[99] and standardization[90], while others place a heavier emphasis on the scale of fragments that can be reliably assembled[48].

Once an assembly method is selected for a set of target constructs, the design questions relating to assembly can be separated into hierarchical assembly planning and primer design for PCR. Hierarchical assembly planning refers to the choice of which fragments to put together in parallel and in series. For large assemblies or large sets of assemblies, it is desirable that fragment re-use is maximized and the number of cloning steps required is minimized to optimize cloning time and material expense. Some prior work has been done on this topic[25,37], although it was restricted to older cloning methods and did not include options to account for many experimental concerns in the cloning process.

Once a plan for hierarchical assembly is determined, flanking regions for each of these fragments must be determined and primers must be designed for PCR such that the correct flanking regions are added to each fragment. This problem has been explored for many contemporary cloning methods[52] and many commercial sequence editors (mentioned above) have integrated tools for these decisions. Many of the decisions on this level are sensitive to cloning method and DNA chemistry and efficiency can be heavily impacted by small changes in flanking region design.

After this series of decisions is made, all decisions required for assembly planning have been made, except for detailed chemical protocols. The design of these protocols, namely PCR, DNA preparation, DNA transformation and organism growth conditions still has to be defined. Some of the required chemical protocols are standardized by the chemical manufacturer, but many cloning reaction reagents are not yet available in the form of a standard master mix with standard thermal reaction cycles. The problem of how to execute all necessary protocols efficiently is the protocol planning problem.

Some open-source[63] and industrial tools have been created to address these problems and automatically instruct liquid-handling robots to execute these tasks, removing the human element from the assembly process in an effort to reduce human cost and improve protocol reproducibility.

1.2.3 Strain Characterization

Physically-assembled DNA constructs must then be evaluated for function in the organism for which they were designed. The process of testing and evaluating synthetic DNA constructs is called strain characterization. The characterization sub-problems include: 1. Selection of measurement method and settings; 2. Definition of types of testing experiments and associated context based on measurement method and settings; 3. Data analysis and processing; 4. Data organization and visual representation.

In the bio-design automation field, the problem of formally selecting a measurement method or set of methods and then defining standard testing experiment types and settings is largely untouched and under-developed. This is likely due to the fact that bio-design automation is relatively new and not widely known or accepted by experimentalists, and even those who collaborate with pure experimental experts typically have little influence over experimental design. Some groups have started to develop standard flow cytometry protocols[75], but this topic has not been explored in depth in the literature.

Analysis tools for experimental data are more numerous and mature. Specifically, the synthetic biology community often uses single-cell fluorescence as the primary measurement technique and there are a wide variety of open-source[38,80] and industrial tools (*BBN Synbio Tools*, *FlowJo* and *BD FACSDiva*) for analyzing single-cell fluorescence data. This topic extends beyond the field of synthetic biology[79,86] and principles and standards for this type of data have been developed[51,85].

The problem of how to use this processed data to create intelligible graphs and present this and other design information to a human designer is another open problem in the bio-design automation field. In electrical engineering, data related to physical components is represented in the form of a datasheet. In the synthetic biology field, the necessary information for datasheets has began to be explored[15,29] and some groups are working on electronic datasheets, but it has yet to be demonstrated how these can be used meaningfully to help human designers refine their designs.

1.2.4 Machine Learning

Automatically using characterization data to revise designs produced from design specifications can be classified as machine learning. These machine learning sub-problems are the least developed and least defined in the bio-design automation space. There have been some recent efforts that use machine learning to revise design[92], but this space remains small due to the fact that the learning algorithms are heavily based upon a deep understanding of the experimental design of characterization experiments which tend to be highly variable as discussed in the prior section.

1.3 Regulation and Biosecurity

The field of synthetic biology and bio-design automation are rapidly growing[55] and broadening their accessibility to international audiences. A large contributor to this rapid and sustained growth is the International Genetically Engineered Machine (iGEM) competition[94]. The first iGEM at MIT in 2004 included 5 teams from 5 US universities, and has been steadily growing to over 280 teams from over 30 countries in 2015. Additionally, other government-funded initiatives and projects such as the *Synthetic Biology Project* at the Woodrow Wilson Center for Scholars and the Synthetic Biology Engineering Research

Center *Synberc* have helped centralize the efforts of academic and industrial researchers, social scientists and legal experts in the synthetic biology field, creating a small but diverse and growing community.

This rapid growth has many positive impacts, namely building general excitement about the positive aspects of synthetic biology technologies, building community, and creating awareness of synthetic biology projects. However, with such fast growth of a field with a small core and potentially revolutionary applications, the risks posed by synthetic biology technologies and applications must be taken seriously. The broad adoption of personal computing and internet technologies combined with the speed of the growth of the field has enabled a massive amount of new information and capabilities to be very broadly accessible.

A substantial, rapid growth of powerful technologies poses a challenge to many regulatory agencies. Debating and creating legislation is a historically slow process and, currently, the technological developments are greatly outpacing the regulatory framework. Recent calls[16,61,76] for regulatory agencies to get some of these technologies on their radar and react to them quickly, but not irrationally, have began to open up the public forum on how to address such issues. There is currently some pressure on international regulatory agencies to begin thinking about these topics, but in many cases it is still unclear what risk is posed by specific synthetic biology technologies, since very little risk-assessment research has been funded in the field. Furthermore, given the diversity of sub-topics in synthetic biology and the interdisciplinary nature of the field, there is still currently no widely adopted definition of 'synthetic biology'.

Although the bio-design automation sub-field is currently almost exclusively focused on basic science research of the core basic research problems and not applications, the long term vision is to automate many, if not all, of the decisions currently made to design these systems and make it easier for people to engineering biology. This goal is noble in the lens of a pure scientist, but there are implications of making an already powerful, accessible, and rapidly growing technology even more powerful and accessible. The specific consequences are hard to predict, but given that regulatory agencies are already behind the pace of the research, accelerating it even more could prove problematic. On the other hand, transitioning the decisions-making processes of biologists entirely into the digital realm creates unprecedented opportunities for documenting and tracking designs of synthetic biology projects. Given the growing world of computing and cyber-security, it is possible that principles of cyber-security could be incorporated into design tools to defend against possible harmful uses of synthetic biology.

1.4 A Design-Build-Test-Learn Tool

With the progress of the bio-design community in identifying and building tools to automate many necessary sub-problems of a synthetic biology design and implementation process, the fast growth of the broader synthetic biology community, and the possible implications of enabling an even broader community of biologists, now is the time to develop a software tool that integrates these design sub-problems into a closed-loop design-build-test-learn cycle. Recent prior work has made efforts to automate the full process of these cycles[21,39,84], but no tool has been able to harness all the pieces of the workflow in an integrated manner and some of the necessary sub-problems of these cycle have not yet been defined and automated.

In this thesis dissertation, I detail my work in creating a design-build-test-learn tool for design decisions in a synthetic biology workflow for creating synthetic genetic regulatory networks in *E. coli*. I accomplish this by re-using or modifying existing tools for solved sub-problems and identifying important sub-problems that have not seen much focus in prior work and creating working solutions for them. I preface the discussion of this work with the discussion of additional biosecurity and biosafety work I did in collaboration with the MIT PoET synthetic biology group and iGEM competition and analysis of regulatory gaps and risk assessment posed by a possible synthetic biology application, RNA-guided gene drives. It is my belief that these types of questions must be asked at the beginning of technical research projects to inform the path of project development. It is my hope that this body of work can both increase the scale and speed of synthetic biology and open opportunities for applying solutions to broader problems that synthetic biology poses.

2 Regulatory Gaps and Biosecurity

Synthetic biology is a unique field in the way that a central goal of all synthetic biology efforts is to produce an organism or genetic system with novel capabilities. This poses a problem because there is generally no precedent for novel engineered organisms or novel genetic systems and there exists minimal bench-marking or definition to use as a reference for determining risk. Furthermore, in addition to having no working reference point for assessing risk, there has been very little work to date to attempt to determine risks posed by specific applications. This problem is further complicated by the fact that these organisms and genetic systems are generally designed to self-replicate, which implies that any harm created by a synthetic biology application would be self-propagating and difficult to control.

With the lack of these reference points for risk, it is difficult to form meaningful safety or security regulations for synthetic biology-specific applications. Combined with the fast pace of technical developments in synthetic biology relative to the pace of regulatory framework development, most regulatory agencies are behind in the process of regulation formation for synthetic biology-specific applications. This leaves the field in a state where although there have yet to be any synthetic biology applications to cause major harm to consumers or the environment, a problem could arise relatively quickly with any particular application and it would be hard to predict or manage with the current sparse risk assessment and regulatory infrastructure.

This under-developed infrastructure imposes both great power and great responsibility to those people who are performing the technical work. Synthetic biologists have both the power to develop potentially revolutionary technologies and applications to advance the state of biotechnology and human health, and at the same time, the power to develop harmful technologies either through either carelessness, malevolence, or inadequate methodologies to assess risk.

These problems would lead a responsible synthetic biologist to ask the following questions: ‘‘Which synthetic biology applications should I develop? Which applications should I not develop? And why? And who will these applications impact?" My colleagues and I are of the opinion that these questions should be asked at the outset of synthetic biology application selection and development. The following section describes recent work we have done in this frame of mind in the sub-fields of biosafety and biosecurity screening, and, risk definition and regulatory gap identification for a potentially impactful synthetic biology application - RNA-guided gene drives.

2.1 Biosecurity

The largest current open community of synthetic biologists are the participants of the annual iGEM competition - an international synthetic biology project competition. The iGEM organization has spent the past decade encouraging teams to push their projects to the frontiers of synthetic biology. However, as the number of projects and their sophistication increases, so does the level of assumed risk. To manage this risk it would be useful to be able to adopt an existing international framework for assessing this risk. However, there is one major problem - no such framework currently exists.

In the absence of a coherent international framework for evaluating these risks in synthetic biology, iGEM has recently engaged with the MIT Program on Emerging Technologies (PoET) to develop a progressive approach for handling questions of safety and security. These two groups have worked together to create a rigorous screening program, acknowledging that a strengthened set of iGEM safety policies ultimately serves to expand, not contract, the universe of acceptable projects. This section reports on the policy process evolution thus far, screening findings from the 2013 competition, and expectations for future policy evolution.

Much like synthetic biology as a whole, iGEM has exploded in size, geographic scope, and technical capabilities over the past ten years. While this growth is beneficial, it also means that advancements have at times outpaced regulations. iGEM has reckoned with this mismatch most directly on issues of biosafety and biosecurity. However, rather than limiting projects' scope to remain conservative in the face of uncertainty, iGEM has engaged directly with safety challenges. Working with MIT (PoET), iGEM Headquarters has begun a multiyear process of developing progressive safety policies. This paper considers the motivations behind these changes, highlights the growth of key partnerships and collaborations, summarizes the 2013 safety screening findings, and looks ahead at opportunities for continued policy evolution.

2.1.1 iGEM Biosafety Screening Background

In 2011 and 2012, iGEM implemented a standardized screening system for teams safety forms. Prior to 2011, there was not a systematic review process in place. The new form consisted of questions prompting teams to (1) consider possible environmental, health, and safety implications of their projects and (2) provide sufficient information about their projects and procedures so that the Safety Committee could identify potential concerns. Before regionals, the MIT PoET group reviewed the forms, and projects that raised concerns were examined by the iGEM Safety Committee. Screening thresholds were set with a deliberate bias toward generating false positives as opposed to false negatives. Completion of the safety form was a requirement for participation.

Comprehensive project screening revealed a series of near misses in the 2011 and 2012 seasons. In iGEM, these apparent near misses were a consequence of inaccurate reporting. For example, one team improperly understood their project, reporting that they were using biological parts from an organism of concern in an insufficiently protective laboratory environment. On further review, the Safety Committee determined that the team had misclassified the biological parts with which they were working, and that the laboratory was appropriate for the true level of risk associated with their project. Near misses can serve as valuable sources of information for tracking potential weaknesses in a system, such as here where the team had clearly been insufficiently informed as to how to differentiate between safe and unsafe work.

The MIT PoET group used the results of two years of project screenings to propose changes to the 2013 process. These revisions were the product of discussions with the Safety Committee, faculty advisors, and iGEM Headquarters. The revisions aimed to shift the point of intervention closer to the time when actual laboratory work was being performed, such that potential hazards could be detected and prevented prior to a harmful event, rather than after the high-risk work had already been completed (**Figure 1**).

In 2013, teams submitted forms describing safety procedures and project implications, and also listed the chassis and parts used in their projects. If any parts or chassis were derived from mammals or organisms above risk group level one, teams also completed more detailed forms addressing areas of potential concern. Here, ‘‘chassis" refers to a host organism, such as Escherichia coli or Bacillus subtilis, into which a synthetic device is placed, while a genetic ‘‘part" is a component of the device, such as a promoter or terminator. Risk group assignments are based on the relative risk of the originating organism, as assigned by organizations such as the World Health Organization[5] and the U.S. National Institutes of Health Recombinant DNA Guidelines[3]. Assignment of organism-level risk group to its component parts is a conservative approach, but currently a necessary starting point due to the limited state of regulations. Developing an expedited screening protocol based on part functionality rather than organism of origin is a near-term goal.

Deadlines were earlier than in previous years and forms were required to be updated to reflect project changes, which facilitated intervention prior to teams conducting potentially dangerous work. However, the 2013 update did not make the screening occur early enough, as the majority of the process still took place beyond the point of maximum utility. This is a policy priority for future years. The program succeeded more significantly on other fronts, though, such as by instituting standardized data entry, requiring updated forms for relevant project changes, and increasing the emphasis placed on consideration of parts' functional properties. The program also strived for increased participant engagement with safety concerns and saw gains in the areas of more in-depth form reporting, team-driven shifts in project scope due to safety concerns, and active participation with the Safety list-serve, suggesting a desire to improve understanding rather than solely ticking check boxes.

Because of the process changes, participants had to be educated on risk group levels, changes in risk due to genetic modifications, the relationship of part functions to risk group assignments, and laboratory biosafety levels. However, as iGEM's new policies outpaced many international biosafety efforts, appropriate supporting educational documents had yet to also evolve. Multiple stakeholders assisted iGEM in providing guidance. One primary contributor was Public Health Canada, which aided in the development of the updated screening criteria. Additionally, J. Christopher Anderson and Terry Johnson of the University of California-Berkeley provided video instruction on traditional biological risk assessments, as well as on understanding and defining responsible conduct in synthetic biology.

2013 also marked the beginning of a collaboration between MIT PoET and iGEM with Synthetic Genomics, Inc. (SGI). This partnership resulted in SGI applying its proprietary screening tool, *Archetype*, to the entire iGEM Parts Registry. *Archetype*, which screens at a higher level of detail than the International Gene Synthesis Consortium (IGSC) standards

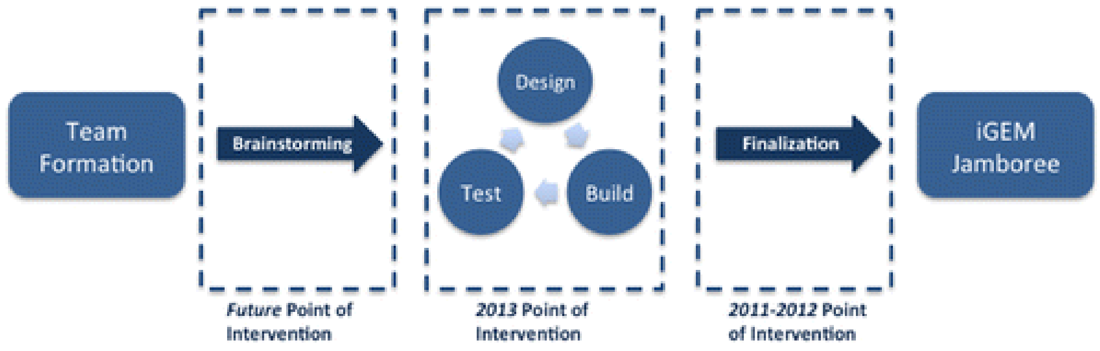

Team Formation
Brainstorming
Design
Test
Build
Finalization
iGEM Jamboree
Future Point of Intervention
2013 Point of Intervention
2011-2012 Point of Intervention

Figure 1: Advancing the point of intervention in biosafety screening. In 2011 and 2012, the safety process was limited to screening after projects had been completed (right); in 2013, the screening shifted closer to intervening during the design-build-test cycle (middle). Future iterations aim to move intervention further up the chain to maximize safety (left).

require, validated previous screening efforts by revealing no concerns that the Safety Committee had not already flagged. The results of this screening were also used to set terms of access to iGEM parts, and are providing an empirical basis for evaluating national regulations and international agreements governing parts safety and security. Continuation of this partnership through an annual screening of all newly submitted parts would institutionalize a vital secondary check within the overall safety system.

2.1.2 2013 Findings

The 2013 collegiate-level safety screening involved the review of 184 wet-lab teams before the regional jamborees. In a continuation of recent trends, the 2013 competition again witnessed increased project complexity and higher possible risk exposure. Here, the primary factors of safety concern-chassis risk group and part risk group are characterized by region and overall. Comments regarding laboratory biosafety levels are also included.

Chassis

The safety screen recorded the highest reported risk group level of chassis used per project. For any efforts involving an organism above risk group level one, a Secondary Form was also required. The vast majority of iGEM teams used chassis from the lowest risk group level; across all competitors, 90% employed no higher than a risk group level one chassis (**Table 1**).

	chassis				**part**			
risk group level	**1**	**2**	**3**	**other**	**1**	**2**	**3**	**other**
North America	92%	6%	0%	2%	56%	37%	0%	8%
Europe	86%	12%	0%	2%	59%	27%	2%	12%
Asia	90%	8%	0%	2%	52%	31%	3%	15%
Latin America	91%	9%	0%	2%	55%	31%	2%	12%
total	90%	9%	0%	2%	55%	31%	2%	12%

Table 1: Highest chassis and part risk group level per team, presented by region and in sum. ‘‘Other" refers to areas of unresolvable assignment uncertainties.

Parts

The 2013 iGEM safety screen also required information on any new or modified coding regions that teams were using in their projects. A Secondary Safety Form was required for any part sourced from a risk group two or higher organism, or from a mammal. Parts from the 2013 Distribution Kit were exempted from review. Overall, 55% of iGEM teams reported no use of parts from higher than a risk group level one organism (**Table 1**). A further 31% reported use of parts from risk group level two organisms.

Teams' detailed reporting in the Basic and Secondary Forms allowed for intervention on all serious concerns prior to the Jamborees. Though not every safety concern was fully resolved, there were no last-minute surprises in 2013. The quality of reporting was mixed, with some teams providing exemplary responses and demonstrating deep consideration of the relevant issues, while others were either cursory in their efforts, or were uninformed about their universities' or countries' biosafety regulations.

The most troubling mistake found across multiple forms was teams incorrectly asserting that their universities had no Institutional Biosafety Committee or equivalent group. Teams' home universities hold the key responsibility for ensuring sound laboratory practices, so a lack of understanding of these resources and requirements is cause for concern, and should be a target for future educational efforts. The importance of home institutions serving as the safety backbone of iGEM is reinforced by the self-reporting nature of the safety policies. While standardized forms requesting specific data helped to improve reporting, there remain no ready means for iGEM to ensure the veracity of statements provided. Therefore, emphasizing compliance and consultation with home institutional biosafety entities, which do have access to laboratories for verification purposes, helps to reduce the uncertainty around responses provided. Further, planned improvements to guidance documents will include better explanations of the intentions of various questions in the coming year, and thus, the safety process should expect more informed responses.

2.1.3 Future Considerations

Changes made to the 2013 screening process marked an important step in the overall evolution of safety policies within iGEM. However, improving safety at iGEM is an iterative process, and lessons learned from 2013 will necessarily inform changes to the 2014 effort. Of these modifications, the following are of top priority:

1. Pre-approval of projects exceeding certain risk thresholds. iGEM is striving to attain points of intervention that optimize participant safety while maintaining project flexibility (**Figure 1**). By requiring advance approval of plans to use organisms or parts more likely to present hazards, iGEM aims to prevent situations such as those in 2011-2013, in which teams worked with dangerous components before the safety screeners were made aware of their plans and had an opportunity to act.

2. Improved clarity and guidance. Much remains unknown about how to assess risk when organisms are broken down to component pieces. iGEM and collaborators have attempted to provide guidance; however, significant room for improvement remains.

Guidance documents must be produced concomitant with policy evolution in order to provide clarity in this area.

3. Increased advisor involvement. The iGEM safety process relies on teams' home universities, and thus, active advisor involvement is vital. The 2014 process will continue to work on facilitating communication and engagement with the overseeing parties. A strengthened set of iGEM safety policies ultimately serves to expand the universe of acceptable projects. By understanding areas of concern, and knowing how to address them responsibly, teams are capable of working safely along the technology frontier. Safety policies are evolving in pursuit of this goal, and with this aim, safety at iGEM is pointing toward the future.

2.1.4 Continued Developments

In 2014 and 2015, safety screening policies in iGEM followed the outlined future considerations from 2013. iGEM headquarters hired a full-time employee to handle the safety screening process and held safety and security-specific meetings at the 2014 and 2015 iGEM giant jamborees to discuss specific projects that exposed new security and safety concerns. Additionally, the yearly screening of sequences in the Registry by SGI was continued and flagged appropriate sequences of concern.

In summary, the safety screening process in iGEM represents one of the most thorough long-term efforts to evaluate synthetic biology designs for both sequence, function and design objective. Although iGEM shifted from volunteer screeners to a full-time employee dedicated to screening projects, these projects do not represent all synthetic biology projects and will not scale as the research field continues to increase in size. In the future, there will need to be automated tools for evaluating and flagging designs with respect to safety and security to supplement these screening efforts.

2.2 Risk Assessment and Regulatory Gaps

The iGEM safety screening process represents a methodology for identifying projects that raise safety and security concerns. Once a project or application is identified, the risks for that application must be assessed and it must be determined if the existing local and global regulations are adequate for governing them.

Since many regulations that would be applied to synthetic biology applications were created before the synthetic biology field existed, considerations for problems exposed by synthetic biology technologies can be insufficient. In the case of RNA-guided gene drives which strongly bias inheritance, my colleagues and I found this to be the case.

2.2.1 RNA-Guided Gene Drives

Genes in sexually reproducing organisms normally have, on average, a 50% chance of being inherited, but some genes have a higher chance of being inherited. These genes can increase in relative frequency in a population even if they reduce the odds that each organism will reproduce. Aided by technological advances, scientists are investigating how populations might be altered by adding, disrupting, or editing genes or suppressed by propagating traits that reduce reproductive capacity[28,43]. Potential beneficial uses of such ‘‘gene drives" include reprogramming mosquito genomes to eliminate malaria, reversing the development of pesticide and herbicide resistance, and locally eradicating invasive species. However, drives may present environmental and security challenges as well as benefits.

Gene drives are subject to two fundamental limitations. First, drives will only function in sexually reproducing species, so they cannot be used to engineer populations of viruses or bacteria. Second, a newly released drive will typically take dozens of generations to affect a substantial proportion of a target population, unless drive-containing organisms are released in numbers constituting a substantial fraction of the population. The process may require only a year or less for some invertebrates, but centuries for organisms with long generation times.

Studies have evaluated the possibility of releasing transgenic mosquitoes to combat the spread of malaria, dengue, and other mosquito-borne diseases, including requirements for containment, testing, controlled release, and monitoring of mosquito gene drives. This work will need to be replicated and extended for proposed gene drives seeking to alter other species[22,67]. It is crucial that this rapidly developing technology continue to be evaluated before its use outside the laboratory becomes a reality.

2.2.2 Technical Developments

One promising method for creating a gene drive uses targeted endonuclease enzymes to cut a specific site in the DNA of the organism. In organisms that inherit one chromosome with this enzyme's gene and one without it, the endonuclease will cut the latter, inducing the cell to copy the endonuclease and surrounding genes onto the chromosome that previously lacked them (**Figure 2**). Ten years ago, Burt proposed using endonuclease drives to spread traits that would control diseases borne by insect vectors[28]. He suggested that drives could be designed to add or delete genes and suppress populations, potentially to the point of extinction. However, no drive capable of spreading efficiently through a wild population has yet been developed. A major reason has been the difficulty of programming drives to cut desired sequences at high efficiency.

Scientists recently developed a powerful and efficient tool for genome engineering that uses the CRISPR nuclease Cas9 to cut sequences specified by guide RNA molecules[33,67]. This technique is in widespread use and has already engineered the genomes of more than a dozen species. Cas9 may enable ‘‘RNA-guided gene drives" to edit nearly any gene in

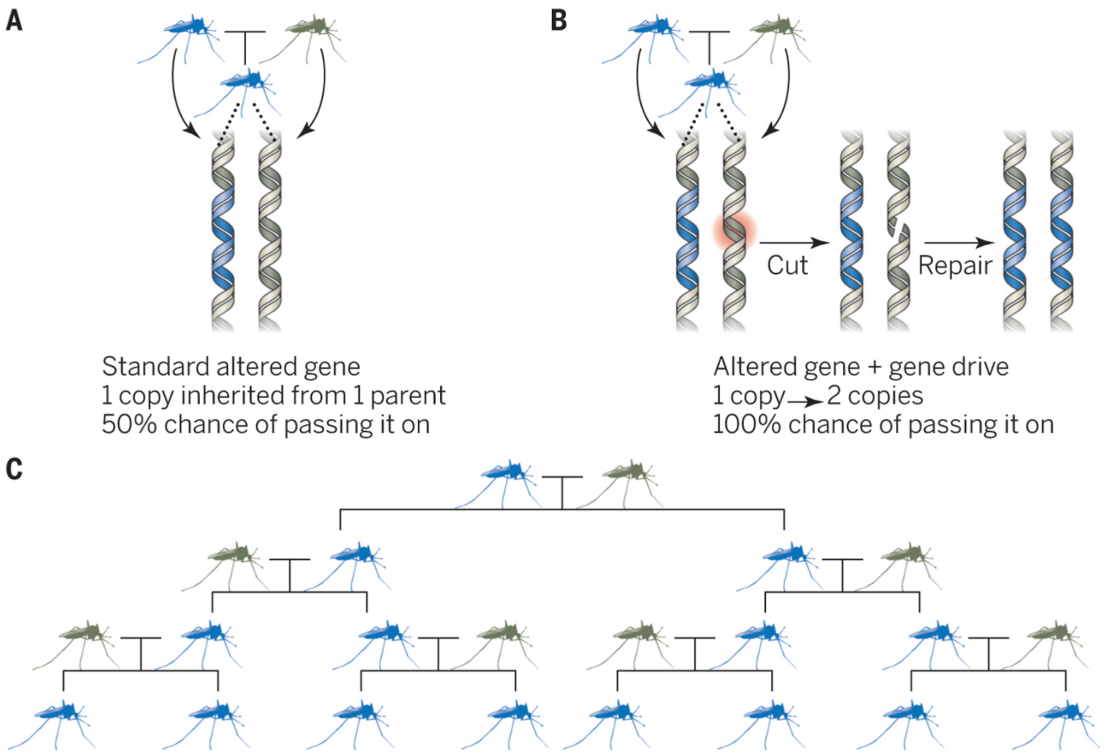

A
B
Cut
Repair
Standard altered gene
1 copy inherited from 1 parent
50% chance of passing it on
Altered gene + gene drive
1 copy → 2 copies
100% chance of passing it on
C

Figure 2: How endonuclease gene drives spread altered genes through populations. (a) Altered genes (blue) normally have a 50% chance of being inherited by offspring when crossed with a wild-type organism (gray). (b) Gene drives can increase this chance to nearly 100% by cutting homologous chromosomes lacking the alteration, which can cause the cell to copy the altered gene and the drive when it fixes the damage. (c) By ensuring that the gene is almost always inherited, the gene drive can spread the altered gene through a population over many generations, even if the associated trait reduces the reproductive fitness of each organism. The recently developed CRISPR nuclease Cas9, now widely used for genome engineering, may enable scientists to drive genomic changes that can be generated with Cas9 through sexually reproducing organisms (1).

sexually reproducing populations[43].

To reduce potential negative effects in advance of construction and testing, Esvelt et al. have proposed several novel types of drives[43]. Precision drives could exclusively affect particular species or subpopulations by targeting sequences unique to those groups. Immunizing drives could block the spread of unwanted gene drives by preemptively altering target sequences. Reversal drives could overwrite unwanted changes introduced by an initial drive or by conventional genome engineering, even restoring the original sequence. However, ecological effects would not necessarily be reversed. These and other RNA-guided gene drives have yet to be demonstrated in the laboratory.

2.2.3 Environmental and Security Aspects

A recent workshop examined key questions concerning effects of development and use of gene drives in varied species and contexts[10,59].

Targeted wild organisms.

Scientists have minimal experience engineering biological systems for evolutionary robustness. Drive-induced traits and altered population dynamics must be carefully evaluated with explicit attention to stability. For example, a drive may move through only part of a population before a mutation inactivates the engineered trait. In some cases, preferred phenotypes might be maintained as long as new drives encoding updates are periodically released. The effects of a strategy dependent on repeatedly releasing drives to alter a population should be thoroughly assessed before use.

Nontargeted wild organisms.

In theory, precision drives could limit alterations to targeted populations, but the reliability of these methods in preventing spread to non-target or related populations will require assessment. To what extent and over what period of time might cross-breeding or lateral gene transfer allow a drive to move beyond target populations? Might it subsequently evolve to regain drive capabilities in populations not originally targeted? There may also be unintended ecological side effects. Contained field trials should be performed before releasing organisms bearing a drive that spreads the trait.

Crops and livestock.

A technology capable of editing mosquito populations to block disease transmission could also be used to alter populations of agricultural plants or livestock by actors intent on doing harm. However, doing so surreptitiously would be difficult because many drive-containing organisms must be released to alter populations within a reasonable time span. Moreover, drives are unlikely to spread undetected in contract seed production farms and animal breeding facilities that test for the presence of trans-genes. It would thus be difficult to use drives to affect food supplies in the United States and other countries that rely on commercial seed production and artificial insemination. Developing countries that do not use centralized seed production and artificial insemination could be more vulnerable.

Humans.

Gene drives will be ineffective at altering human populations because of our long generation times. Furthermore, whole genome sequencing in medical diagnostics could be used to detect the presence of drives. Drives are thus not a viable method for altering human populations. Rare individuals might experience an allergic reaction to peptides in the Cas9 protein if exposed to an affected organism. Thus, toxicological studies should be conducted to confirm that proposed drive components are safe.

2.2.4 Toward Risk Management

We recommend the following steps toward integrated management of environmental and security risks:

i Before any primary drive is released in the field, the efficacy of specific reversal drives should be evaluated. Research should assess the extent to which the residual presence of guide RNAs and/or Cas9 after reversal might affect the phenotype or fitness of a population and the feasibility of reaching individual organisms altered by an initial drive.

ii Long-term studies should evaluate the effects of gene drive use on genetic diversity in target populations. Even if genome-level changes can be reversed, any population reduced in numbers will have reduced genetic diversity and could be more vulnerable to natural or anthropogenic pressures. Genome-editing applications may similarly have lasting effects on populations owing to compensatory adaptations or other changes.

iii Investigations of drive function and safety should use multiple levels of molecular containment to reduce the risk that drives will spread through wild populations during testing. For example, drives should be designed to cut sequences absent from wild populations, and drive components should be separated.

iv Initial tests of drives capable of spreading through wild populations should not be conducted in geographic areas that harbor native populations of target species.

v All drives that might spread through wild populations should be constructed and tested in tandem with corresponding immunization and reversal drives. These precautions would allow accidental releases to be partially counteracted.

vi A network of multipurpose mesocosms and microcosms should be developed for testing gene drives and other advanced biotechnologies in contained settings.

vii The presence and prevalence of drives should be monitored by targeted amplification or metagenomic sequencing of environmental samples.

viii Because effects will mainly depend on the species and genomic change rather than the drive mechanism, candidate gene drives should be evaluated on a case-by-case basis.

ix To assess potentially harmful uses of drives, multidisciplinary teams of experts should be challenged to develop scenarios on deliberate misuse.

x Integrated benefit-risk assessments informed by the actions recommended above should be conducted to determine whether and how to proceed with proposed gene drive applications. Such assessments should be conducted with sensitivity to variations in uncertainty across cases and to reductions in uncertainty over time.

2.2.5 Regulatory Gaps

The prospective development of drives highlights the need for regulatory reform. Currently, U.S. regulations would treat drives as veterinary medicines or toxins. U.S. policies and international security regimes rely on a listed-agent-and-toxin approach. Neither addresses challenges posed by gene drives and other advanced biotechnologies.

U.S. environmental regulations

Responsibility for regulating animal applications of drives in the United States rests with the Food and Drug Administration (FDA). An FDA guidance issued in 2009 states that genetically engineered DNA constructs intended to affect the structure and function of an animal, regardless of their use, meet the criteria for veterinary medicines and are regulated as such. Developers are required to demonstrate that such constructs are safe for the animal. Approval of new veterinary medicines is to be based on the traditional FDA criterion ‘‘that it is safe and effective for its intended use"[6]. It is unclear whether these requirements can be reconciled with projected uses of drives, including suppression of invasive species. Nor is it clear how this guidance would apply to insects. The application of existing U.S. Department of Agriculture (FDA) and U.S. Environmental Protection Agency (EPA) regulations governing genetically modified organisms to gene drives is also ambiguous, with jurisdictional overlaps across the Federal Insecticide, Fungicide, and Rodenticide Act, the Toxic Substances Control Act, and the Animal and Plant Health Inspection Service[8].

International environmental conventions

Existing international conventions cover international movements of gene drives, but do not define standards for assessing effects, estimating damages, or mitigating harms. International movements of living modified organisms are treated under the 2003 Cartagena Protocol on Biosafety, ratified by 167 nations not including the United States and Canada. Article 17 of the Protocol obligates parties to notify an International Biosafety Clearinghouse and affected nations of releases that may lead to movement of living modified organisms with adverse effects on biological diversity or human health. Other provisions empower nations to use border measures to limit international movements, but these measures are not likely to control diffusion of drives. The 2010 Nagoya-Kuala Lumpur Supplementary Protocol calls on Parties to adopt a process to define rules governing liability and redress for damage from international movements. Neither the process nor rules have been defined[4].

U.S. security policies

The draft U.S. Government Policy on Dual Use Research of Concern (DURC) combines a broad definition of concerns with a narrow definition of scope of oversight, the latter focusing on experiments of concern on listed pathogens and toxins[9]. The listed-agent-toxins approach is also used in the U.S. Select Agent Rule, USDA Select Agents/Toxins, and Commerce Department export control regulations. Drives do not fall within the scope of required oversight of DURC and other listed-agent-toxin-based policies.

International security conventions

The UN Biological Weapons Convention defines areas of concern in broad terms with the intention of providing latitude to adapt to evolving technologies and threats. Article 1 bans development, production, or stockpiling of all biological agents or toxins that have no justification for prophylactic, protective, and other peaceful purposes and weapons, equipment or means of delivery designed to use such agents or toxins for hostile purposes[1,2]. However, national implementation measures defining operational oversight and Australia Group Guidelines governing exports rely on narrow lists of organisms, toxins, and associated experiments[7]. Gene drives and most other advanced applications of genomic engineering do not use proscribed agents or create regulated toxins and hence fall beyond the scope of operational regulations and agreements.

Filling the regulatory gaps

We recommend adopting a function-based approach that defines risk in terms of the ability to influence any key biological component the loss of which would be sufficient to cause harm to humans or other species of interest. The agents and targets of concern with a functional approach could include DNA, RNA, proteins, metabolites, and any packages thereof. Thus, suppression drives would be covered because they would cause loss of reproductive capability in an animal population, whereas an experimental reversal drive that could only spread through engineered laboratory populations could be freely developed. Steps taken to mitigate environmental concerns will address security concerns and vice versa. Regulatory authority for each proposed RNA-guided gene drive should be granted to the agency with the expertise to evaluate the application in question. All relevant data should be made publicly available and, ideally, subjected to peer review[82].

2.2.6 Conclusions

For emerging technologies that affect the global commons, concepts and applications should be published in advance of construction, testing, and release. This lead time enables public discussion of environmental and security concerns, research into areas of uncertainty, and development and testing of safety features. It allows adaptation of regulations and conventions in light of emerging information on benefits, risks, and policy gaps. Most important, lead time will allow for broadly inclusive and well-informed public discussion to determine if, when, and how gene drives should be used.

2.3 Implications of Bio-Design Automation

As discussed in this chapter, there are some synthetic biology applications that expose considerable regulatory gaps in both U.S. and international regulatory frameworks and these applications require additional risk-assessment research. Moving forward, there will need to be mechanisms for automatically flagging projects and applications with such concerns. One such solution is to build software tools for automated risk assessment of sequences and applications. Currently, the only known tool to solve this problem is

Synthetic Genomics' *Archetype* tool. However, since it is a proprietary software, the way in which it screens sequences is not clearly documented and there is no open-source tool for evaluating sequences in an open community.

Since the bio-design automation field aims to accelerate the scale and pace of synthetic biology, it important to begin developing tools for addressing biosecurity concerns. It would also be useful to include metrics for risk assessment in future design tools, so that designs that create risk are flagged for additional evaluation.

At its current scale and pace of development, considerable regulatory gaps are already exposed by current synthetic biology applications and the number of projects with such concerns will only increase as the pace of technological advance accelerates. Fortunately, this also poses great potential for software developers to create tools to flag and document risky designs or designs that raise biosecurity concerns. In coming years it will become more and more important for synthetic biologists to ask themselves which projects they are pursuing and why before the path of research is defined, as their design power increases from bio-design automation tools.

3 Design Workflow Overview

Once a synthetic biologist has determined which application to develop and given due consideration to the biosecurity and risk assessment aspects, they must form a plan for developing the technology. In synthetic biology, this requires a full workflow that includes creating genetic designs, physically assembling them, testing them, and using the outcomes to refine designs, if necessary. This thesis is focused on building a tool to automate decisions in each one of these areas for a workflow tailored to building transcriptional regulatory networks in *E. coli*. To accomplish this task, we created a tool called *Phoenix*. *Phoenix* combines seven existing software tools and introduces more than four additional sub-problem definitions and solutions. A majority of these tools are not presented directly to a user, but are used extensively on the back end.

Phoenix receives an input design specification and parts library and produces a design hierarchy for the input design specification. It then returns a set of experimental instructions for the lowest lever of the design hierarchy for the user to implement. After the user completes the experiment, they return the collected data to *Phoenix* as per the experimental instructions and, as additional phases of building and testing are necessary, *Phoenix* returns new sets of building and testing instructions to the user for each level in the design hierarchy. In instances where the acquired experimental results do not match the simulated predictions, *Phoenix* instructs additional building and testing instructions to attempt to isolate design failures.

This process starts with uploading of all DNA sequences features and plasmids into a sequence editor tool (*Benchling*) and exporting annotated sequences into a multi-part Genbank file. These file are then uploaded into *Phoenix* and saved into the user's *Clotho* database (**Step 0**). *Phoenix* is then idle until it receives a set of design specifications

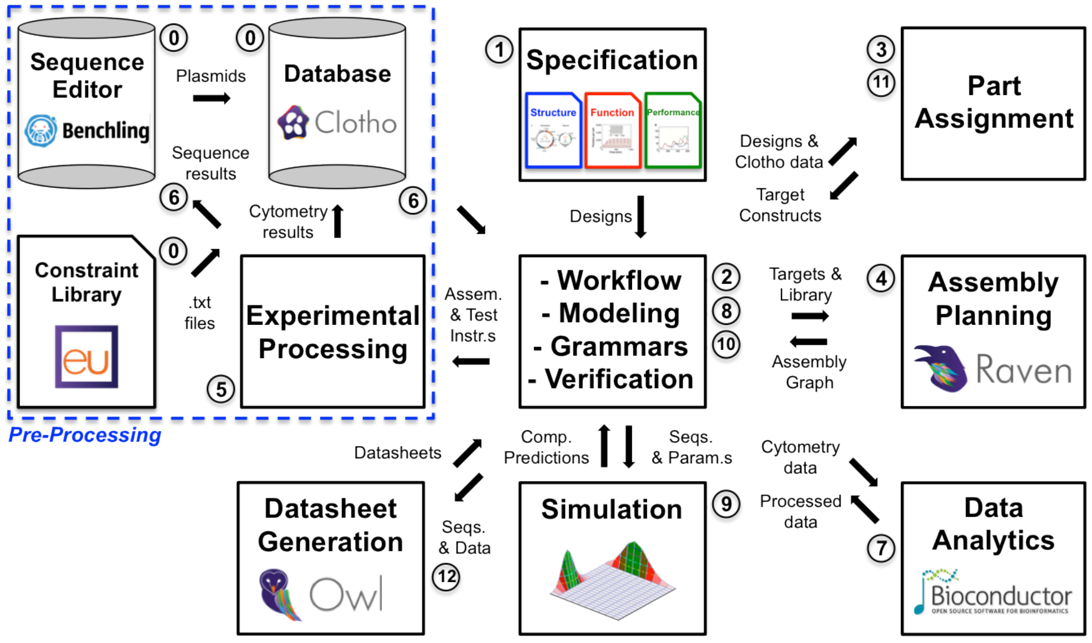


Figure 3: *Phoenix* tool software architecture

from the user (**Step 1**) in the form of a performance-constrained signal temporal logic equation (STL) (‘function’ and ‘performance’ specifications) and a constraint set used to produce abstract genetic regulatory networks (AGRNs) using *miniEugene* (‘structural’ specification). These specifications are verified for validity and then decomposed using structure- and function-based grammars within *Phoenix* (**Step 2**).

These decomposed genetic structures are then supplemented with temporary testing parts and the first round of optimized part assignment is performed to determine the sequence-assigned constructs for the first testing phase (**Step 3**). These target parts and the parts library are then exported into *Raven*[14] to produce an optimized DNA assembly plan (**Step 4**). At this point, the user is returned a file with instructions for DNA assembly, oligos needed for assembly, and a ‘key file’ that represents the testing conditions required for subsequent data analysis (**Step 5**) after building is complete.

The user then executes the building and testing instructions and returns cytometry data and new multi-part GenBank files for the new plasmids back into *Clotho* via the sequence editor and the annotated key file (**Step 6**). Raw cytometry data is processed by a data analysis script using the *Bioconductor* library in R (**Step 7**) and the processed data is then mapped to an SBML file, representative of the core *Phoenix* mechanistic model. An algorithm for parameter estimation is applied from the *COPASI* toolbox to determine rate constants for each construct from the input data (**Step 8**).

At this point, all parameters for the current stage of the design hierarchy are known and can be used to simulate compositional behavior with *COPASI* (**Step 9**). Next, each simulation is model-checked with the decomposed functional specification for each construct and those constructs which do not satisfy the specified performance are filtered out, and those that satisfy the decomposed functional specification are rank-ordered for robustness (**Step 10**). Then, the constructs that are predicted to most robustly meet the decomposed functional specification are mapped back to parts and another round of part assignment optimization (**Step 11**) is performed. Lastly, the results of both the simulated and observed behavior can be viewed with electronic datasheets using *Owl*[13] (**Step 12**).

In instances where there are mismatches between the simulated compositional behavior and the data acquired upon building and testing, *Phoenix* uses grammars that define common failure modes of transcriptional regulatory networks to instruct the building and testing of additional intermediate networks to probe changes in the rate constants with respect to the rate constants deduced from the first phase of testing.**Steps 4-12** are repeated iteratively up the design hierarchy determined by the *Phoenix* grammars (**Step 2**), but the user only ever views building end testing instructions (**Step 5**), a graphical map of the design hierarchy (**Step 2**) and results in the user interface (**Step 12**).

4 Genetic Regulatory Network Design Specification

The entry point to any design-build-test-learn cycle is the design specification. This represents the formal description of the desired outcome (i.e. ‘target’) of a project. In synthetic biology projects, design specifications are typically defined informally as ‘design objectives’ that can range from a target yield of a small molecule produced by cells’ metabolism to a target level of expression of a gene of interest. Design specifications can also be formal mathematical descriptions of functional behavior. Other engineering fields rely on formal specification tools and methods because they are well-defined, which allows for quantitative reproducibility and can be analyzed to create more complex function. Furthermore, formal specifications provide a strong foundation for compositional designs that rely on the defined function of their components.

In synthetic biology, formal design specification has not been explored deeply, but many formal specifications come in the form of boolean truth table. This is a powerful and simple abstraction borrowed from electrical engineering that a digital output (1 or 0) for an arbitrary number of digital inputs (also 1 or 0). Recent work has demonstrated sophisticated boolean logic function for a variety of genetic network types. Boolean logic has been demonstrated for transcriptional regulatory networks[97], recombinase-based networks[91], and RNA-based networks[65] and still currently serves as the inspiration for many synthetic biology projects.

While this abstraction has garnered a lot of excitement due to its importance in other engineering fields and has recently been widely adopted in parts of the synthetic biology community, there has yet to be a clear application of genetic logic used to solve a real-world problem. From a technical perspective, boolean specifications are not time-dependent and generally are not performance-constrained. For example, this type of specification cannot be easily used to specify a genetic toggle switch[47] or a genetic oscillator[40,95]. These specifications also do not have easy ways for attaching quantitative performance metrics.

An ideal specification for synthetic biology applications would define the desired function and performance for a wide variety of synthetic genetic networks. Given the breadth of platforms used in synthetic biology, it would also be useful and necessary to supply a structure-based specification, or AGRN. Also considering this breadth of platforms, it does not seem reasonable that a function alone can be used to determine what performance should be required of any target network or which type of network platform would be best for satisfying an particular function. For these reasons, in *Phoenix*, a user must enter a specification that individually defines the desired structure, function and performance of a target network. It would be desirable to create a design 'language' that can be used a specification tools to create and validate designs.

4.1 Grammars for Design Decomposition

In computer science, an important abstraction for validating expressions a language is grammars. Formal grammars are defined by four components: 1. Terminal symbols which serve as the elemental building blocks; 2. A start symbol to represent a complete statement; 3. Non-terminal symbols which serve as intermediate symbols between terminals and the start; and 4. Production rules for defining how terminal symbols can be combined to produce non-terminals and ultimately a start. To give a non-technical example, consider a sentence in English. A sentence would be the start symbol and the words would be the terminals. Non-terminals would include clauses, phrases and participles which are made from terminals and used to build a full sentence.

Grammars can be useful for generating sets of complete start symbols given a library of terminals, but can also be used to decompose a start symbol into its terminals. This can be useful for breaking very large expressions into it's elements. In *Phoenix*, we apply this abstraction to structural genetic design specifications to decompose these specifications into abstract design trees.

4.1.1 Signal Temporal Logic

In systems engineering, there is a set of formal methods used for time-dependent specifications called temporal logics. These logics are used for a variety of applications, including robotics, and also include the set of all boolean expressions. One particular logic in this family of specification languages, signal temporal logic STL, can specify a signal over time with performance constraints. Me and my collaborators selected this language as the form of function and performance specification in *Phoenix*.

- *Function*: Inverter (NOT gate)
- *English Specification*: If the *TetR* signal is higher than T_H, the *rfp* signal is expected to become lower than T_L within a time period of k_1 and persist for a time period of k_2.
- *STL Specification*: $(x_{TetR} > T_H) \Rightarrow (F_{[0,k_1)} G_{[0,k_2)} x_{rfp} < T_L)$

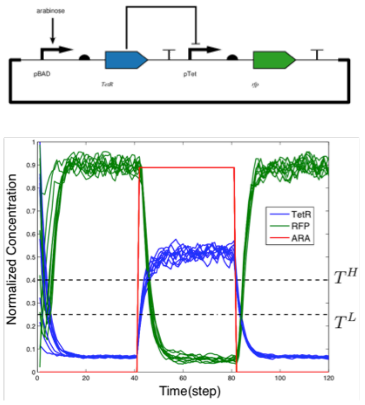
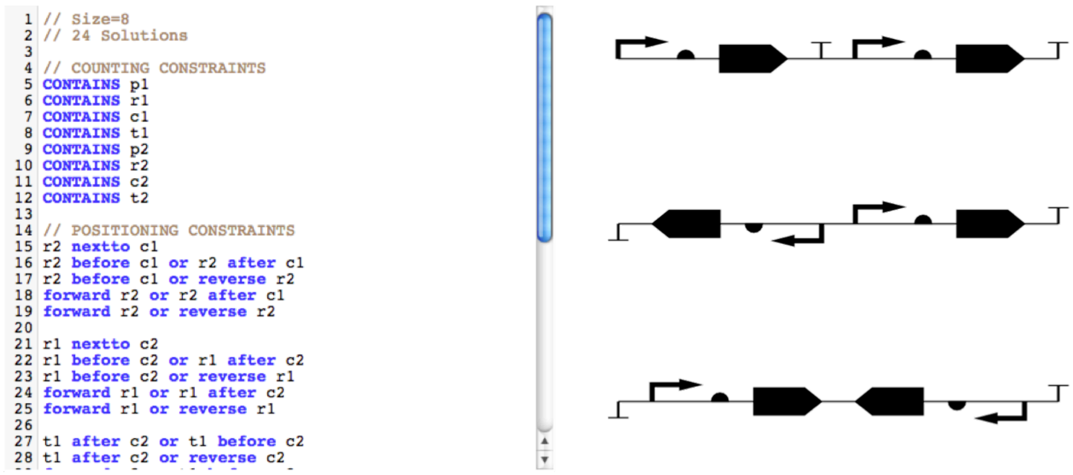


Figure 4: Specification of a 'NOT gate'. The English language specification and STL specification (left). An example of a NOT gate structure and illustration of the temporal function (right).

This language has its own grammar to validate STL expressions independent of their applications, but no current mechanism by which to decompose an STL function into its sub-functions. Since this math does not yet exist, in *Phoenix*, we leverage another STL variant, parameterized STL (PSTL) to compose to compose large specifications consisting of the functional terminals: 'NOT gate', 'toggle switch' and 'oscillator'.

4.1.2 Abstract Genetic Regulatory Network Specification

Each genetic platform has different structural genetic elements. Transcriptional GRNs in textitE. coli can be defined as compositions of promoters, ribosome binding sites (RBSs), regulator coding sequences, terminators and vectors with origins of replication and antibiotic resistance. From these five elements, an infinite number of AGRNs can be created, but only a subset of those AGRNs contain complete sets of transcriptional units arranged to express all contained regulatory coding sequences.



```
// Size=8
// 24 Solutions

// COUNTING CONSTRAINTS
CONTAINS p1
CONTAINS r1
CONTAINS c1
CONTAINS t1
CONTAINS p2
CONTAINS r2
CONTAINS c2
CONTAINS t2

// POSITIONING CONSTRAINTS
r2 nextto c1
r2 before c1 or r2 after c1
r2 before c1 or reverse r2
forward r2 or r2 after c1
forward r2 or reverse r2

r1 nextto c2
r1 before c2 or r1 after c2
r1 before c2 or reverse r1
forward r1 or r1 after c2
forward r1 or reverse r1

t1 after c2 or t1 before c2
t1 after c2 or reverse c2
```

Figure 5: Structural specification in *Phoenix*. An example *miniEugene* file with structural constraints (left). Three valid AGRNs from this structural specification (right)

Phoenix, has a grammar for determining if a candidate AGRN contains complete sets of transcriptional units, considering both the forward and reverse strands. *Phoenix* comes pre-loaded with files that can be input into an AGRN-generation tool, *miniEugene*, and define the set of all valid AGRNs for a transcriptional 'NOT gate', 'toggle switch' and 'oscillator'. These AGRN outputs are considered the starts of the structural grammar and map to the terminals of the STL grammar.

In *Phoenix*, we use this grammar to decompose genetic structures into abstract design trees. The foundation of these grammars are the biochemical interactions involved in genetic

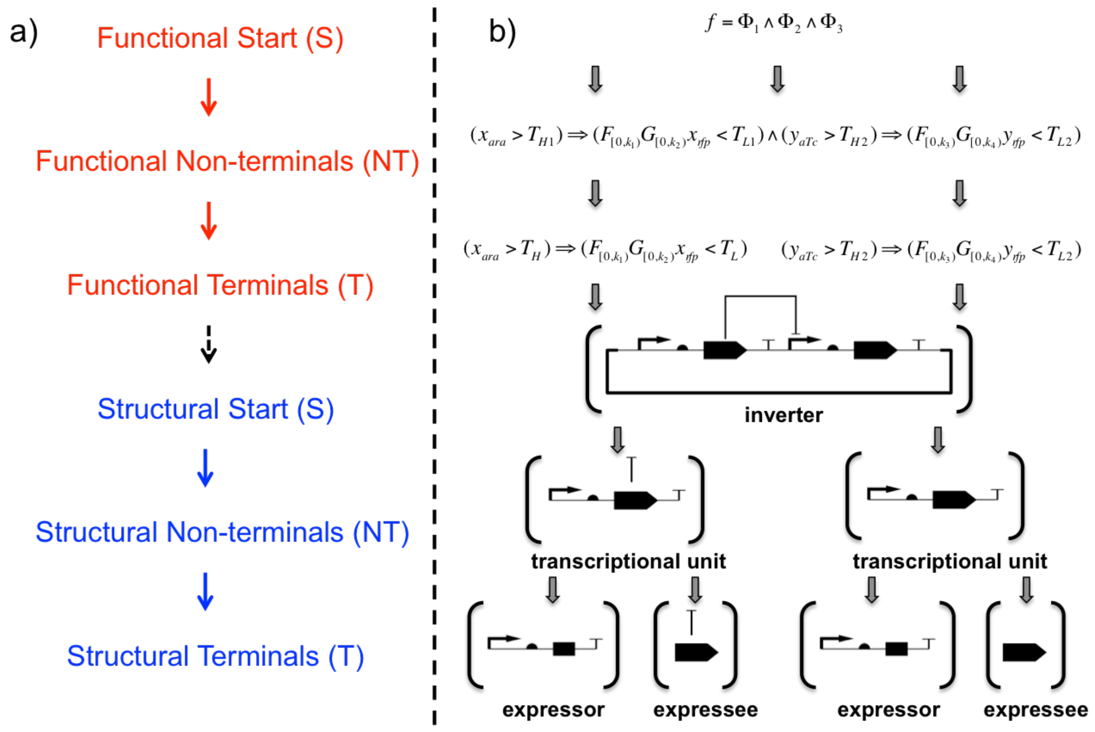

a)
Functional Start (S)
Functional Non-terminals (NT)
Functional Terminals (T)
Structural Start (S)
Structural Non-terminals (NT)
Structural Terminals (T)
b)
$f = \Phi_1 \wedge \Phi_2 \wedge \Phi_3$
$(x_{ara} > T_{H1}) \Rightarrow (F_{[0,k_1)} G_{[0,k_2)} x_{rfp} < T_{L1}) \wedge (y_{aTc} > T_{H2}) \Rightarrow (F_{[0,k_3)} G_{[0,k_4)} y_{rfp} < T_{L2})$
$(x_{ara} > T_H) \Rightarrow (F_{[0,k_1)} G_{[0,k_2)} x_{rfp} < T_L)$
$(y_{aTc} > T_{H2}) \Rightarrow (F_{[0,k_3)} G_{[0,k_4)} y_{rfp} < T_{L2})$
inverter
transcriptional unit
transcriptional unit
expressor
expressee
expressor
expressee

Figure 6: (a) Grammar in *Phoenix* going from a functional non-terminal to a structural terminal (left). (b) A symbolic representation of an example design decomposition.

transcriptional regulation and the types of structural genetic elements that are required for transcription, translation, and transcriptional regulation. Our grammars recognize four types of genetic elements: transcriptional promoters, ribosome binding sites (RBSs), coding sequences for transcription factors (CDSs) and transcriptional terminators. The use of specialized RBSs, called BCDs[71] is assumed throughout to shield the impact of adjacent CDSs on expression. The grammar is used to identify sets of these genetic elements on both strands and determine if candidate sets of these elements exclusively produce 1 or more transcription factors (i.e. no partial production of bio-molecules is allowed). Once it is determined that a set of genetic elements is valid, the grammar identifies subsets of these structures that produce single transcriptional regulatory ‘arcs’ and identifies single transcriptional units within those structures. Finally, the grammar identifies sets of components within a transcriptional unit that enable the expression of a transcription factor (called *EXPRESSORs*) and the coding sequences that get expressed as proteins (called *EXPRESSEEs*). The decomposition is done this way such that the leaves of the tree (*EXPRESSORs* and *EXPRESSEEs*) represent sets of elements that are responsible for independent measureable parameters in the enzymatic models for these structures. Specifically, *EXPRESSORs* are the ordered sets of parts that contribute to the expression parameter of a transcriptional unit and *EXPRESSEEs* are the elements are ultimately translated into proteins in the cell which have a degradation, regulation parameter, and sometimes a small molecule interaction parameter.

4.2 Mechanistic Modeling of Genetic Regulation

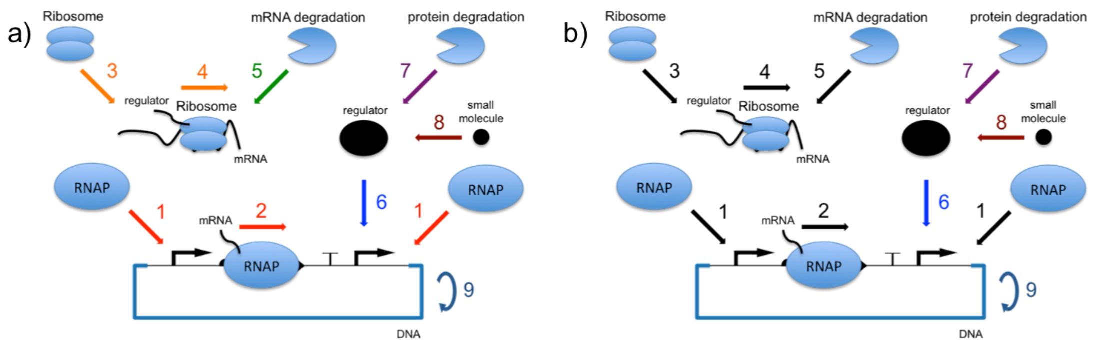
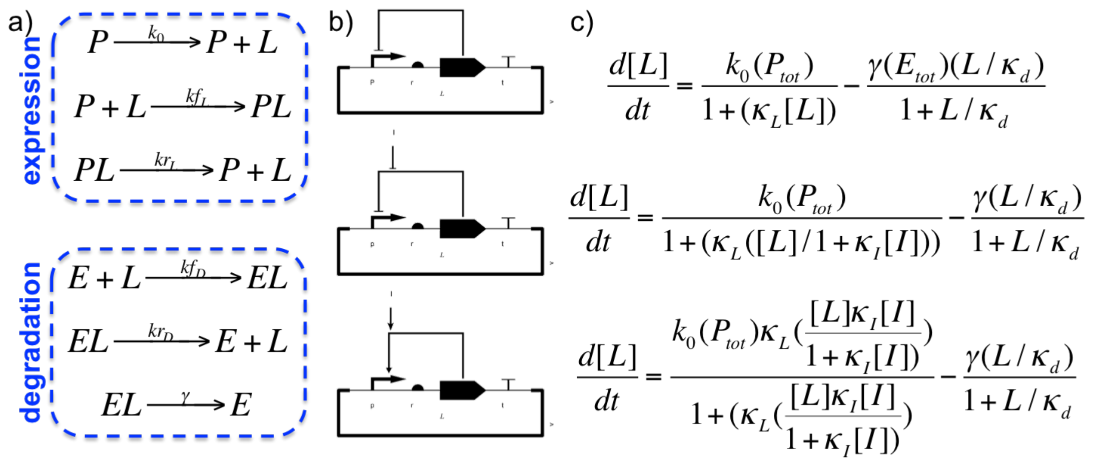


Figure 7: As opposed to performing measurements to determine rate constants specific to single interactions for expression (a), *Phoenix* 'black-boxes' these expression interactions into a single composite constant for expression.

The decomposition of transcriptional GRNs is informed by the modeling paradigm used to understand the mechanistic interactions driving these networks. In *Phoenix*, a modeling paradigm from other current work is adopted that black-boxes protein expression to one rate constant that represents all reactions required to produce a protein. This includes transcription initiation, elongation, and termination; translation initiation and elongation;

mRNA degradation; and DNA replication. This composite expression constant is the only constant measured for *EXPRESSORs*.



$$P \xrightarrow{k_0} P + L$$
$$P + L \xrightarrow{kf_I} PL$$
$$PL \xrightarrow{kr_I} P + L$$
$$E + L \xrightarrow{kf_D} EL$$
$$EL \xrightarrow{kr_D} E + L$$
$$EL \xrightarrow{\gamma} E$$

$$\frac{d[L]}{dt} = \frac{k_0(P_{tot})}{1+(\kappa_L[L])} - \frac{\gamma(E_{tot})(L/\kappa_d)}{1+L/\kappa_d}$$

$$\frac{d[L]}{dt} = \frac{k_0(P_{tot})}{1+(\kappa_L([L]/1+\kappa_I[I]))} - \frac{\gamma(L/\kappa_d)}{1+L/\kappa_d}$$

$$\frac{d[L]}{dt} = \frac{k_0(P_{tot})\kappa_L(\frac{[L]\kappa_I[I]}{1+\kappa_I[I]})}{1+(\kappa_L(\frac{[L]\kappa_I[I]}{1+\kappa_I[I]})} - \frac{\gamma(L/\kappa_d)}{1+L/\kappa_d}$$

Figure 8: (a) The core models in *Phoenix* consider both expression and degradation kinetics (b) Three example regulatory networks (c) Mass action kinetic models for the three example networks with the *Phoenix* modeling approach.

More individual reactions are measured for *EXPRESSEEs*: protein degradation; protein expression of a promoter as a function of the quantity of its corresponding *EXPRESSEE*; and the degree with which a small molecule inducer in the environment impacts this relationship. Each one of these reactions are measured individually in the *Phoenix* environment with the addition of standard testing components.

4.3 Insertion of Testing Components

Specifications decomposed with the *Phoenix* grammars result in a tree of genetic components that are still not visible to any form of measurement. In the *Phoenix* tool, we use single-cell fluorescence as the mechanism of functional measurement. Since this technology has single-cell resolution and can measure a large number of fluorescent reporters simultaneously, it is an ideal choice to measure the signal produced by genetic regulatory networks.

When using the production of fluorescent reporters as probes for the regulatory function, there arises a question of how to position this probe in the construct being measured. There is a variety of options for this probe including using a separate transcriptional unit driven by the same promoter, bi-cistronic transcriptional unit designs, and fluorescent-fused regulators. In *Phoenix*, fluorescence-fused regulators are the method of choice since they are the most direct measurement of a regulator's expression and share the same degradation properties in a fused complex.

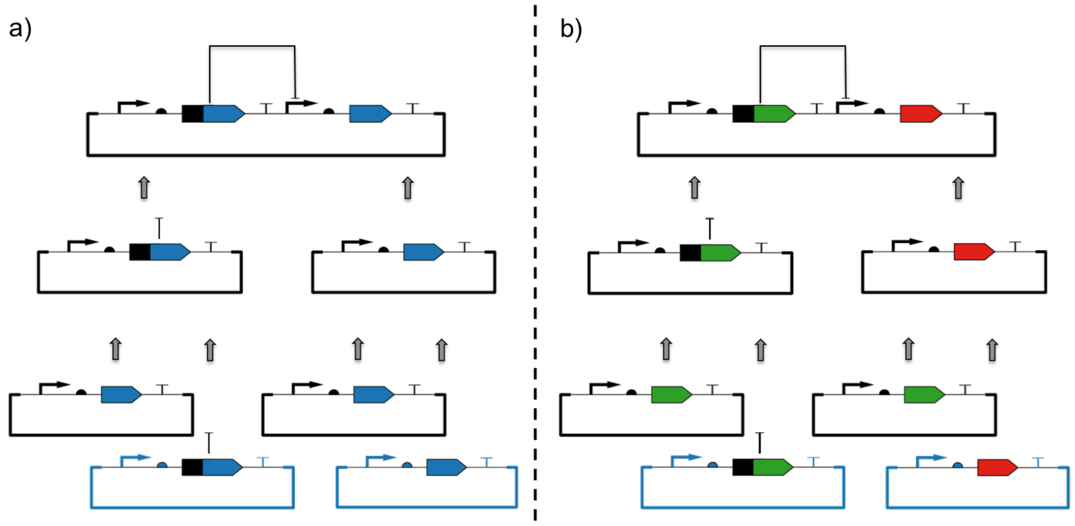


Figure 9: (a) After design decomposition, testing components (blue) must be added to the decomposed design components (black). (b) Once the position of fluorescence-fused regulators is determined, a fluorescent protein must be selected from a set of available fluorescent proteins with respect to a particular measurement machine.

Phoenix applies this probing strategy throughout, but in cases where multiple fluorescence-fused regulators appear in one construct (which is true of all functional terminals and thus also larger designs), it must be determined which fluorescent protein to use in each fusion. Since there currently exist no algorithms for selecting sets of fluorescent proteins, we developed an algorithm for selecting fluorescent proteins based upon minimizing spectral overlap across fluorescence emission spectra.

Once the fluorescent proteins are selected for the construct at the functional root, these selections are propagated down the design tree. For some constructs, all of the elements necessary for expressing a fluorescent protein signal for measurement are not yet present. *Phoenix* solves this problem by assigning default testing elements with pre-measured properties where necessary.

In the final set of operations for adding testing components, *Phoenix* instructs the creation of plasmids to be used as controls for measuring the properties of *EXPRESSEEs*. These plasmids have two transcriptional units - one controlled by the *EXPRESSEE*'s promoter to express a fluorescent protein and one to express a gene for the environmentally-inducible control of the promoter driving the expression of the *EXPRESSEE*. This allows for controllable measurements of how an *EXPRESSEE* regulates its corresponding promoter.

4.4 Part Assignment

Upon the creation of the abstract design tree, the sub-structure of genetic elements at each node is determined, but the DNA sequences of each element are not sequence-assigned. The process of filling in this AGRN to create a sequence-complete GRN is called part assignment.

There are some existing tools for part assignment[102], but they are restricted to logic-based abstractions and do not make considerations for AGRN arrangements, or default testing sequences. Upon initial design decomposition and testing element addition, a partial sequence assignment is performed for promoter-regulator pairs to assure that only orthogonal pairs of regulators and promoters are used in each functional terminal. In this initial partial assignment step, RBSs, terminators and vectors are left unassigned. The user is instructed to multiplex all of their eligible RBSs, terminators, and vectors in the terminal testing constructs to maximize functional diversity. This is done because the functional diversity represents the ‘performance library’ for the target design and a larger library implies more possible combinatorial compositional designs.

5 Interactive DNA Assembly

5.1 DNA Assembly Background

Once a full set of GRNs has been assigned, it must be determined how to clone the designs efficiently. Cloning technology has increased in scale and complexity since it was invented 40 years ago. In the past decade, a number of DNA assembly methods have emerged[42,48,62,81,89,99] that include restriction-ligationbased and homologous recombinationbased cloning systems, and many follow the design model of assembling genetic 'constructs' from genetic 'parts'[41]. While the precise definitions of these terms have been debated[15], the consensus is that parts are DNA segments and constructs are ordered sets of these parts. Given this design model, two fundamental questions arise: First, how should we identify and select parts to create the desired functional genetic constructs? Second, once these constructs have been selected, how do we physically assemble them? The following work formalizes this second question and provides algorithms that address experimental realities to improve the speed, modularity and experimental efficiency of this process for state-of-the-art DNA cloning techniques.

Cloning-based assembly approaches can be broadly classified into binary assembly techniques, where two DNA parts are assembled in one cloning step, and multi-way (one-pot) assembly techniques, where two or more parts are assembled in one step. Generally speaking, multi-way assembly methods are faster because they can minimize cloning steps and can be exploited to leave no assembly artifacts in a completed construct, while binary assembly methods typically require more cloning steps, but in some cases utilize specific cloning sites to allow a simpler standardization of part composition and modularity.

Previous work[25,37] detailed hierarchical assembly algorithms for binary assembly[90] but

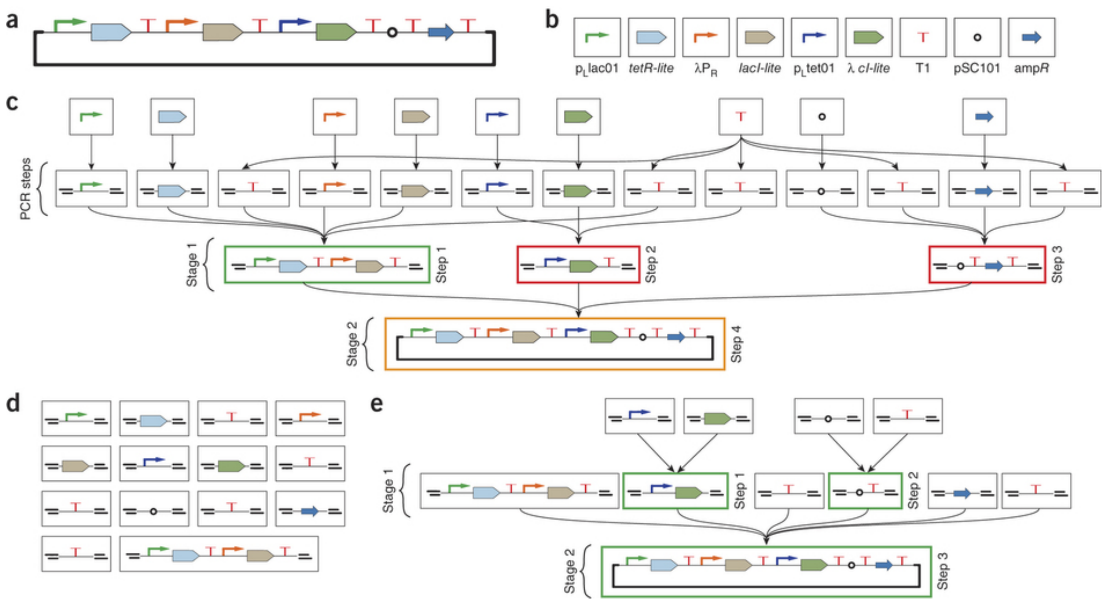


Figure 10: The graphic symbols are composed using Pigeon[23] (http://www.pigeoncad.org/) from SBOL[45] (http://www.sbolstandard.org/) visual images to denote part types. (a) The repressilator. (b) Starting library consists only of template DNA. (c) A plan for assembling the repressilator given b requires 13 PCRs, 4 steps and 2 stages. Two steps fail (steps 2 and 3; red boxes), one step succeeds (step 1; green box) and the dependent step in the second stage cannot be attempted (orange box). (d) The updated library contains basic parts and intermediate parts with specific overhangs from c. (e) An optimized plan, in which all steps succeed (green boxes), is generated with no PCRs, three steps and two stages.

lacked formulation to address more modern multi-way assembly techniques. There are few automated tools to exploit the high degrees of modularity and reuse for multi-way techniques, and no tools for producing complete assembly plans. Some approaches[52] detail the process of automatically selecting oligonucleotides and analyze trade-offs between cloning and gene synthesis for multi-way assembly; however, they do not optimize cloning steps and stages.

This section describes a method for performing optimizations on intermediate cloning step selection and part junction selection for any number of target constructs while considering a library of existing parts for reuse. My collaborators and I show that for sets of thousands of variants of multiple types of contemporary genetic constructs and a large set of constructs from the literature[26,27,44,64,69,91,96,97], our program outperforms unoptimized solutions (*P(z)* < 0.001), and we then experimentally verify a small subset of these optimized solutions by reconstructing ‘genetic counter’ and ‘repressilator’ constructs. This work also details, to our knowledge, the first automated cloning workflow in which experimental outcomes may be directly fed back into the software to recalculate an alternative assembly plan.

The algorithms presented are housed in an online web application called Raven that produces full assembly plans in human- and computer-readable instructions and graphical Synthetic Biology Open Language (SBOL)-compatible images for each of the supported assembly methods[42,48,62,81,90,99].

In this work, I break the problem of assembling a set of DNA constructs with a selected cloning technique into three main subproblems. First, the algorithms determine an optimized hierarchical cloning plan for assembling a set of constructs. For this part, Raven uses dynamic programming to reduce the computational time it takes to solve the large problem of selecting intermediate cloning steps for a set of target constructs into smaller sub-problems. Because the heuristic scores for assembling a specific intermediate are assumed to remain constant regardless of previous or future cloning steps, once an optimized heuristic solution for an intermediate is found, the solution is reused if the candidate intermediate step is encountered again. For constructs that share parts internally or share parts with other target constructs, sharing of assembly intermediates can reduce the total step count considerably, and, if many steps can be done in parallel, the number of cloning stages can also be minimized. The cloning step solution comprising all the stages and steps necessary to build the target constructs under consideration constitutes a hierarchical ‘assembly graph’.

Second, the algorithms determine an optimized set of part junctions (hereafter referred to as ‘overhangs’) required to perform the selected cloning steps. Overhang assignment, which aims to minimize overhang generation cost (hereafter ‘PCR steps’), is determined in three steps. First, the requirements for overhang uniqueness based on the selected assembly method are determined (for example, all parts must have unique pairs in each individual cloning step). Second, for modular overhang assignment, overhang pair sharing is maximized for all cloning steps where sharing can eliminate extra steps and the total number of unique overhang sequences is minimized. The second step is skipped for some assembly methods because it is assumed that all overhangs are either the same or all are unique. Third, after all overhang pairs have been determined, the existing parts library is used to map abstract overhangs to DNA sequences, maximizing library reuse with a constrained Cartesian product (**Figure 15**).

Third, based on the cloning steps and their overhangs, oligonucleotides for PCR are automatically designed (**Figure 10a-c**). For all PCR steps, primers are designed on the basis of part, overhang sequence, direction (forward or reverse strand) and assembly method. The primer designs are optimized for length and melting temperature, but other complex optimizations are not considered. For more sophisticated primer designs, we provide outputs compatible with existing state-of-the-art primer design software[52]. The summary of all cloning steps, PCR steps and oligonucleotide designs constitutes a complete 'assembly plan'.

Following Raven's assembly instructions, a user might encounter some assembly steps that fail. The user can then mark each step in the plan as successful, failed or not attempted (as a result of step failures in an earlier stage) (**Figure 10c**) to recalculate an alternative assembly plan. The parts from the successful steps are added to the library and failed steps are forbidden from appearing in a new plan (**Figure 10d,e**). The interactive refinement of an assembly plan is meant to be independent of specific protocols and reaction conditions and can complement troubleshooting specific reactions in a preliminary plan. This process continues iteratively until all target parts are assembled. As this algorithm relies on heuristics in many locations, we cannot make any claims that it is optimal. However, we can prove that the solutions are correct in linear time as a function of the number of intermediates.

Construct Source	No.	Un-Optimized Solutions			Raven Solution		
		RFC10	RFC94	Gibson	RFC10	RFC94	Gibson
Bonnet *et al.*[26]	6	6.09|45.7|21	3.83|47.7|58	2.67|12.3|34	5|43|21	3|39|54	2|8|34
		0.80|1.24|0	0.56|2.07|0	0.59|2.05|0	0.08|0.02|0.5	0.07|<0.01|0	0.13|0.02|0.5
Bonnet *et al.*[27]	13	6.81|80.9|23	4.41|116|127	3.27|32.9|55	5|73|23	3|75|104	2|24|55
		0.64|2.61|0	0.52|4.50|0	0.50|2.73|0	<0.01|<0.01|0.5	<0.01|<0.01|0	<0.01|<0.01|0.5
Friedland *et al.*[44]	5	8.97|89.7|27	4.99|101|125	4.99|33.3|50	6|74|27	3|49|62	2|19|50
		0.87|3.49|0	0.59|4.05|0	0.59|2.91|0	<0.01|<0.01|0.5	<0.01|<0.01|0	<0.01|<0.01|0.5
Lou *et al.*[64]	191	7.81|482|90	6.15|869|420	4.13|477|449	5|624|90	3|401|266	2|381|449
		0.59|23.2|0	0.54|34.4|0	0.36|10.1|0	<0.01|1*|0.5	<0.01|<0.01|0	<0.01|<0.01|0.5
Moon *et al.*[69]	15	8.70|106|37	4.82|135|140	3.66|38.0|84	6|102|37	3|71|90	2|22|84
		0.99|3.03|0	0.58|4.87|0	0.60|3.62|0	<0.01|0.09|0.5	<0.01|<0.01|0	<0.01|<0.01|0.5
Suiti *et al.*[91]	23	7.39|125|19	4.87|224|255	3.60|71.4|97	5|121|19	3|171|234	2|55|97
		0.63|4.38|0	0.48|7.51|0	0.59|3.99|0	<0.01|0.18|0.5	<0.01|<0.01|0	<0.01|<0.01|0.5
Tabor *et al.*[96]	6	7.76|49.2|12	4.61|76.9|98	3.49|23.1|26	5|39|12	3|39|51	2|14|26
		0.82|3.56|0	0.58|3.47|0	0.56|2.36|0	<0.01|<0.01|0.5	<0.01|<0.01|0	<0.01|<0.01|0.5
Tamsir *et al.*[97]	14	6.30|60.6|15	4.24|84.1|93	2.89|28.8|53	4|56|15	3|59|69	2|21|53
		0.53|2.27|0	0.50|4.24|0	0.52|2.71|0	<0.01|0.02|0.5	<0.01|<0.01|0	0.04|<0.01|0.5
ALL	273	9.28|969|204	6.17|1566|907	4.36|717|844	6|1092|204	3|879|797	2|544|844
		0.85|23.8|0	0.48|24.7|0	0.51|13.2|0	<0.01|0.99*|0.5	<0.01|<0.01|0	<0.01|<0.01|0.5

Table 2: Raven-optimized and average unoptimized assembly scores for constructing plasmids from the literature. Number of constructs (no.) considered in each set is shown. Numbers in each set of columns refer to cloning stages, cloning steps and PCR steps, respectively. Unoptimized solutions are represented by averages (top of each row) and s.d. (bottom of each row). Raven solutions are reported (top), along with the probability, *P(z)*, of selecting this solution randomly using a statistical z-test (bottom). P values are calculated assuming a normal distribution of assembly outcomes.

5.2 *In silico* assembly of thousands of constructs

To determine the quality of Raven's assembly plans, me and my colleagues compared our solutions against unoptimized solutions for each data set by randomly sampling the assembly plan space for each set of constructs under consideration. For both unoptimized solutions and Raven solutions, we assumed no preexisting library of parts except template DNA and constrained our assembly calculations such that a maximum of six parts could be assembled per reaction for one-pot assemblies, as reactions with more parts show low efficiency.

First I considered several published sets of complex genetic constructs covering a variety of sizes, types and architectures[26,27,44,64,69,91,96,97]. For each of these sets, we determined optimized and unoptimized solutions for BioBricks (BioBricks Foundation request for comments (BBF RFC) 10), MoClo (BBF RFC 94) and Gibson assembly methods.

Assembly solutions were scored in terms of cloning stages, cloning steps and PCR steps, and the Raven solutions are compared to average unoptimized solutions (**Table 2**) and the best unoptimized solutions (**Table 3**). Raven's solutions were significantly better than unoptimized solutions for assembly stages ($P(z) < 0.01$) for all three assembly methods for nearly all construct sets. Raven's MoClo solutions were significantly better for both cloning steps and PCR steps ($P(z) < 0.01$). Raven's solutions had significantly fewer cloning step solutions for Gibson ($P(z) < 0.01$) for all construct sets, and in only one BioBricks solution did the unoptimized plans result in fewer steps. However, as Raven's strongest scoring heuristic is cloning stages, when selecting the best assembly plan, Raven allows additional steps in favor of fewer stages. Similarly, the summary of all sets has a better cloning-step solution for BioBricks because of the inclusion of the aforementioned set, which contains by far the greatest number of constructs of the considered construct sets. For BioBricks and

Gibson cloning, the number of PCRs is not optimized, so all Raven answers are equivalent to those of the unoptimized solution.

Next, to demonstrate the power of Raven solutions on an even larger scale, I used Eugene[73] to generate a set of 1,000 or more variant constructs for five separate types of constructs: DNA invertase cascade (DIC) counters, toggle switches, repressilators, transcriptional NOR gates and invertase NOR gates (**Figure 11a**). Because it is common for large constructs to need tuning to achieve function, these sets contain variants to represent a spectrum of possible function and provide many opportunities to share intermediates.

To determine one unoptimized solution for each of these sets, a script randomly selected 500 constructs and calculated an unoptimized, one-pot hierarchical assembly graph. We repeated this experiment 1,000 times for each of the five designs to get a distribution for each type of design and found that Raven's algorithms were able to select assembly graphs that require significantly fewer cloning steps than the average unoptimized graphs for all five designs ($P(z) < 0.001$) (**Figure 11b**). Because these data sets were made from combinatorial part substitutions, there exist many opportunities to share cloning intermediates and assembly vectors using modular overhangs. My team observed that our modular overhang assignment solutions required significantly fewer PCRs than unoptimized solutions for overhang assignment for each of the five design types ($P(z) < 0.001$) (**Figure 11c**).

Finally, I determined how Raven's solutions performed as a function of the number of constructs under consideration. We repeated the *in silico* experiments for the five design types for variable numbers of constructs. I found that Raven's solutions significantly outperformed the unoptimized solution spaces for both cloning steps and PCR steps (**Figure 11d**) at a small scale of 5 constructs (P(z) < 0.001) as well as at a larger scale of 500 constructs ($P(z) < 0.001$). As the number of constructs under consideration increases exponentially, Raven's solutions for both hierarchical assembly and overhang assignment also improve

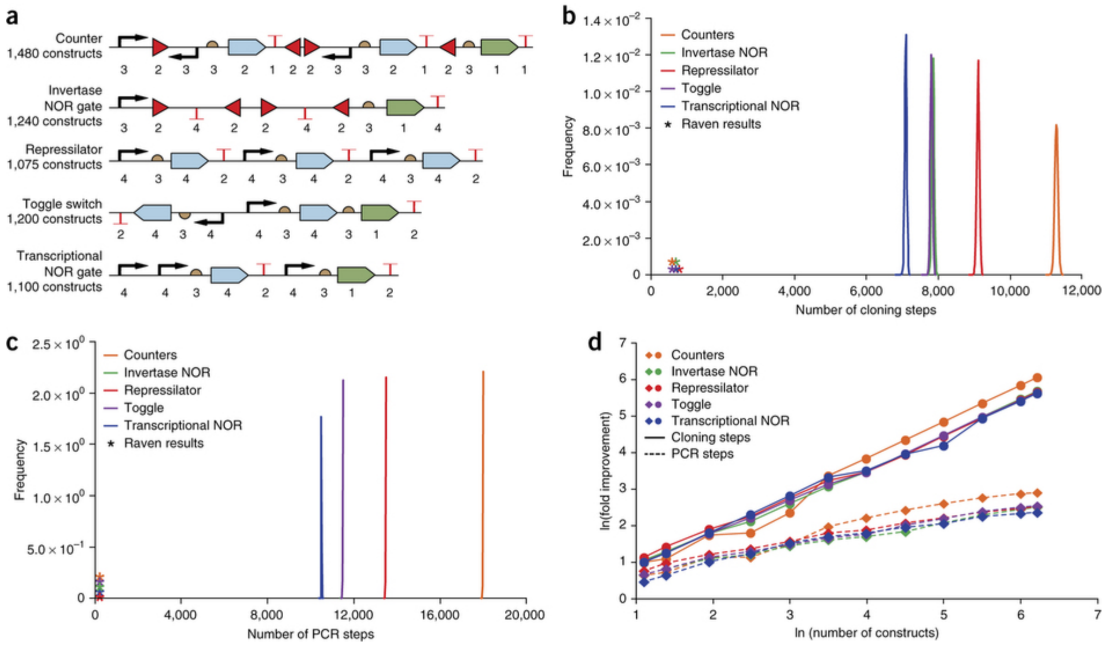


Figure 11: (a) SBOL visual representations of the DIC counter, invertase NOR gate, repressilator, toggle switch and transcriptional NOR gate constructs, indicating the number of parts we sampled at each position and the total possible construct variants after application of Eugene rules. (b) Cloning steps required for MoClo assembly of a 500-construct subset of each set of 1,000 or more constructs from a. Asterisks represent the Raven solution; other points represent unoptimized cloning step solutions. (c) PCR steps required for MoClo assembly given the cloning step solution in b. Asterisks represent the Raven solution; other points represent random PCR step solutions. (d) The fold improvement of Raven's solution compared to unoptimized solutions in b,c as a function of construct quantity. Raven's solutions improve as the number of constructs per assembly plan increases. PCR steps (dashed) and cloning steps (solid) are shown separately.

exponentially compared to unoptimized solutions (**Figure 11d**).

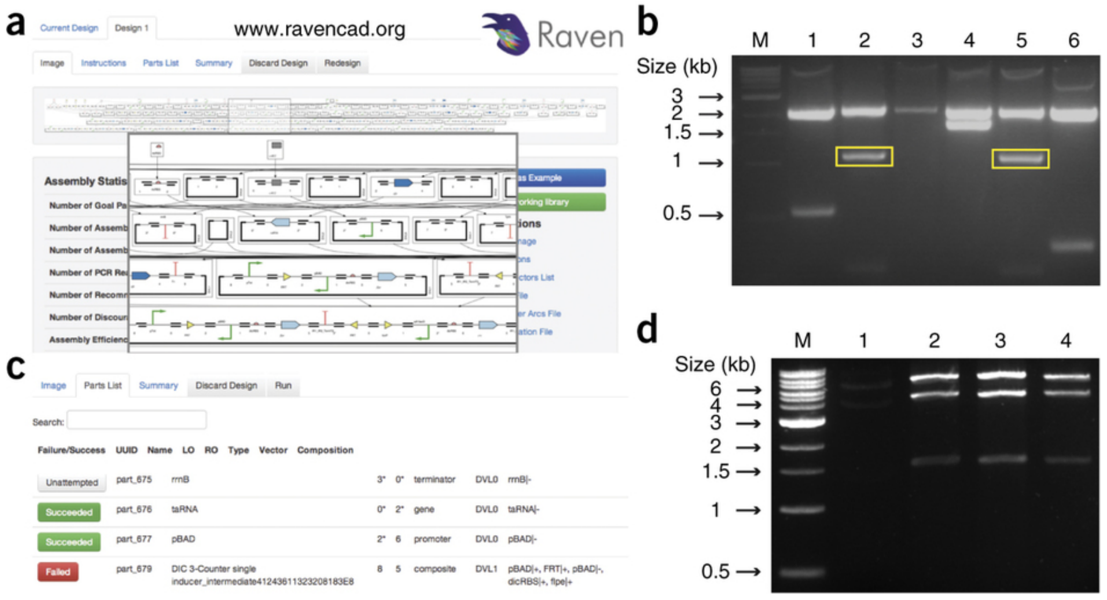
5.3 Interactive Assembly of Genetic Constructs

To highlight Raven's ability to utilize an existing library of constructs, I used it to calculate an assembly plan for six repressilator[40] constructs using an existing library with an existing overhang schema. The constructs were designed based on previously published schema using the CIDAR (http://www.cidarlab.org/) MoClo library as a resource. Design constraints allowed only up to four parts per cloning step, as opposed to six. The assembly plan for these six constructs required 17 assembly steps, 2 assembly stages, 0 PCR reactions and 23 shared parts, and we successfully constructed two constructs without modification to this plan (**Figure 25**).

I then selected a subset of the constructs from Friedland *et al.*[44] (representing some of the largest and most complex constructs in the sets) and constructed them using Raven. I used a MoClo assembly plan (BBF RFC 94) for the DIC counter constructs, assuming a library of only template DNA and cloning vectors. The Raven-designed oligonucleotides from the assembly plan were used to amplify parts using the original constructs as template (**Figure 17**), overhang sites were chosen from a preselected set of 4-bp modular scars and it was assumed that all cloning steps would have equivalent cloning efficiency. The assembly plan for all four constructs required 29 steps, 3 stages and 34 PCR steps (**Figure 12a, Figure 19**). I implemented this preliminary assembly plan as specified by the human-readable instructions that Raven generated, using standard reaction conditions (**Figure 12a**).

This initial plan was not successful. However, Raven has four primary mechanisms for interactively modifying assembly plans to circumvent unsuccessful cloning steps. First, Raven can detect undesirable restriction sites that can be removed with PCR. Second, intermediate clones flagged for expressing undesirable genes (such as the *flpe* recombinase) or other traits (**Figure 12b**) can be biased for or against appearing in an assembly plan (**Figure 23**). Third, default cloning vectors assigned to each assembly stage based on each assembly method may be substituted. Finally, cloning efficiency as a function of number of parts assembled per cloning reaction may be modified from default equivalent-efficiency values (**Figure 24**).

In cases where users have already started a large assembly but get stuck on unforeseen challenges, they can use the Raven redesign feature (**Figure 12c**) to calculate a new plan. When using this feature, Raven automatically adds the successful parts into the library and



Failure/Success	UUID	Name	LO	RO	Type	Vector	Composition
Unattempted	part_675	rrnB	3*	0*	terminator	DVL0	rrnB|-
Succeeded	part_676	taRNA	0*	2*	gene	DVL0	taRNA|-
Succeeded	part_677	pBAD	2*	6	promoter	DVL0	pBAD|-
Failed	part_679	DIC 3-Counter single inducer_intermediate41243611323208183E8	8	5	composite	DVL1	pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+

Figure 12: Interactive Assembly (a) An initial assembly plan in the Raven UI. (b) A *SpeI* restriction analysis for Level 1 cloning intermediates (1-6) with expected bands at 0.5kb, 1.7kb & 0.2kb, 2.1kb, 1.7kb, 1.5kb and 2kb, and 3.4kb, respectively, with a 2.1kb vector band. Incorrect bands seen for lanes 2 & 5. (c) In the Raven UI redesign tab, failure of these intermediates and success of all other intermediates is reported and a new plan is generated. (d) A *PstI* restriction analysis for the complete genetic counter constructs should be 7kb, 4kb & 1.6kb

forbids failed intermediates from appearing in the alternative solution. When I got stuck on the first plan for the counters, we used a redesigned solution, which required seven steps, two stages and two PCR steps (**Figure 20**). This plan reused the four successful intermediates from the initial plan and split up the two unsuccessful intermediates into smaller intermediates. One of these intermediates was also unsuccessful, so a second redesign with the same cost was implemented that succeeded in the cloning of all intermediates (**Figure 21**).

This third plan, although successful for creating all intermediates, was not successful for cloning the final constructs. This was because BBF RFC 94 assumes the use of high-copy plasmids for all cloning steps. In this case, since it is critical that recombinases are not expressed, it was problematic to clone the final counter constructs into high-copy plasmids owing to leaky promoter behavior. To address this, I forced an extra cloning stage by requiring the construction of larger intermediates and assigned a pBAC for the final cloning stage (**Figure 21**).

Using this plan, all cloning intermediates were constructed and used to build the final constructs successfully (**Figure 12d**). Several of the intermediates incurred mutations as a result of cloning artifacts, but these were located at internal part junctions and the flanking junctions needed for future steps remained unaltered. Therefore, Raven cannot guarantee the production of an exact target sequence; in vivo recombination events are difficult to predict and outside the scope of the assembly plan. Moreover, as long as the necessary restriction sites and part junctions remain intact, the Raven plan remains valid.

5.4 Multiplex Assembly of Genetic Variants

The current paradigm in the cloning community is to define the complete sequence of target constructs before assembly, and set up reactions with only one part in each cloning position. However, in instances where function diversity of genetic constructs is desired, it can be advantageous to perform cloning reactions where instead of putting single defined sequences in each position of a target clone, a library of variants is used. This style of cloning is called multiplexing cloning and can be used to create DNA construct variants.

In theory, if equimolar concentrations of all variants in a multiplex cloning reaction are present, one should expect there to be equal representation of variants among the screened clones. To confirm this for our cloning reactions, we performed experiments for a transcriptional expression cassette, where we multiplexed functionally diverse variants for the promoter, RBS, and terminator positions for up to 8 different variants. We then screened the number of clones necessary to theoretically pick at least one of each variant assuming an equal population. We found there to be no detectable bias for any particular sequence among this set.

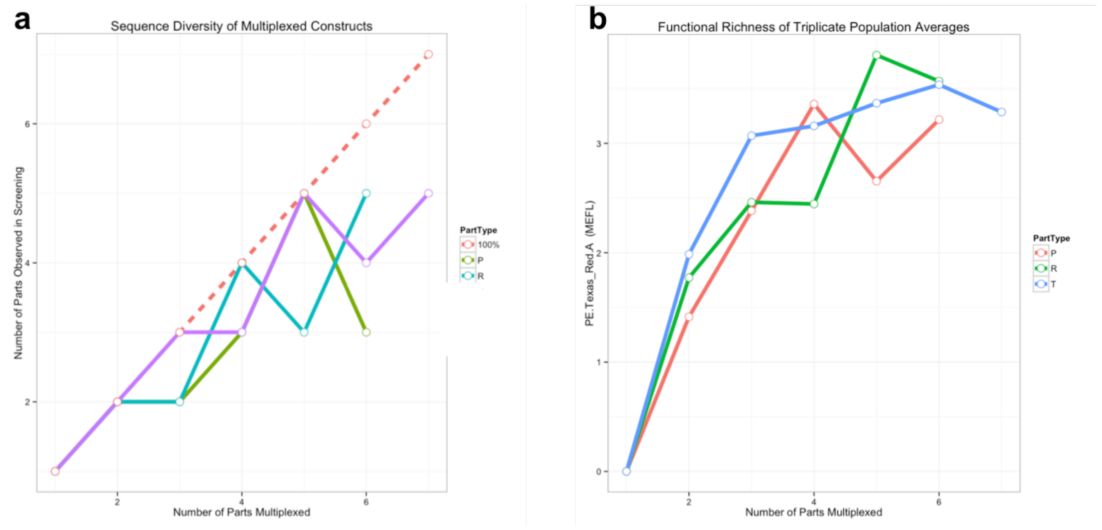


Figure 13: (a) The number of sequences observed in sequence screening as a function of the number of parts multiplexed. In all cases, the theoretical number of colonies needed to encounter at least one of each sequence was screened. (b) Functional diversity value as a function of the number of parts multiplexed for the promoter, RBS, and terminator position.

Since we multiplex primarily to obtain diverse functional outcomes, it needed to be determined if the expression profiles across clones produced diverse expression profiles.

To determine how functional richness was affected by the number of parts multiplexed, we measured the expression profile of the variants from each multiplex reaction to measure the functional richness[70]. We found that generally as the number of parts multiplexed increased, the functional richness increased.

5.5 Complexity Analysis

The following complexity analysis shows that the algorithms me and my colleagues developed for partitioning a goal part into intermediates and assigning overhangs to each parts is both efficient and scalable (**Figure 14**).

5.5.1 Multi Goal Part Algorithm

This section describes the run time of the multi-goal-part algorithm. Here, I describe the worst-case computational scenario for producing an assembly graph. We make no assumptions about biologically informed compositional rules for genetic constructs, which could reduce the computational cost of calculating an assembly graph - compositional rules can be captured using required, recommended, discouraged, and forbidden parts, the use of which is described in our online documentation at ravencad.org. The multi goal part algorithm is an extension of the algorithm presented by Densmore *et al.*[37] and uses a similar dynamic programming scheme to reduce complexity. The worst case for run time is when there are no parts in the library.

Let there be *n* goal parts in an assembly with each goal part consisting of an average of *m* basic parts. The algorithm begins by making a call to the determineSlack() helper routine, which calls createAsmGraph_sgp() once for each goal part to determine the size of the largest graph, the slack.

The first step in createAsmGraph_sgp() is to determine how to partition a part. Supposing that a number of parts can be assembled in each reaction, then for a goal part of size m, is assembled from $ceil(\frac{m}{a})$ number of intermediates.

If the number of intermediates required to assemble a larger intermediate of size b exceeds $a+1$, then there would be at most, b choose a ways to partition the larger intermediate into smaller intermediates. Exploring the full combinatorial space of the partitions exceeds exponential in complexity. Using a least squares approach encapsulated in the getPartitions() helper method, finding the appropriate partitions for any goal part or intermediate can be done in constant time assuming a reasonable number of forbidden parts, that is the number of forbidden parts is less than the Bell number for a set of size m.

Each of the smaller intermediates in turn, would need to be assembled $ceil(\frac{m}{a})$ intermediates in turn. And so createAsmGraph_sgp() would need to make $log(m)$ recursive calls to createAsmGraph_sgp() to create an increasingly smaller subgraph. The cost for these recursive calls can be written in the form $T(n) = a * T(\frac{m}{a}) + c$, where c is the constant cost of finding the partitions and combining several subgraphs, both of which are effectively constant time operations. Assuming According to Master's Theorem, these recursive calls are $O(m)$ in complexity.

And so in total, it costs on the order of nm operations to determine slack. Next, the multi goal part algorithm makes n calls to createAsmGraph_sgp(), incurring yet another nm operations. So overall, the multi goal part algorithm is $O(mn)$ in complexity.

5.5.2 Overhang Assignment Algorithm

This section presents the run time analysis for MoClo overhang assignment. The algorithm for overhang assignment uses a 3 pass, dynamic programming approach. In the first pass, metadata is added to each node to enforce the MoClo overhang rules -parent and child nodes, and adjacent nodes (the children of each node are ordered) have the same overhang. The second step minimizes the number of overhangs used and the number of vectors used. The final step maps overhangs assigned in the second step to overhangs that exist in the library.

In first step of overhang assignment, a recursive helper method is called for each of the n graphs, which each contain on average m leaf nodes. The helper method visits each node once, storing metadata and assigning overhangs at each visit, expending $n * \sum_{i=0}^{log_p m} p^i$ operations where p is the maximum number of parts that can be assembled in a single assembly step operations. Next for each graph, every node is stored in a hash that stores the maximum level that the node's overhang impacts; it takes $log_p m$ operations to determine the maximum level for each node. And so, the first step of our overhang assignment algorithm costs $n * (\sum_{i=0}^{log_p m} p^i) + n * m * log_p m$ operations.

The second step of the overhang assignment algorithm iterates over all basic part nodes for each graph twice, once in the forward direction and once in the backwards direction, costing m operations per graph. In each iteration, an incremented value is assigned to the left and right overhang of a basic part. Depending on the type of the basic part, the assigned values are set aside in typeOHHashLeft and typeOHHashRight for assignment to parts of the same time. To ensure that the restored values give valid assignments, we iterate over the neighbors of the basic parts, removing any overhangs that would give an invalid assignment, requiring a total of m operations. So overall, the second step costs a total of nm^2 operations.

The final step of the overhang assignment maps abstract overhangs to concrete overhangs that exist in a user's part library. In the third step of the overhang assignment, abstract overhangs are first linked to all possible concrete overhangs; this requires iterating over all basic part nodes in all graphs, as well as all parts in the library, l, adding up to $nm + l$ operations. Using this information, a constrained Cartesian product is performed to determine the best assignment of concrete overhangs. In the worst case, no overhang pairs are reused and overhangs are used just once, and so there would be $2mn$ overhangs that need to be assigned. Given l parts in the library, each abstract overhang would have roughly $\frac{l}{nm}$ concrete options. The Cartesian product produces at most $(\frac{l}{nm})^n m$ valid solutions. The next iteration occurs over each valid solution to score for the best. A final traversal is then necessary over all graphs to assign the overhangs to the graph. In total, $(nm + l) + (\frac{l}{nm})^n m + n * (\sum_{i=0}^{log_p m} p^i)$ operations are required for the third step.

Summing the cost for all three steps, overhang assignment costs $2n * (\sum_{i=0}^{log_p m} p^i) + nm * log_p m + 2nm^2 + (nm + l) + (\frac{l}{nm})^{nm}$ operations. In most situations, $l > m > n$, and so the overhang assignment algorithm has $O(\frac{l}{nm})^{nm}$ complexity. However, in practice there are rarely $(\frac{l}{nm})^{nm}$ valid solutions in the third step, which would require that each basic part in an assembly is unique within the assembly. Raven is designed suited for parallelized assembly plans characteristic of pathway engineering experiments, in which case the number of basic parts would be the main contributing factor, giving a practical complexity of $O(nm^2)$.

5.6 Eugene Rules for 1000+ Construct Sets

A permutation of all possible constructs in **Figure 11** for a specific construct type yields a combinatorially large space. For larger constructs, this space can exceed 1,000 constructs, but not all constructs are functionally valid. Me and my colleagues wished to constrain the construct sets to those which are qualitatively valid, so we use Eugene[24] to limit this combinatorial space for each type of construct. Below are the rules for each individual type of construct.

5.6.1 Counter constructs

```
construct DIC2(Promoter, InvertaseSite, −Promoter, RBS, Gene
    , Terminator, −InvertaseSite, InvertaseSite, −Promoter,
    RBS, Gene, Terminator, −InvertaseSite, RBS, Reporter,
    Terminator);

Rule r(
          ON DIC:

          // Matching Invertase sites
          DIC[1] MATCHES DIC[6]
                    AND
          DIC[7] MATCHES DIC[12]

                    AND

          // Different Invertase sites
          FRT WITH loxP

                    AND

          CONTAINS rbs1 AND CONTAINS rbs2 AND CONTAINS rbs3
);

construct[] lst = product(DIC, strict, 1000);
```

5.6.2 Toggle-switch constructs

```
construct ToggleSwitch(-Terminator, -Gene, -RBS, -Promoter,
    Promoter, RBS, Gene, RBS, Reporter, Terminator);

Rule r(
          ON ToggleSwitch :

          // Rule 1 and 2: Repression Interactions
          ToggleSwitch[1] REPRESSES ToggleSwitch[4]
                    AND
          ToggleSwitch[6] REPRESSES ToggleSwitch[3]

                    AND

          // Rule 3: Different Promoters
          ToggleSwitch[3] NOTEQUALS ToggleSwitch[4]

                    AND

          // Rule 4: We prefer GFP as reporter (optional rule)
          CONTAINS GFP
);

construct [] lst = product(ToggleSwitch, 1000);
```

5.6.3 Repressilator constructs

```
construct Repressilator(Promoter, RBS, Gene, Terminator,
    Promoter, RBS, Gene, Terminator, Promoter, RBS, Gene,
    Terminator);

Rule r(
          ON Repressilator:

          // REPRESSION relationships
          Repressilator[2] REPRESSES Repressilator[4]
                    AND
          Repressilator[6] REPRESSES Repressilator[8]
                    AND
          Repressilator[10] REPRESSES Repressilator[0]

                    AND

          pLux NOTMORETHAN 1
                    AND
          pLtetO1 NOTMORETHAN 1
                    AND
          lambdaPr NOTMORETHAN 1
                    AND
          pLlacO1 NOTMORETHAN 1
```

```
                AND

        rbs1 WITH rbs2 AND rbs2 WITH rbs3
);

construct[] lst = product(Repressilator, strict, 1000);
```

5.6.4 Transcriptional NOR-Gate constructs

```
construct Repressingconstruct(Promoter, Promoter, RBS, Gene,
    Terminator);
construct Reportingconstruct(Promoter, RBS, Reporter,
   Terminator);
construct NorGate(Repressingconstruct, Reportingconstruct);

Rule r(
        ON NorGate:

        // REPRESSES relationship
        Repressingconstruct.Gene REPRESSES
            Reportingconstruct.Promoter

                AND

        // different RBSs
        Repressingconstruct.RBS NOTEQUALS Reportingconstruct
            .RBS

                AND

        CONTAINS GFP

        // different promoters
                AND
        Repressingconstruct[0] NOTEQUALS Repressingconstruct
            [1]
                AND
        Repressingconstruct[1] NOTEQUALS Reportingconstruct.
            Promoter

                AND

        // different Terminators
        CONTAINS T1 /**AND CONTAINS T7**/
);

construct[] lst = product(NorGate, strict, 1000);
```

5.6.5 Invertase-based NOR-Gate constructs

```
construct NorGate(Promoter, InvertaseSite, -Terminator, -
    InvertaseSite, InvertaseSite, −Terminator, −InvertaseSite
    , RBS, Reporter, Terminator);

Rule r(
        ON NorGate:

        NorGate[1] MATCHES NorGate[3]
                AND
        NorGate[4] MATCHES NorGate[6]

                AND

        Bxb1_attB WITH phiC31_attB
);

construct[] lst = product(NorGate, strict, 1000);
```

5.7 Algorithm

5.7.1 Algorithmic Flow

Users may interact with Raven at a variety of levels and degrees of complexity. At the highest level, a user interacts with Raven by supplying a DNA parts library and a set of constructs to be made from that library. Raven takes this input and produces a set of assembly instructions which can be performed by either a human or computer user (**Figure 15a**). After the plan has been attempted a number of times, some number of the constructs and intermediates will be successfully cloned and some target constructions might still be incomplete. The complete and incomplete constructs can be input back into Raven, where a new plan with the updated library is generated. This cycle continues until all target parts have been assembled.

At a lower level, the libraries and target constructs are uploaded into Raven and some subset of the target constructs is selected to be assembled with one of the supported assembly methods (**Figure 15b**). Raven calculates an optimized assembly plan for these constructs using the selected method and produces assembly instructions in the form of an assembly graph and oligonucleotides necessary to implement the assembly plan. These instructions are given in either human-readable or computer-readable format. A user may calculate assembly plans for parts without specifying part sequences, but the oligo designs will be incomplete and these oligos will need to be designed manually.

The algorithms have three major components: hierarchical step optimizations, overhang assignment optimization and oligo-nucleotide design. The hierarchical cloning step plan is the input for the overhang assignment algorithms and the complete graph with assigned overhangs is the input for oligo design (**Figure 15c**). Optimizations are broken into these three chunks to reduce computational complexity and serve as modular peices of the code base; the hierarchical step algorithms are common to all assembly methods, overhang selection is common to some assembly methods and primer design is also common to some methods.

Function 1 and Function 2 are illustrated in the flowchart presented in **Figure 16**. Function 1 is concerned with creating the best single goal part solutions for all of the goal parts in a loop adding the best solution to a growing list that is used to create the next best solution until all graphs have been created. Function 2 deals with building the best single goal part taking into account legal partitions of that goal part into subparts along with how many of those subparts can be put together in any one reaction. This is a recursive calling function where the single goal part is broken into these subparts with are themselves single goal parts to solve. Functions minCostSlack, minCost, determineSlack, and combineGraphs are helper functions for these two functions are not shown for clarity.

5.7.2 Definitions

- Let part *P* have an ID (string), composition(part)
- Let basicPart, *bP*, have composition.size() = 1
- Let goalPart, *gP*, have composition.size() > 1
- Each part in the composition of *gP* is a *bP*
- Given *gP*, an intermediatePart, *iP*, is any sequential subset of parts in the composition of *gP*
- Given *gP*, there exists an optimal assemblyGraph *aG*, such that *gP* is assembled from *bP*s to construct *gP* (and *iP*s)
- Given *aG*, all *iP*s within *aG* are assemblyIntermediates, *aI*
- aI are parts, *iP*s are not parts
- Given *gP* and *aG*, *aI* which must be in *aG* are requiredIntermediates, *rqI*

- Given *gP* and *aG*, *aI* which must not be in *aG* are fobiddenIntermediates, *fbI*
- Given *gP* and *aG*, *aI* which are biased to be in *aG* are recommendedIntermediates, *rcI*

5.7.3 Pseudocode

The pseudocode is broken into four pieces: Multi-Goal-Part Algorithm, Multi--Goal-Part Algorithm Helper Methods, Overhang Assignment Algorithm, and Overhang Assignment Algorithm Helper Methods. This pseudocode provides an abstracted description of the Raven source-code and covers the most critical Raven algorithms and subroutines. Source code for the algorithms are also available online (https://github.com/CIDARLAB/raven-public). None of the Raven UI code is detailed and oligonucleotide design pseudocode is omitted, as it completes template designs for each assembly method and optimizes according to nearest-neighbor melting temperature methods and desired homology length. More sophisticated and optimized primer designs are detailed in other work[52].

5.7.4 Multi Goal Part Algorithm

This section covers the primary methods used to determine optimized solutions for cloning steps for any number of goal parts under consideration. The multi-goal-part algorithm considers a set of target constructs and initially determines the maximum number of stages with which all constructs can be assembled. This number restricts the number of stages all constructs will be assembled in and allows smaller constructs to be assembled in more stages if they can share intermediate steps and thus reduce the total cloning step count. The algorithm then determines the single-goal part cost for each individual construct and saves the solution for the construct which scores best according to the multi-goal part algorithm heuristics. This part is removed from the goal-part set and this process is repeated until all goal parts have optimized solutions.

The single-goal-part algorithm finds an optimized solution for assembling a construct based upon our heuristic scores. This algorithm recursively explores the assembly space for the construct in question by determining the cost of the intermediate constructs used to assemble final constructs.

5.7.5 Multi Goal Part Algorithm Helper Methods

These helper methods describe the order of the scoring heuristics for the single-goal-part algorithm and the way in which the space is explored. For each single goal part under consideration, at least one solution for each allowed number of parts-per-reaction is explored up to the specified upper limit, unless no such paths are possible. If specific intermediates are forbidden, alternative paths are explored. Additional space is explored for parts in a library which may be re-used that would not be considered on the default path.

5.7.6 Overhang Assignment Algorithm

Once the multi-goal-part algorithm has run and provided an optimized solution, this graph is used as input for the overhang assignment algorithms. These algorithms vary across assembly methods, but can be broken into three steps. In the first step, the rules of each assembly method are applied. In most cases, this asserts that there can be no redundancy of overhangs within a single cloning reaction - this would result in cloning reactions with more than one selectable product. Once these restrictions are applied, all opportunities to share overhangs in for all target constructs are optimized. For scarless assembly methods, all overhangs are assumed to be unique unless a part is adjacent to the same two neighboring parts in another instance and all parts are in the same orientation. For assembly methods where all overhang regions are assumed to be the same, this step is also omitted. For assembly methods where this can be leveraged, sharing is optimized based upon part type and orientation. In the final step, a library of parts is considered and a partial Cartesian product is performed in an attempt to re-use as many parts from the parts library as possible without violating steps one and two.

5.7.7 Overhang Assignment Algorithm Helper Methods

In these methods, special considerations are taken to maximize the sharing of overhangs based on part type and orientation. Specifically, overhang sharing is maximized to include sharing opportunities for part types that appear both on the forward and reverse strand.

5.7.8 Modular Overhang Site Selection

Overhang sites used by Raven are selected through a combination of experimental results and a number of computational heuristics. We begin by generating a list of all 256 possible 4bp overhang sequences. From the full list of possible sequences, 12 sequences were experimentally validated, and thus ranked as the top 12 sequences in the list of overhang. The remaining sequences were then iteratively selected by using a scoring scheme that incorporates a number of heuristics as well as traditional alignment methods. We generate a score matrix for all pairwise alignments between the possible sequences. When selecting the next overhang sequence, we sum the scores of a potential overhang sequence aligned to all the sequences already selected. Next we modulate the score of each potential sequence by using heuristics to check for palindromes, monomeric runs, and GC content. The sequence with the lowest score is selected. When a sequence is selected, its reverse complement is also considered selected. In (**Table 2**), ‘‘*’’ indicates the reverse complement of a sequence. Once this list is computed, it is stored as a static resource, which does not need to be recomputed whenever Raven's algorithms are used.

5.8 *In Silico* Random Sampling Experimental Description

Construct Source	#	Best Un-Optimized Solutions		
		RFC 10	RFC 94	Gibson
Bonnet *et al.*[26]	6	5 | 42 | 21	3 | 37 | 58	2 | 7 | 34
Bonnet *et al.*[27]	13	5 | 73 | 23	4 | 98 | 127	2 | 24 | 55
Friedland *et al.*[44]	5	7 | 79 | 27	4 | 90 | 125	3 | 25 | 50
Lou *et al.*[64]	191	7 | 440 | 90	5 | 751 | 420	3 | 447 | 449
Moon *et al.*[69]	15	7 | 97 | 37	4 | 98 | 140	3 | 28 | 84
Suiti *et al.*[91]	23	6 | 111 | 19	4 | 187 | 255	3 | 59 | 97
Tabor *et al.*[96]	6	6 | 37 | 12	4 | 59 | 98	2 | 16 | 26
Tamsir *et al.*[97]	14	5 | 54 | 15	3 | 68 | 93	2 | 21 | 53
ALL	273	7 | 930 | 204	5 | 1497 | 907	4 | 674 | 844

Table 3: The single best un-optimized assembly scores for assembling constructs from the literature discovered in the *in silico* experiments. Literature datasets are described by number of constructs considered in each set. All solutions are reported in a 'Cloning Stages | Cloning steps | PCR steps' format.

To assess the quality of Raven's solutions, we randomly sampled the solution space for both our part junction assignment algorithm and our hierarchical assembly step algorithm, conducting 1000 trials for each data point shown in **Figure 11**. To ensure that our results were not biased by designs of a particular composition or size, we used Eugene to generate a total of 5000 designs. These 5000 designs were based upon five well studied designs published in the literature: the toggle switch[47], repressilator[40], invertase NOR gate[91], transcriptional NOR gate[27], and DIC counter[44]. Given constraints specified by Eugene rules, Eugene performs a Cartesian product for all the possibilities for each part position in each construct design. The Eugene rules used are given in the Eugene Rules section of the supplement and the 5000 constructs generated (1000 for each of the five construct designs) are available separately as CSV files in Raven format. For more details about Raven format, please visit ravencad.org.

Sampling the hierarchical algorithm effectively reduces to randomly sampling the space of partitioning each of the goal parts into cloning steps. For each experiment, we randomly chose a subset of the 1000 constructs for each design (the number selected is shown on the x-axis). For each of the randomly selected sets of goal parts, we partitioned it randomly, pinning solutions so that part reuse can occur. Given that for a part of size n that can be broken into *m* pieces, there would be $\sum_{k=0}^{m}\binom{n}{k}$ possible ways to partition the construct. And so generating all possible partitions and then sampling from the set of all possible partitions is computationally intractable. Instead, we used a geometric distribution to select the value. The geometric distribution biases the selected number of partitions to favor a higher number; this sampling behavior is appropriate given that one-pot reactions are frequently used to assemble many parts at once. Once m is selected in constant time, the partition positions can be generated in O(*mn*) time, which is trivial since *m* and *n* are usually fairly small.

The experiments for sampling the part junction assignment solution space followed a procedure similar to the experiments for the hierarchical assembly step algorithm. For each trial, the script again randomly select a subset of the 1000 constructs for each design. The script constrained the number of steps and stages to be the same between each trial; the hierarchical assembly step algorithm was used to determine the number of steps and stages. After the number of steps and stages are computed, the part junction space was sampled. To sample the part junction solution space, the first step of our overhang assignment algorithm was reused, assigning abstract part junctions such that the assignment produced by random sampling will be correct. Given *n* parts of roughly size *m*, there will be at most *n(m+1)* different overhangs. For each and every partition junction, we enumerate from the first possible assignment value until a valid assignment is reached, which we believe produces the most intuitive naive assignment.

5.9 PCR Verification and Cloning Efficiency

Raven's algorithms assume that all PCR steps will be successfully executed and in cases where a basic part is cloned into a vector, it will also be achieved without need for assembly plan redesign. We verify that Raven-generated oligos produce successful PCRs for MoClo assembly (**Figure 17**) and report the observed efficiency for all attempted cloning reactions. Here, efficiency is calculated as the percentage of white colonies divided by the total number of colonies on a plate after transformation of a MoClo cloning reaction.

Single-part cloning steps yielded the highest average cloning efficiency (72%), with nearly 6-fold greater efficiency than six-part reactions (**Figure 18**). Overall, the cloning efficiency had an approximately negative linear relationship with the number of parts cloned in a single reaction. Using the observed average cloning efficiencies and extrapolating efficiency from a linear regression of this trend for 3- and 4-part reactions as parameters for the assembly plan, Raven calculates a plan with 3 additional steps, and 3 additional PCR steps at an increased average efficiency of 14.4% (**Figure 24**) per cloning step, using the observed efficiency values. This is an example of a case where observed efficiency information can be interactively used within Raven to produce plans with higher average efficiency at the expense of additional steps.

5.10 Raven Assembly Plans

5.10.1 Initial Plan

Raven assembly graphs are shown in a hierarchical format, with library parts at the top and target parts at the bottom. The assembly graphs are organized by stages and automatically generated by graphviz, using auto-generated SBOL-compliant glyphs from

Pigeon (pigeoncad.org)[23]. The initial assembly plan (**Figure 19**) is generated assuming equal efficiency for all reactions and a set of existing cloning vectors and template DNA. Some existing vectors are used in the assembly plan.

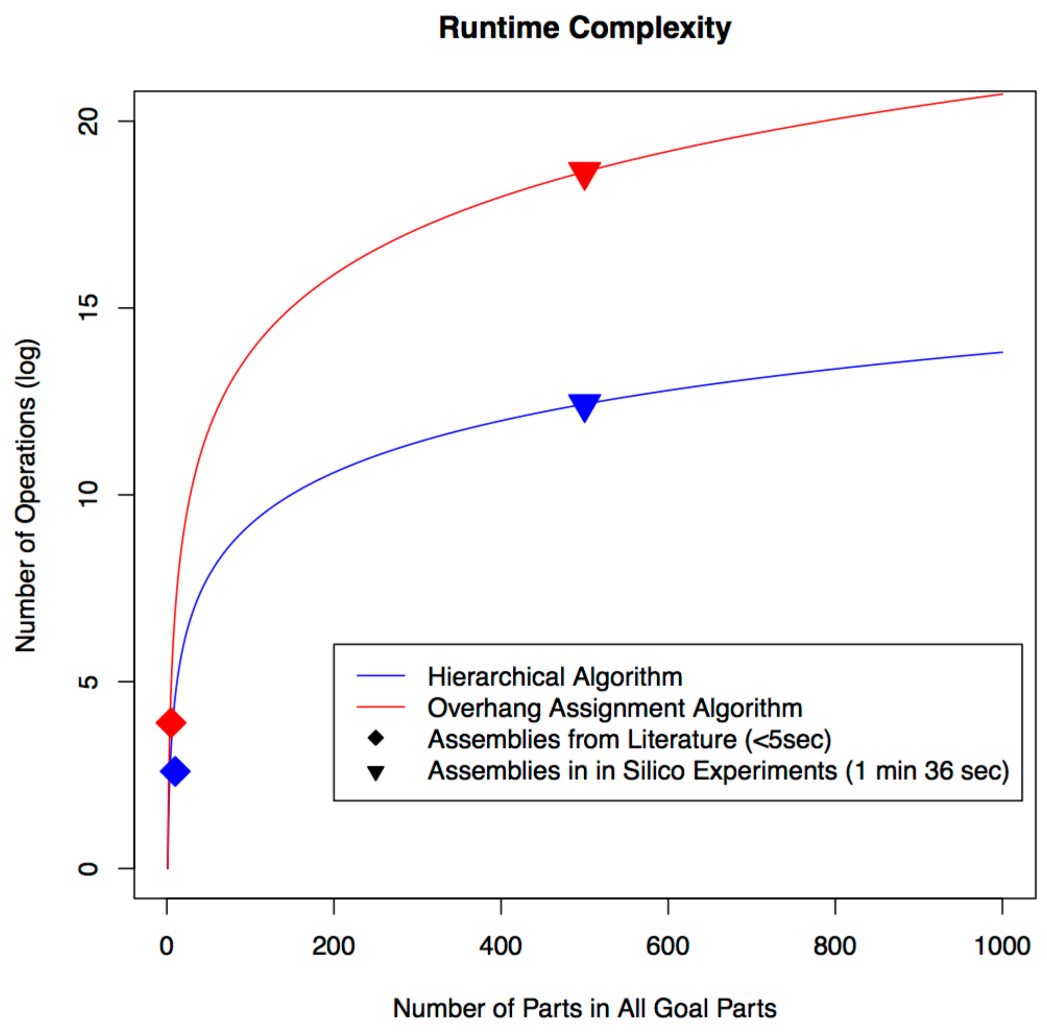


Figure 14: Runtime complexity of Raven algorithms. Raven's two main algorithmic components are bound by n, the number of goal parts and m, the average number of basic parts per goal part, which we show on the horizontal axis as the ‘‘Number of Parts in All Goal Parts", nm. The vertical axis gives the approximate number of calculations according to conventional complexity analysis. Points are shown to give the approximate number of operations required to calculate assembly plans for all of the constructs from the publications shown in Table 1 and the number of operations required to calculate 500 constructs of a design shown in Figure 2A. Separate curves are shown for the hierarchical algorithm and the overhang assignment algorithm. Constructs from each publication and constructs from each design in Figure 2A were calculated individually and then averaged over all publications and designs respectively, giving the time shown in the legend. Note that the times shown are approximations of realtime performance as times are machine and condition dependent (Intel Xeon 2 x 6 core cpu, 24 GB RAM machine used to compute times).

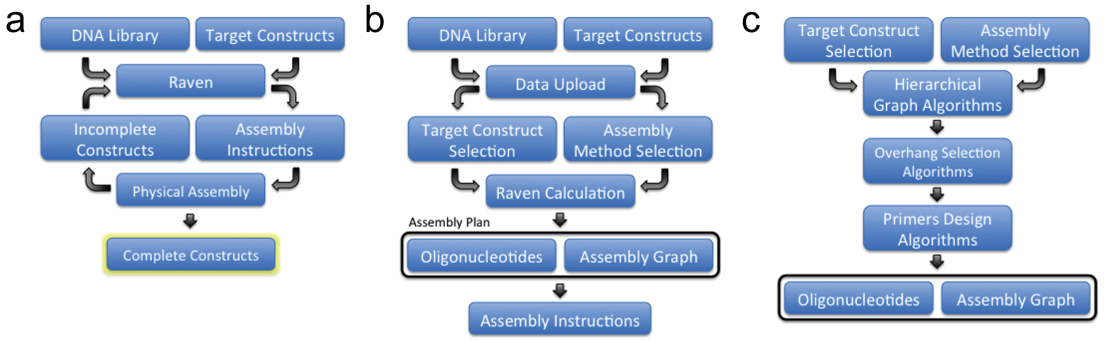


Figure 15: Raven flow charts (a) A user inputs their DNA library and set of target construct into Raven, which generates assembly instructions. These assembly instructions are then implemented and some constructs will be completed (yellow box) according to plan and some will not. These incomplete constructs are input back into Raven for assembly redesign. (b) In the Raven UI, a user inputs a DNA library and set of target construct and selects an assembly method with which to construct a subset of the target constructs. Raven calculates an optimized assembly graph and oligonucleotide designs necessary to execute the plan (black box) and generates human- or computer-readable instruction files. (c) Raven optimizations are calculated in three major sequential pieces: Hierarchical step optimization, overhang selection optimization and finally primer designs for PCR steps. The sum of all these optimizations is an assembly graph and set of oligonucleotides (black box).

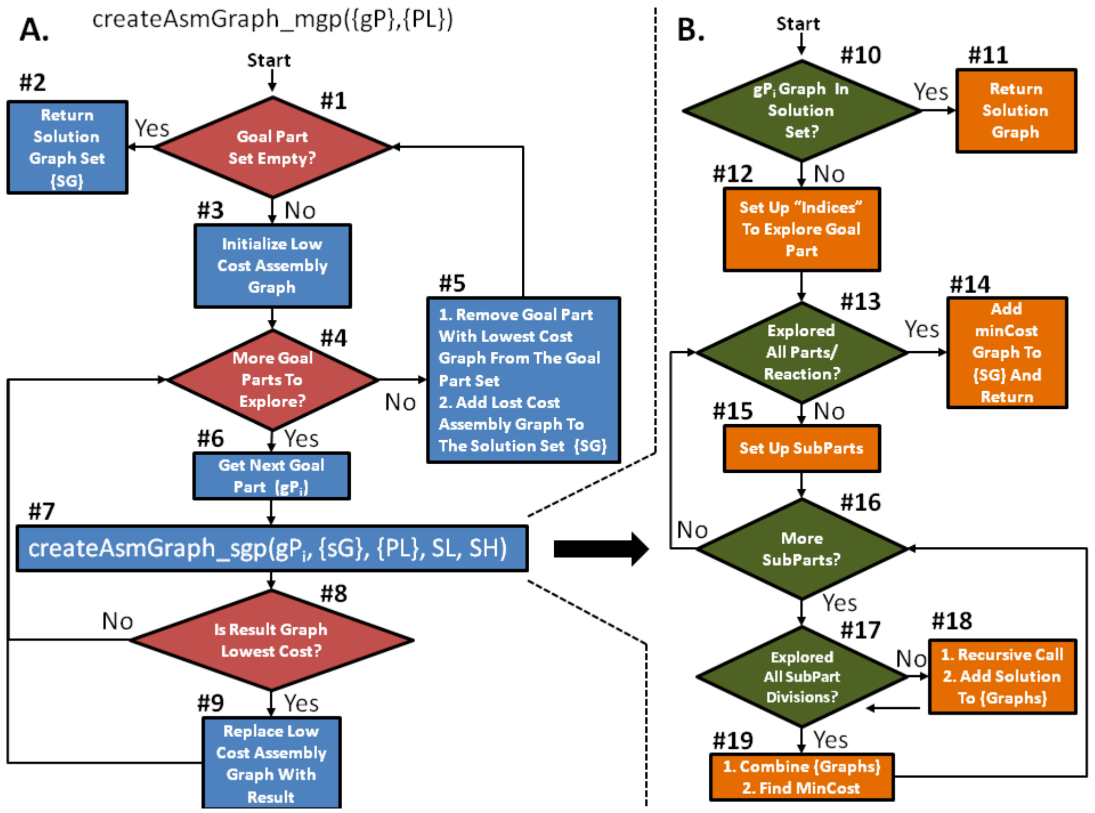

A.
createAsmGraph_mgp({gP},{PL})
Start
#1
Goal Part Set Empty?
Yes
#2
Return Solution Graph Set {SG}
No
#3
Initialize Low Cost Assembly Graph
#4
More Goal Parts To Explore?
No
#5
1. Remove Goal Part With Lowest Cost Graph From The Goal Part Set
2. Add Lost Cost Assembly Graph To The Solution Set {SG}
Yes
#6
Get Next Goal Part (gP_i)
#7
createAsmGraph_sgp(gP_i, {sG}, {PL}, SL, SH)
#8
Is Result Graph Lowest Cost?
No
Yes
#9
Replace Low Cost Assembly Graph With Result
B.
Start
#10
gP_i Graph In Solution Set?
Yes
#11
Return Solution Graph
No
#12
Set Up "Indices" To Explore Goal Part
#13
Explored All Parts/ Reaction?
Yes
#14
Add minCost Graph To {SG} And Return
No
#15
Set Up SubParts
#16
More SubParts?
No
Yes
#17
Explored All SubPart Divisions?
No
#18
1. Recursive Call
2. Add Solution To {Graphs}
Yes
#19
1. Combine {Graphs}
2. Find MinCost

Figure 16: The assembly algorithm is illustrated here as a set of two flowcharts. (A) presents the initial function call to ‘‘createAsmGraph_mgp" (create assembly graph multiple-goal-parts). Here a set of goal parts to be assembled is presented {gP} along with a parts library {PL}. Once that set is empty #1 (all goal parts have been removed as their solutions are found), the function returns (#2). Otherwise a low cost baseline is established (#3) and each goal part is explored (#4, #6,#7). If a solution for a particular goal part results in a lower cost than the baseline, it becomes the baseline (#8 and #9). Eventually all goal parts have been explored. The goal part with the lowest cost graph is removed from the goal part set and its graph is added to the solutions (#5). The process repeats again with the goal part solutions accumulating allowing their intermediate assemblies to be available for subsequent solutions. The right hand side figure (B) illustrates the call to ‘‘createAsmGraph_sgp" (create assembly graph single-goal-part). If the goal part already has a solution the function returns (#10, #11). Otherwise legal indices to partition the single part into subparts are set up in #12. #13 and #14 demonstrate that once all parts per reaction (how many legal subparts) are explored, the latest created graph is returned. #15, #16, and #17 illustrate that subpart divisions are explored ultimately resulting in a recursive call to this same function with each of the legal divisions of this part (#18). #19 illustrates the subpart solutions must be combined and that result compared with the lowest discovered graph so far for this goal part. We refer the reader to both the pseudocode and the open source code for more details.

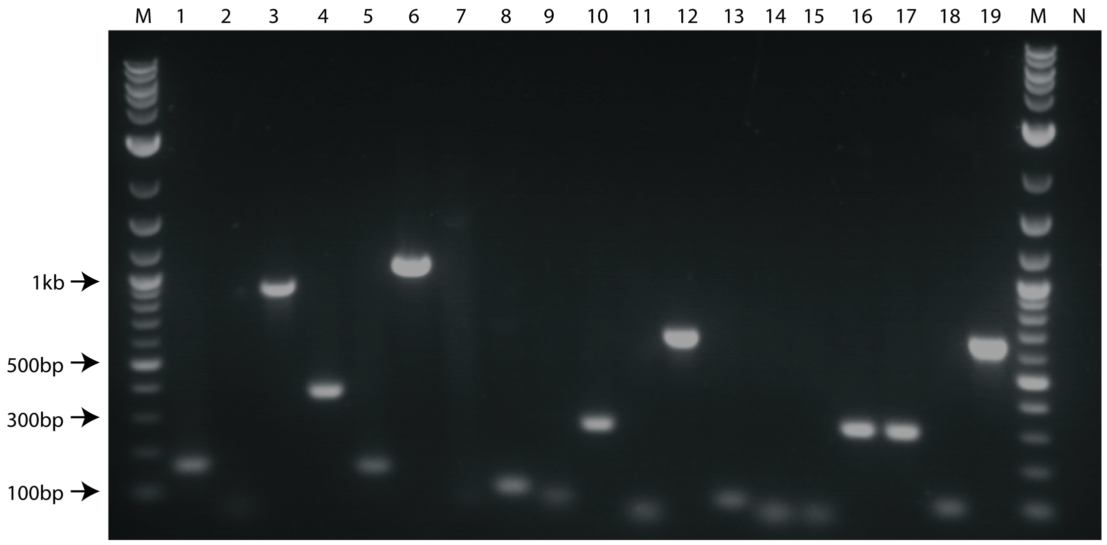


Figure 17: PCR products for basic parts. Expected sizes (Left to Right) IR1_IR2_Term2|5|3* 178bp, loxP|1|3 62bp, flpe|3|m1 941bp, flpe|m1|5 392bp, IR1_IR2_Term2|0|3 178bp, cre|6|0 1093bp, pTet|7|0 112bp, T1|4|8 133bp, pA1LacO|1|2* 105bp, pBAD|7|0 314bp, FRT|0|2 62bp, gfp|6|m2 652bp, gfp|m2|0 93bp, loxP|0|1 62bp, FRT|3*|0 62bp, pBAD|2|1 314bp, pBAD|1|2* 314bp, gfp|m2|4 93bp, gfp|2|m2 652bp

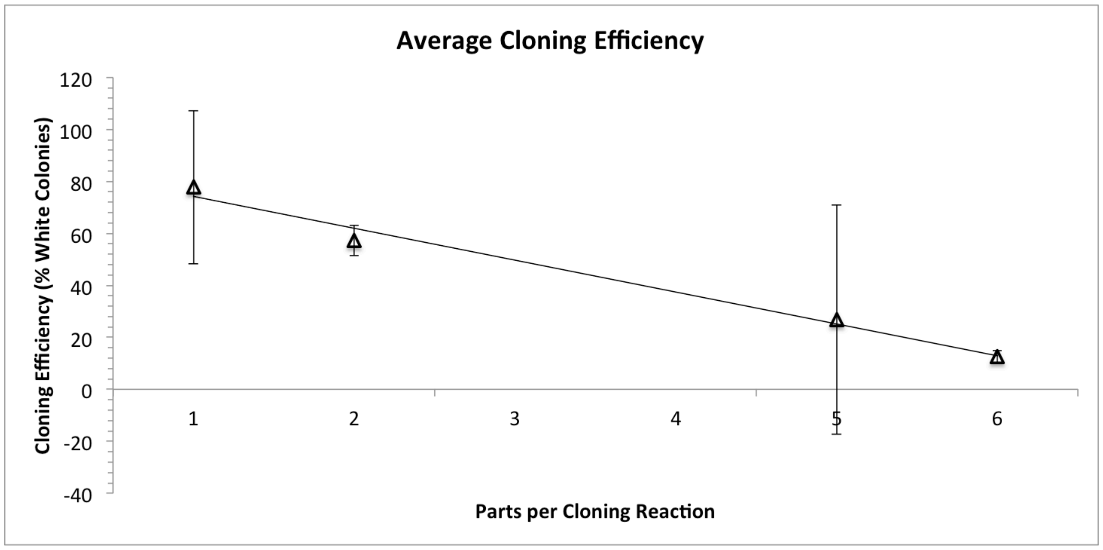


Figure 18: Average cloning efficiency as a function of parts per cloning reaction for MoClo (BBF RFC 94). Measured average values marked with a triangle and error bars represent one standard deviation.

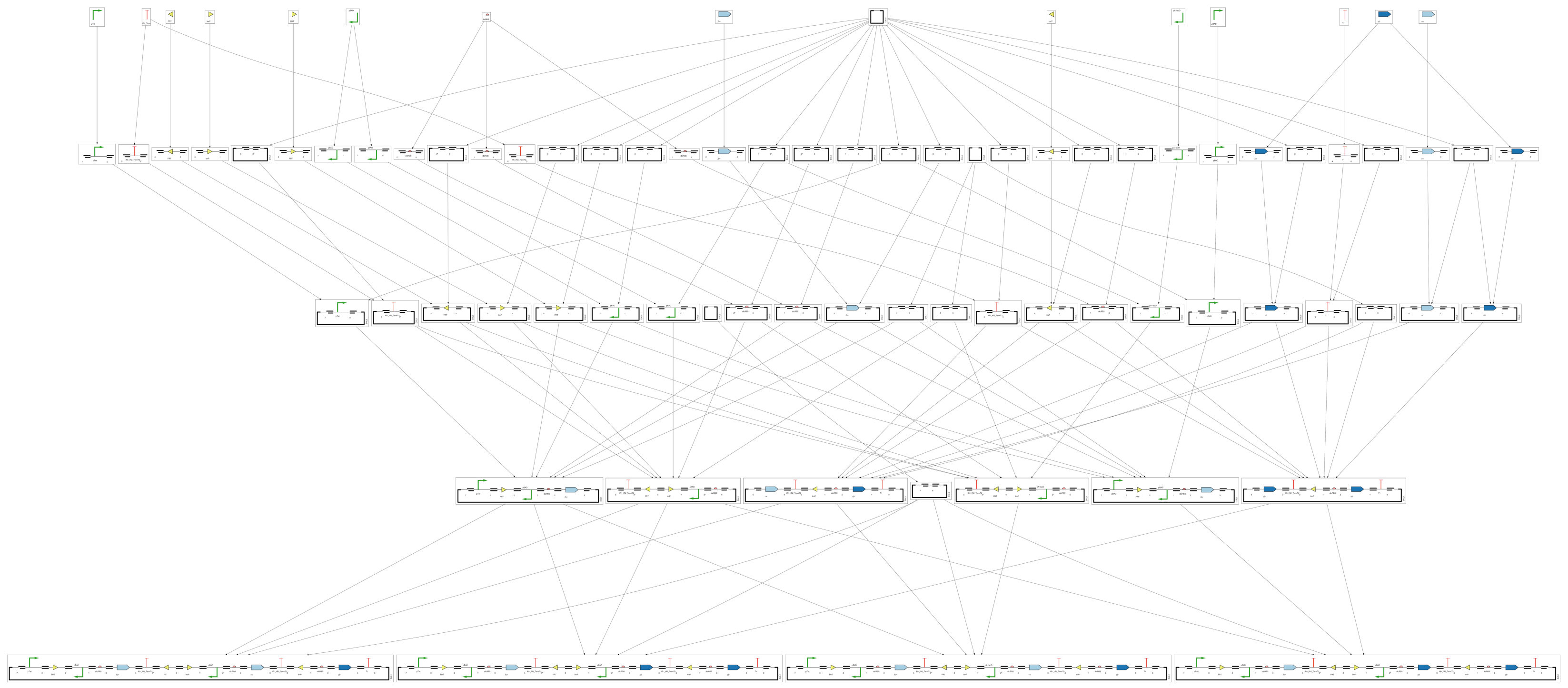


Figure 19: Preliminary assembly plan for genetic counter constructs.

5.10.2 Redesigned Plans

Given the failed intermediates [pBAD|+, FRT|+, pBAD|-, rbs|+, flpe|+] and [pTet|+, FRT|+, pBAD|-, rbs|+, flpe|+] in **Figure 19** and the success of all other intermediate steps, Raven can be leveraged to generate a new plan to complete the assembly of the target constructs (**Figure 24**). In the calculation of this plan, all successful steps are added to the library that can be used for assembly and the intermediates that failed are forbidden from appearing in the redesigned solution. This revised solution required only 2 stages, 7 cloning steps and 2 PCRs.

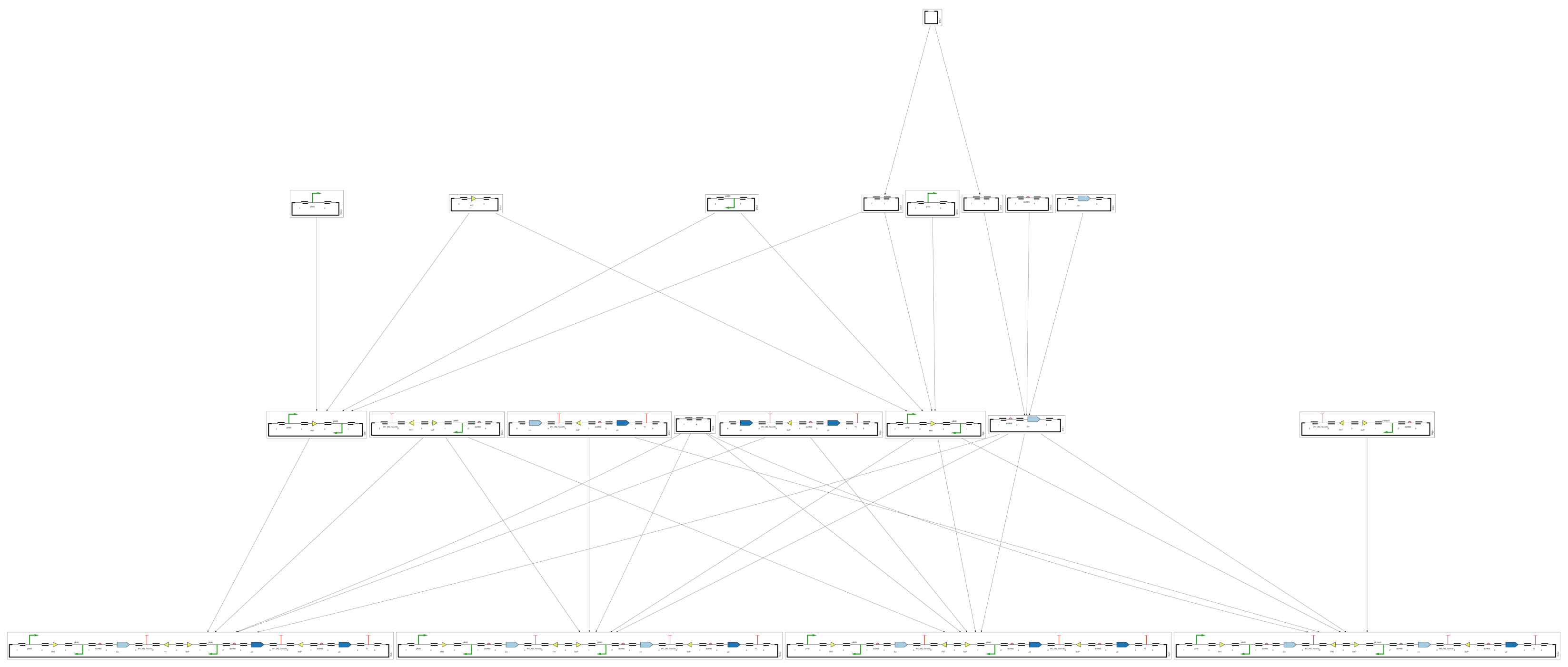


Figure 20: First redesigned assembly plan for genetic counter constructs.

The first plan redesign was also unsuccessful. Intermediate [pBAD|+, FRT|+, pBAD|-] was not successful, but the other two intermediates were constructed successfully. Although this intermediate did not express any coding sequences, it was hypothesized that this plasmid creates too much transcriptional activity in a high copy plasmid.

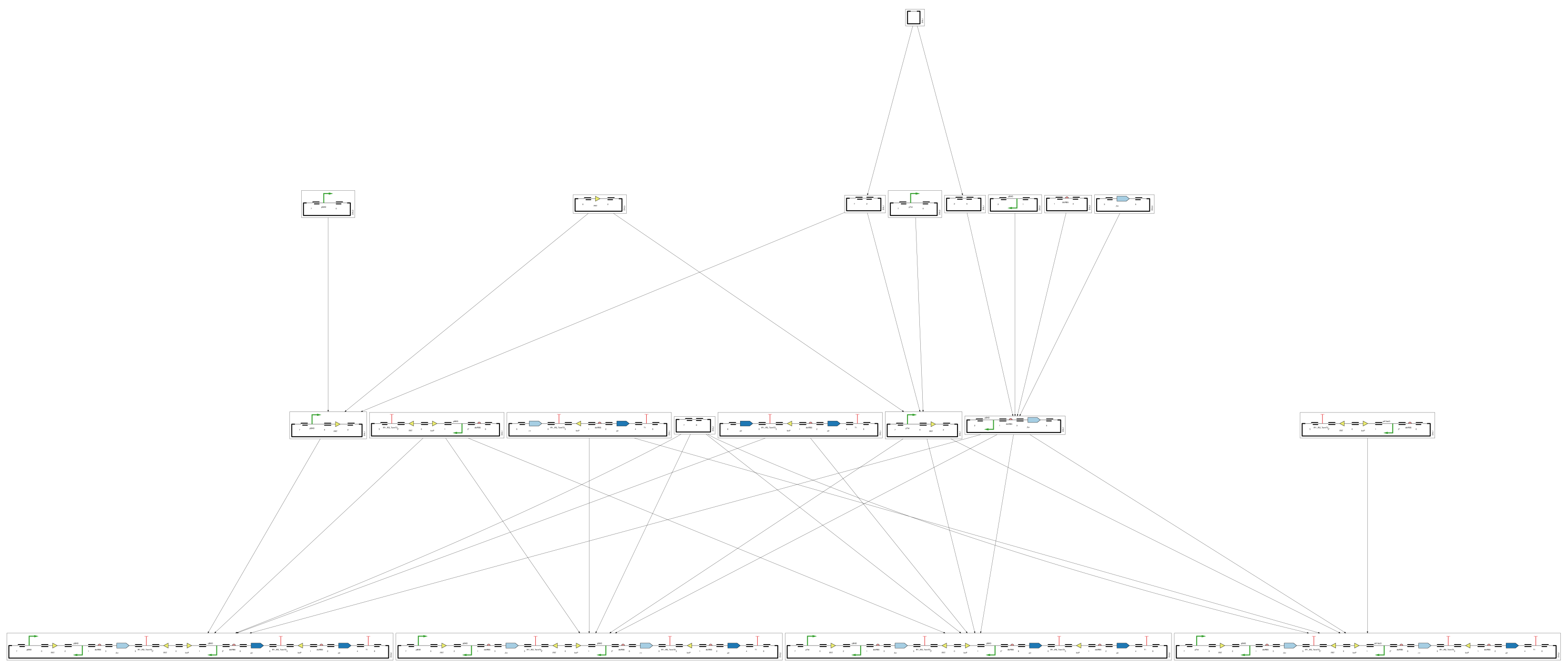


Figure 21: Second redesigned assembly plan for genetic counter constructs.

This second redesign was capable of producing all desired intermediates, but was not successful in assembling the final constructs. We hypothesized that this was due to high expression of recombinases in a high copy plasmid, similar to the problems experienced in cloning the intermediates. To circumvent this problem, all intermediates and a bacterial aritificial chromosome (pBAC) backbone were digested, gel-extracted, and ligated independently. These ligations, however did not produce any successful clones.

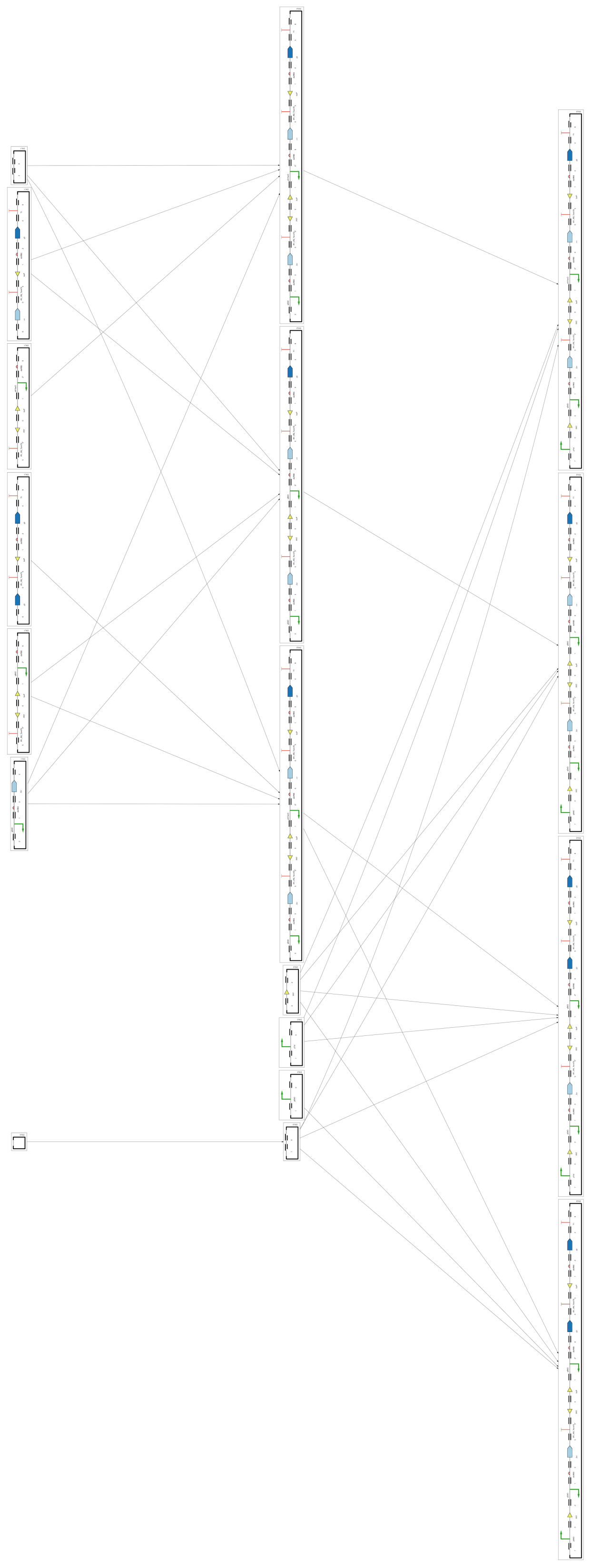


Figure 22: Final redesigned assembly plan for genetic counter constructs.

The final redesign forced the creation of intermediates 12, 13, and 14, that accounted for the entirety of the final constructs' compositions, except for the leading promoter that drives each construct's function. Requiring an extra stage to build these constructs allowed for a change in antibiotic resistance, so that there was no need for independent digestion, gel extraction and ligation steps and also decreased the number of parts to be cloned in into the pBAC backbone in the final cloning steps.

5.10.3 Biased Plans Outside Core Heuristics

If certain intermediate constructs are known to cause cloning difficulties outside of the core Raven heuristics, these intermediates can be biased against using the Raven 'discouraged' markings. Intermediates marked as discouraged will be scored worse than otherwise equivalent solutions with no discouraged intermediates. Alternatively, if a user is aware of certain types of intermediates that are desirable outside of the Raven core heuristics, specific intermediates may be 'recommended' in the Raven UI. Solutions with more recommended intermediates will be scored higher than otherwise equivalent solutions.

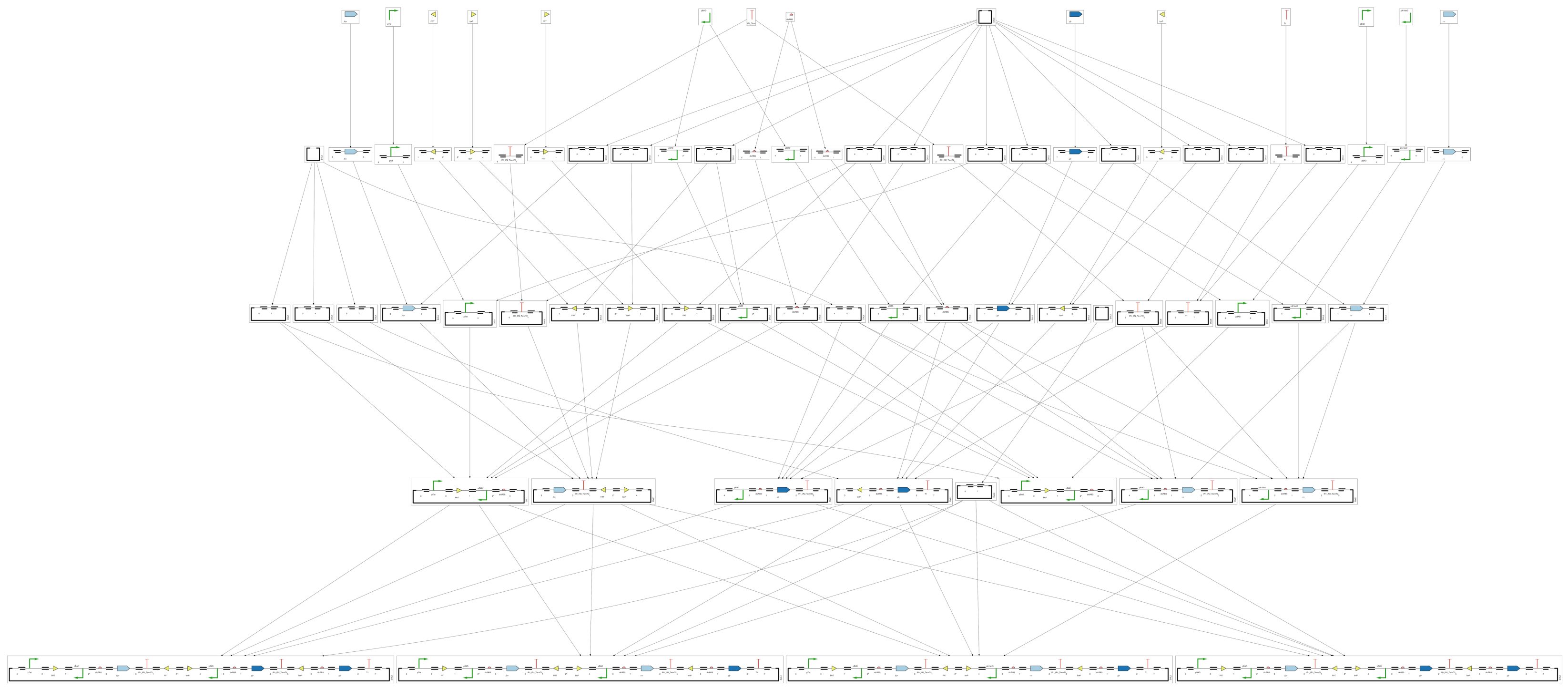


Figure 23: MoClo assembly plan for genetic counter constructs discouraging all possible intermediates that could constitutively express a recombinase.

In the example of the genetic counter constructs, constitutive expression of recombinases can cause significant difficulty in each cloning step. To avoid an assembly plan that contains intermediates that might express recombinases, all such intermediates could be discouraged. Specifically, the following intermediates can be explicitly discouraged in the Raven UI to produce a MoClo solution for these constructs and these discouraged intermediates (**Figure 23**):

-> pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+, gfp|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+, gfp|+, IR1_IR2_TermT2|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+, gfp|+, IR1_IR2_TermT2|+, loxP|-]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+, gfp|+, IR1_IR2_TermT2|+, loxP|-, dicRBS|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+, gfp|+, IR1_IR2_TermT2|+, loxP|-, dicRBS|+, gfp|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+, cre|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+, cre|+, IR1_IR2_TermT2|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+, cre|+, IR1_IR2_TermT2|+, loxP|-]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+, cre|+, IR1_IR2_TermT2|+, loxP|-, dicRBS|+]

-> [pBAD|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pBAD|-, dicRBS|+, cre|+, IR1_IR2_TermT2|+, loxP|-, dicRBS|+, gfp|+]

-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+]

-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+]

-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-]

-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+]

-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pA1lacO|-
]

-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pA1lacO|-
, dicRBS|+]

-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pA1lacO|-
, dicRBS|+, cre|+]

-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pA1lacO|-
, dicRBS|+, cre|+, IR1_IR2_TermT2|+]

-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pA1lacO|-
, dicRBS|+, cre|+, IR1_IR2_TermT2|+, loxP|-]

-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pA1lacO|-

, dicRBS|+, cre|+, IR1_IR2_TermT2|+, loxP|-, dicRBS|+]
-> [pTet|+, FRT|+, pBAD|-, dicRBS|+, flpe|+, IR1_IR2_TermT2|+, FRT|-, loxP|+, pA1lacO|-
, dicRBS|+, cre|+, IR1_IR2_TermT2|+, loxP|-, dicRBS|+, gfp|+]

This plan takes 3 stages, 28 cloning steps and 33 PCR steps. This plan is a 1-cloning step and 1-PCR step decrease from the initial plan and includes no intermediates capable of expressing recombinases. This serves as an example where using recommended or discouraged intermediates results in an approximately equivalent-cost solution that does not include problematic intermedaites.

5.10.4 Efficiency-Optimized Plan

Using the measured average efficiency values from the implemented plans as a function of parts-per-reaction, an alternative assembly plan can be considered that has additional steps compared to the initial plan (**Figure 19**), but at higher average efficiency (i.e. more steps with higher expected efficiency).

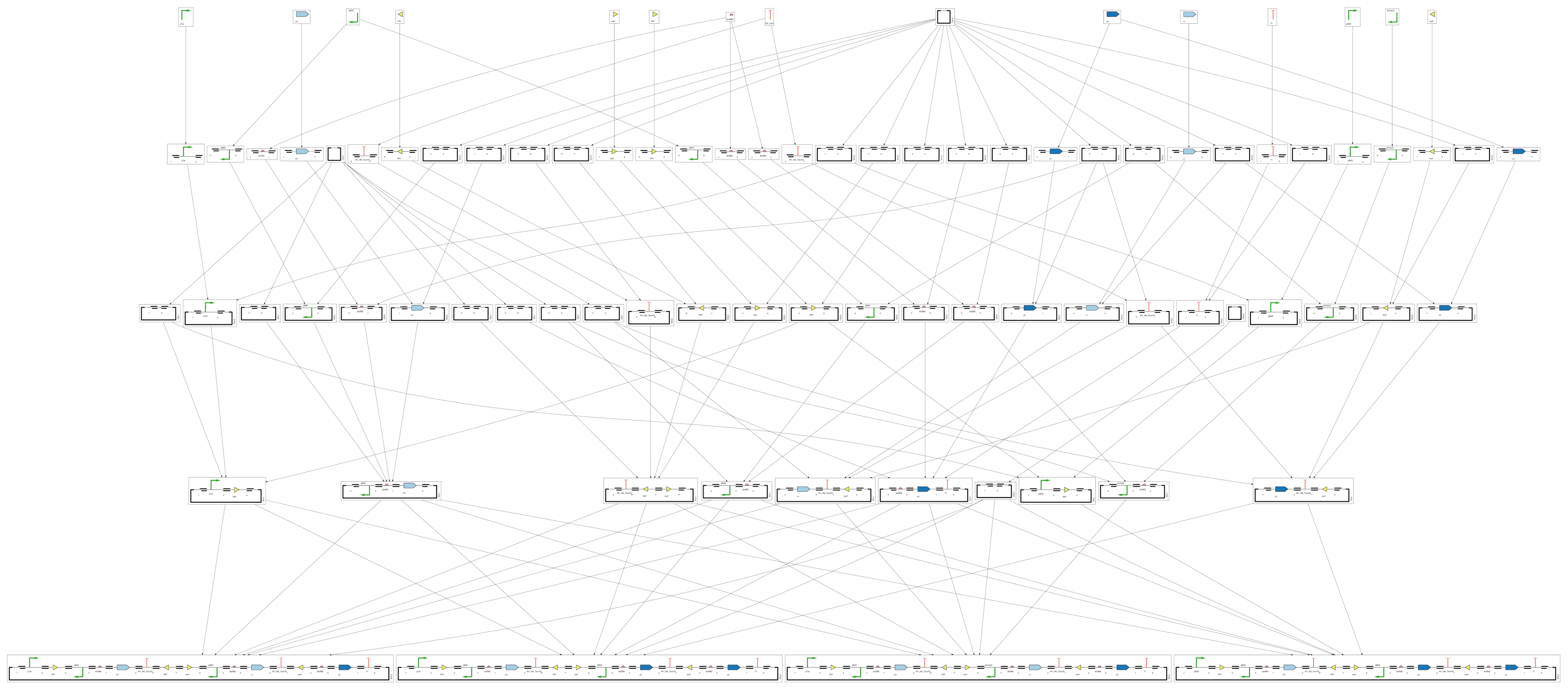


Figure 24: Efficiency-Optimized MoClo assembly plan for the four genetic counter constructs based on measured efficiency from prior implemented plans.

5.10.5 Repressilator Constructs

The repressilator MoClo assembly plan (**Figure 25**) is generated assuming equal efficiency for all reactions not exceeding 4 parts per reaction and a set of existing cloning vectors and DNA parts. No new vectors or Level 0 parts are created in this assembly plan, so there are 0 PCR steps required. This plan leverages the CIDAR MoClo Library, that already contains each destination vector and Level 0 part required to create the repressilators.

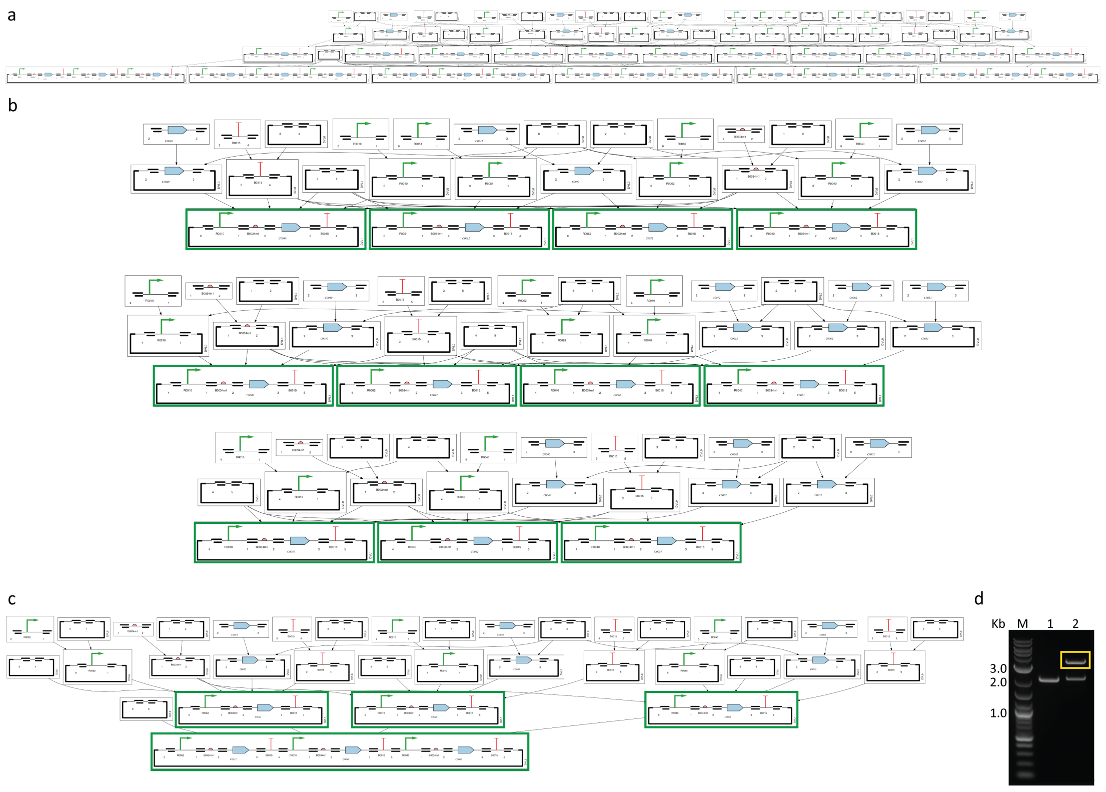


Figure 25: MoClo assembly plan for Repressilator Constructs. (a) Assembly plan for six repressilators with shared parts. (b) The intermediate constructs required to build the six repressilators shown in (a). The green boxes indicate a successful assembly. (c) Assembly plan for one repressilator using intermediates shown in (b) with the green boxes indicating the assembly was a success. (d) An agarose gel (1% TAE) showing a restriction map for the repressilator shown in (c). Plasmid DNA (1000 ng) was digested with SpeI enzyme (NEB). Lane M shows the molecular marker (2-log ladder from NEB), lane 1 shows an empty Level 2 vector (2204bps), and lane 2 shows the insert containing the repressilator (3304 bps; yellow box) cut out from its Level 2 vector backbone (2204 bps).

Raven detected two previously undetected BbsI recognition sites in the *lacI* gene (sequence cloned from BBa_C0012), so these reactions (which contain a BbsI site) yielded lower efficiencies than expected when generating the final repressilators, which all contained *lacI*. Despite this problem, two of the final six repressilators were assembled correctly. While the *lacI* gene contains two BbsI sites, the 4 bps overhangs it produces do not match the overhangs used in that cloning step so the final construct can still be generated, albeit at a lower efficiency than normally observed for 4-part reactions (**Figure 18**).

5.11 Raven Human-Readable Assembly Instructions

Human-readable assembly instructions are generated automatically by Raven in an output text-file format. The assembly files are organized by construct. For each construct, the full set of assembly instructions is organized by stages. Assembly instructions tell the user which overhangs each part is supposed to get and the direction of the parts within each construct, intermediate and basic part. At the end of the file is all oligos that must be ordered to construct all constructs in this selected assembly. These oligonucleotides are not ordered by part or construct.

5.11.1 Initial Instructions for Counter Constructs

The following assembly file was used to implement the initial assembly plan for the genetic counter constructs. The graphical plan associated with this plan is also shown (**Figure 19**).

```
*********************************************

Assembly Instructions for target part: Counter4
```

```
********************************************

-> Assemble Counter4|7|8|[+, +, -, +, +, +, -, +, -,
    +, +, +, -, +, +, +] by performing a MoClo
   cloning reaction with: intermediate2|7|5|[+, +, -, +,
   +], intermediate1|5|6|[+, -, +, -, +], intermediate3
   |6|8|[+, +, -, +, +, +], DVL2|7|8

-> Assemble intermediate5|7|5|[+, +, -, +, +] by
   performing a MoClo cloning reaction with: pTet|7|0|[+],
   FRT|0|2|[+], pBAD|2|1|[-], dicRBS|1|3|[+], flpe|3|5|[+],
   DVL1|7|5

-> Assemble intermediate1|5|6|[+, -, +, -, +] by
   performing a MoClo cloning reaction with: IR1_IR2_TermT2
   |5|3*|[+], FRT|3*|0|[-], loxP|0|1|[+], pBAD|1|2*|[-],
   dicRBS|2*|6|[+], DVL1|5|6

-> Assemble intermediate3|6|8|[+, +, -, +, +, +] by
   performing a MoClo cloning reaction with: cre|6|0|[+],
   IR1_IR2_TermT2|0|3|[+], loxP|3|1|[-], dicRBS|1|2|[+], gfp
   |2|4|[+], T1|4|8|[+], DVL1|6|8
```

```
-> Assemble pBAD|7|0|[+] by performing a MoClo cloning
   reaction with: pBAD|7|0|[+], DVL0|7|0

-> Assemble FRT|0|2|[+] by performing a MoClo cloning
   reaction with: FRT|0|2|[+], DVL0|0|2

-> Assemble pBAD|2|1|[-] by performing a MoClo cloning
   reaction with: pBAD|2|1|[-], DVL0|2|1

-> Assemble dicRBS|1|3|[+] by performing a MoClo cloning
   reaction with: dicRBS|1|3|[+], DVL0|1|3

-> Assemble flpe|3|5|[+] by performing a MoClo cloning
   reaction with: flpe|3|5|[+], DVL0|3|5

-> Assemble IR1_IR2_TermT2|5|3*|[+] by performing a MoClo
   cloning reaction with: IR1_IR2_TermT2|5|3*|[+], DVL0|5|3*

-> Assemble FRT|3*|0|[-] by performing a MoClo cloning
   reaction with: FRT|3*|0|[-], DVL0|3*|0

-> Assemble loxP|0|1|[+] by performing a MoClo cloning
   reaction with: loxP|0|1|[+], DVL0|0|1
```

```
-> Assemble pBAD|1|2*|[-] by performing a MoClo cloning
   reaction with: pBAD|1|2*|[-], DVL0|1|2*

-> Assemble dicRBS|2*|6|[+] by performing a MoClo cloning
   reaction with: dicRBS|2*|6|[+], DVL0|2*|6

-> Assemble cre|6|0|[+] by performing a MoClo cloning
   reaction with: cre|6|0|[+], DVL0|6|0

-> Assemble IR1_IR2_TermT2|0|3|[+] by performing a MoClo
   cloning reaction with: IR1_IR2_TermT2|0|3|[+], DVL0|0|3

-> Assemble loxP|3|1|[-] by performing a MoClo cloning
   reaction with: loxP|3|1|[-], DVL0|3|1

-> Assemble dicRBS|1|2|[+] by performing a MoClo cloning
   reaction with: dicRBS|1|2|[+], DVL0|1|2

-> Assemble gfp|2|4|[+] by performing a MoClo cloning
   reaction with: gfp|2|4|[+], DVL0|2|4

-> Assemble T1|4|8|[+] by performing a MoClo cloning
   reaction with: T1|4|8|[+], DVL0|4|8
```

```
PCR loxP with oligos: oligo1 and oligo2 to get part: loxP
    |3|1|[-]

PCR IR1_IR2_TermT2 with oligos: oligo3 and oligo4 to get
    part: IR1_IR2_TermT2|0|3|[+]

PCR loxP with oligos: oligo5 and oligo6 to get part: loxP
    |0|1|[+]

PCR FRT with oligos: oligo7 and oligo8 to get part: FRT
    |0|2|[+]

PCR cre with oligos: oligo9 and oligo10 to get part: cre
    |6|0|[+]

PCR IR1_IR2_TermT2 with oligos: oligo11 and oligo12 to get
    part: IR1_IR2_TermT2|5|3*|[+]

Anneal oligos: oligo13 and oligo14 to get part: dicRBS
    |2*|6|[+]

PCR T1 with oligos: oligo15 and oligo16 to get part: T1
    |4|8|[+]
```

```
PCR gfp with oligos: oligo17 and oligo18 to get part: gfp
    |2|4|[+]

PCR pBAD with oligos: oligo19 and oligo20 to get part: pBAD
    |2|1|[-]

PCR pBAD with oligos: oligo21 and oligo22 to get part: pBAD
    |1|2*|[-]

PCR flpe with oligos: oligo23 and oligo24 to get part: flpe
    |3|5|[+]

PCR FRT with oligos: oligo25 and oligo26 to get part: FRT
    |3*|0|[-]

PCR pBAD with oligos: oligo27 and oligo28 to get part: pBAD
    |7|0|[+]

Anneal oligos: oligo29 and oligo30 to get part: dicRBS
    |1|2|[+]

Anneal oligos: oligo31 and oligo32 to get part: dicRBS
    |1|3|[+]
```

```
PCR lacZ with oligos: oligo33 and oligo34 to get vector:
    DVL1|7|5

PCR lacZ with oligos: oligo35 and oligo36 to get vector:
    DVL0|3|1

PCR lacZ with oligos: oligo37 and oligo38 to get vector:
    DVL0|0|2

PCR lacZ with oligos: oligo39 and oligo40 to get vector:
    DVL0|3*|0

PCR lacZ with oligos: oligo41 and oligo42 to get vector:
    DVL0|6|0

PCR lacZ with oligos: oligo43 and oligo44 to get vector:
    DVL0|4|8

PCR lacZ with oligos: oligo45 and oligo46 to get vector:
    DVL0|2*|6

PCR lacZ with oligos: oligo47 and oligo48 to get vector:
    DVL0|7|0
```

```
PCR lacZ with oligos: oligo49 and oligo50 to get vector:
   DVL1|6|8

PCR lacZ with oligos: oligo51 and oligo52 to get vector:
   DVL0|2|4

PCR lacZ with oligos: oligo53 and oligo54 to get vector:
   DVL0|1|3

PCR lacZ with oligos: oligo55 and oligo56 to get vector:
   DVL0|1|2*

PCR lacZ with oligos: oligo57 and oligo58 to get vector:
   DVL0|0|3

PCR lacZ with oligos: oligo59 and oligo60 to get vector:
   DVL0|5|3*

PCR lacZ with oligos: oligo61 and oligo62 to get vector:
   DVL2|7|8

********************************************

Assembly Instructions for target part: Counter1
```

```
*********************************************

-> Assemble Counter1|7|8|[+, +, -, +, +, +, -, +, -,
   +, +, +, -, +, +, +] by performing a MoClo
  cloning reaction with: intermediate2|7|5|[+, +, -, +,
   +], intermediate1|5|6|[+, -, +, -, +], intermediate4
  |6|8|[+, +, -, +, +, +], DVL2|7|8

-> Assemble intermediate2|7|5|[+, +, -, +, +] by
  performing a MoClo cloning reaction with: pBAD|7|0|[+],
  FRT|0|2|[+], pBAD|2|1|[-], dicRBS|1|3|[+], flpe|3|5|[+],
  DVL1|7|5

-> Assemble intermediate1|5|6|[+, -, +, -, +] by
  performing a MoClo cloning reaction with: IR1_IR2_TermT2
  |5|3*|[+], FRT|3*|0|[-], loxP|0|1|[+], pBAD|1|2*|[-],
  dicRBS|2*|6|[+], DVL1|5|6

-> Assemble intermediate4|6|8|[+, +, -, +, +, +] by
  performing a MoClo cloning reaction with: gfp|6|0|[+],
  IR1_IR2_TermT2|0|3|[+], loxP|3|1|[-], dicRBS|1|2|[+], gfp
  |2|4|[+], T1|4|8|[+], DVL1|6|8

-> Assemble pBAD|7|0|[+] by performing a MoClo cloning
  reaction with: pBAD|7|0|[+], DVL0|7|0
```

```
-> Assemble FRT|0|2|[+] by performing a MoClo cloning
   reaction with: FRT|0|2|[+], DVL0|0|2

-> Assemble pBAD|2|1|[-] by performing a MoClo cloning
   reaction with: pBAD|2|1|[-], DVL0|2|1

-> Assemble dicRBS|1|3|[+] by performing a MoClo cloning
   reaction with: dicRBS|1|3|[+], DVL0|1|3

-> Assemble flpe|3|5|[+] by performing a MoClo cloning
   reaction with: flpe|3|5|[+], DVL0|3|5

-> Assemble IR1_IR2_TermT2|5|3*|[+] by performing a MoClo
   cloning reaction with: IR1_IR2_TermT2|5|3*|[+], DVL0|5|3*

-> Assemble FRT|3*|0|[-] by performing a MoClo cloning
   reaction with: FRT|3*|0|[-], DVL0|3*|0

-> Assemble loxP|0|1|[+] by performing a MoClo cloning
   reaction with: loxP|0|1|[+], DVL0|0|1

-> Assemble pBAD|1|2*|[-] by performing a MoClo cloning
   reaction with: pBAD|1|2*|[-], DVL0|1|2*
```

```
-> Assemble dicRBS|2*|6|[+] by performing a MoClo cloning
   reaction with: dicRBS|2*|6|[+], DVL0|2*|6

-> Assemble gfp|6|0|[+] by performing a MoClo cloning
   reaction with: gfp|6|0|[+], DVL0|6|0

-> Assemble IR1_IR2_TermT2|0|3|[+] by performing a MoClo
   cloning reaction with: IR1_IR2_TermT2|0|3|[+], DVL0|0|3

-> Assemble loxP|3|1|[-] by performing a MoClo cloning
   reaction with: loxP|3|1|[-], DVL0|3|1

-> Assemble dicRBS|1|2|[+] by performing a MoClo cloning
   reaction with: dicRBS|1|2|[+], DVL0|1|2

-> Assemble gfp|2|4|[+] by performing a MoClo cloning
   reaction with: gfp|2|4|[+], DVL0|2|4

-> Assemble T1|4|8|[+] by performing a MoClo cloning
   reaction with: T1|4|8|[+], DVL0|4|8

PCR loxP with oligos: oligo1 and oligo2 to get part: loxP
   |3|1|[-]
```

```
PCR IR1_IR2_TermT2 with oligos: oligo3 and oligo4 to get
    part: IR1_IR2_TermT2|0|3|[+]

PCR gfp with oligos: oligo63 and oligo64 to get part: gfp
    |6|0|[+]

PCR loxP with oligos: oligo5 and oligo6 to get part: loxP
    |0|1|[+]

PCR FRT with oligos: oligo7 and oligo8 to get part: FRT
    |0|2|[+]

PCR IR1_IR2_TermT2 with oligos: oligo11 and oligo12 to get
    part: IR1_IR2_TermT2|5|3*|[+]

Anneal oligos: oligo13 and oligo14 to get part: dicRBS
    |2*|6|[+]

PCR T1 with oligos: oligo15 and oligo16 to get part: T1
    |4|8|[+]

PCR gfp with oligos: oligo17 and oligo18 to get part: gfp
    |2|4|[+]
```

```
PCR pBAD with oligos: oligo19 and oligo20 to get part: pBAD
    |2|1|[-]

PCR pBAD with oligos: oligo21 and oligo22 to get part: pBAD
    |1|2*|[-]

PCR flpe with oligos: oligo23 and oligo24 to get part: flpe
    |3|5|[+]

PCR FRT with oligos: oligo25 and oligo26 to get part: FRT
    |3*|0|[-]

PCR pBAD with oligos: oligo27 and oligo28 to get part: pBAD
    |7|0|[+]

Anneal oligos: oligo29 and oligo30 to get part: dicRBS
    |1|2|[+]

Anneal oligos: oligo31 and oligo32 to get part: dicRBS
    |1|3|[+]

PCR lacZ with oligos: oligo49 and oligo50 to get vector:
    DVL1|6|8
```

```
PCR lacZ with oligos: oligo41 and oligo42 to get vector:
    DVL0|6|0

PCR lacZ with oligos: oligo33 and oligo34 to get vector:
    DVL1|7|5

PCR lacZ with oligos: oligo35 and oligo36 to get vector:
    DVL0|3|1

PCR lacZ with oligos: oligo37 and oligo38 to get vector:
    DVL0|0|2

PCR lacZ with oligos: oligo39 and oligo40 to get vector:
    DVL0|3*|0

PCR lacZ with oligos: oligo61 and oligo62 to get vector:
    DVL2|7|8

PCR lacZ with oligos: oligo43 and oligo44 to get vector:
    DVL0|4|8

PCR lacZ with oligos: oligo45 and oligo46 to get vector:
    DVL0|2*|6
```

```
PCR lacZ with oligos: oligo47 and oligo48 to get vector:
    DVL0|7|0

PCR lacZ with oligos: oligo51 and oligo52 to get vector:
    DVL0|2|4

PCR lacZ with oligos: oligo53 and oligo54 to get vector:
    DVL0|1|3

PCR lacZ with oligos: oligo55 and oligo56 to get vector:
    DVL0|1|2*

PCR lacZ with oligos: oligo57 and oligo58 to get vector:
    DVL0|0|3

PCR lacZ with oligos: oligo59 and oligo60 to get vector:
    DVL0|5|3*

******************************************

Assembly Instructions for target part: Counter2

******************************************
```

```
-> Assemble Counter2|7|8|[+, +, -, +, +, +, -, +, -,
    +, +, +, -, +, +, +] by performing a MoClo
   cloning reaction with: intermediate5|7|5|[+, +, -, +,
    +], intermediate1|5|6|[+, -, +, -, +], intermediate4
   |6|8|[+, +, -, +, +, +], DVL2|7|8

-> Assemble intermediate5|7|5|[+, +, -, +, +] by
   performing a MoClo cloning reaction with: pTet|7|0|[+],
   FRT|0|2|[+], pBAD|2|1|[-], dicRBS|1|3|[+], flpe|3|5|[+],
   DVL1|7|5

-> Assemble intermediate1|5|6|[+, -, +, -, +] by
   performing a MoClo cloning reaction with: IR1_IR2_TermT2
   |5|3*|[+], FRT|3*|0|[-], loxP|0|1|[+], pBAD|1|2*|[-],
   dicRBS|2*|6|[+], DVL1|5|6

-> Assemble intermediate4|6|8|[+, +, -, +, +, +] by
   performing a MoClo cloning reaction with: gfp|6|0|[+],
   IR1_IR2_TermT2|0|3|[+], loxP|3|1|[-], dicRBS|1|2|[+], gfp
   |2|4|[+], T1|4|8|[+], DVL1|6|8

-> Assemble pBAD|7|0|[+] by performing a MoClo cloning
   reaction with: pBAD|7|0|[+], DVL0|7|0
```

```
-> Assemble FRT|0|2|[+] by performing a MoClo cloning
   reaction with: FRT|0|2|[+], DVL0|0|2

-> Assemble pBAD|2|1|[-] by performing a MoClo cloning
   reaction with: pBAD|2|1|[-], DVL0|2|1

-> Assemble dicRBS|1|3|[+] by performing a MoClo cloning
   reaction with: dicRBS|1|3|[+], DVL0|1|3

-> Assemble flpe|3|5|[+] by performing a MoClo cloning
   reaction with: flpe|3|5|[+], DVL0|3|5

-> Assemble IR1_IR2_TermT2|5|3*|[+] by performing a MoClo
   cloning reaction with: IR1_IR2_TermT2|5|3*|[+], DVL0|5|3*

-> Assemble FRT|3*|0|[-] by performing a MoClo cloning
   reaction with: FRT|3*|0|[-], DVL0|3*|0

-> Assemble loxP|0|1|[+] by performing a MoClo cloning
   reaction with: loxP|0|1|[+], DVL0|0|1

-> Assemble pBAD|1|2*|[-] by performing a MoClo cloning
   reaction with: pBAD|1|2*|[-], DVL0|1|2*
```

```
-> Assemble dicRBS|2*|6|[+] by performing a MoClo cloning
   reaction with: dicRBS|2*|6|[+], DVL0|2*|6

-> Assemble gfp|6|0|[+] by performing a MoClo cloning
   reaction with: gfp|6|0|[+], DVL0|6|0

-> Assemble IR1_IR2_TermT2|0|3|[+] by performing a MoClo
   cloning reaction with: IR1_IR2_TermT2|0|3|[+], DVL0|0|3

-> Assemble loxP|3|1|[-] by performing a MoClo cloning
   reaction with: loxP|3|1|[-], DVL0|3|1

-> Assemble dicRBS|1|2|[+] by performing a MoClo cloning
   reaction with: dicRBS|1|2|[+], DVL0|1|2

-> Assemble gfp|2|4|[+] by performing a MoClo cloning
   reaction with: gfp|2|4|[+], DVL0|2|4

-> Assemble T1|4|8|[+] by performing a MoClo cloning
   reaction with: T1|4|8|[+], DVL0|4|8

PCR loxP with oligos: oligo1 and oligo2 to get part: loxP
   |3|1|[-]
```

```
PCR IR1_IR2_TermT2 with oligos: oligo3 and oligo4 to get
    part: IR1_IR2_TermT2|0|3|[+]

PCR gfp with oligos: oligo63 and oligo64 to get part: gfp
    |6|0|[+]

PCR loxP with oligos: oligo5 and oligo6 to get part: loxP
    |0|1|[+]

PCR FRT with oligos: oligo7 and oligo8 to get part: FRT
    |0|2|[+]

PCR IR1_IR2_TermT2 with oligos: oligo11 and oligo12 to get
    part: IR1_IR2_TermT2|5|3*|[+]

Anneal oligos: oligo13 and oligo14 to get part: dicRBS
    |2*|6|[+]

PCR T1 with oligos: oligo15 and oligo16 to get part: T1
    |4|8|[+]

PCR gfp with oligos: oligo17 and oligo18 to get part: gfp
    |2|4|[+]
```

```
PCR pBAD with oligos: oligo19 and oligo20 to get part: pBAD
    |2|1|[-]

PCR pBAD with oligos: oligo21 and oligo22 to get part: pBAD
    |1|2*|[-]

PCR flpe with oligos: oligo23 and oligo24 to get part: flpe
    |3|5|[+]

PCR FRT with oligos: oligo25 and oligo26 to get part: FRT
    |3*|0|[-]

PCR pBAD with oligos: oligo27 and oligo28 to get part: pBAD
    |7|0|[+]

Anneal oligos: oligo29 and oligo30 to get part: dicRBS
    |1|2|[+]

Anneal oligos: oligo31 and oligo32 to get part: dicRBS
    |1|3|[+]

PCR lacZ with oligos: oligo49 and oligo50 to get vector:
    DVL1|6|8
```

```
PCR lacZ with oligos: oligo41 and oligo42 to get vector:
    DVL0|6|0

PCR lacZ with oligos: oligo33 and oligo34 to get vector:
    DVL1|7|5

PCR lacZ with oligos: oligo35 and oligo36 to get vector:
    DVL0|3|1

PCR lacZ with oligos: oligo37 and oligo38 to get vector:
    DVL0|0|2

PCR lacZ with oligos: oligo39 and oligo40 to get vector:
    DVL0|3*|0

PCR lacZ with oligos: oligo61 and oligo62 to get vector:
    DVL2|7|8

PCR lacZ with oligos: oligo43 and oligo44 to get vector:
    DVL0|4|8

PCR lacZ with oligos: oligo45 and oligo46 to get vector:
    DVL0|2*|6
```

```
PCR lacZ with oligos: oligo47 and oligo48 to get vector:
   DVL0|7|0

PCR lacZ with oligos: oligo51 and oligo52 to get vector:
   DVL0|2|4

PCR lacZ with oligos: oligo53 and oligo54 to get vector:
   DVL0|1|3

PCR lacZ with oligos: oligo55 and oligo56 to get vector:
   DVL0|1|2*

PCR lacZ with oligos: oligo57 and oligo58 to get vector:
   DVL0|0|3

PCR lacZ with oligos: oligo59 and oligo60 to get vector:
   DVL0|5|3*

********************************************

Assembly Instructions for target part: Counter6

********************************************
```

```
-> Assemble Counter6|7|8|[+, +, -, +, +, +, -, +, -,
    +, +, +, -, +, +, +] by performing a MoClo
   cloning reaction with: intermediate5|7|5|[+, +, -, +,
    +], intermediate6|5|6|[+, -, +, -, +], intermediate3
   |6|8|[+, +, -, +, +, +], DVL2|7|8

-> Assemble intermediate5|7|5|[+, +, -, +, +] by
   performing a MoClo cloning reaction with: pTet|7|0|[+],
   FRT|0|2|[+], pBAD|2|1|[-], dicRBS|1|3|[+], flpe|3|5|[+],
   DVL1|7|5

-> Assemble intermediate6|5|6|[+, -, +, -, +] by
   performing a MoClo cloning reaction with: IR1_IR2_TermT2
   |5|3*|[+], FRT|3*|0|[-], loxP|0|1|[+], pA1lacO|1|2*|[-],
   dicRBS|2*|6|[+], DVL1|5|6

-> Assemble intermediate3|6|8|[+, +, -, +, +, +] by
   performing a MoClo cloning reaction with: cre|6|0|[+],
   IR1_IR2_TermT2|0|3|[+], loxP|3|1|[-], dicRBS|1|2|[+], gfp
   |2|4|[+], T1|4|8|[+], DVL1|6|8

-> Assemble pTet|7|0|[+] by performing a MoClo cloning
   reaction with: pTet|7|0|[+], DVL0|7|0
```

```
-> Assemble FRT|0|2|[+] by performing a MoClo cloning
   reaction with: FRT|0|2|[+], DVL0|0|2

-> Assemble pBAD|2|1|[-] by performing a MoClo cloning
   reaction with: pBAD|2|1|[-], DVL0|2|1

-> Assemble dicRBS|1|3|[+] by performing a MoClo cloning
   reaction with: dicRBS|1|3|[+], DVL0|1|3

-> Assemble flpe|3|5|[+] by performing a MoClo cloning
   reaction with: flpe|3|5|[+], DVL0|3|5

-> Assemble IR1_IR2_TermT2|5|3*|[+] by performing a MoClo
   cloning reaction with: IR1_IR2_TermT2|5|3*|[+], DVL0|5|3*

-> Assemble FRT|3*|0|[-] by performing a MoClo cloning
   reaction with: FRT|3*|0|[-], DVL0|3*|0

-> Assemble loxP|0|1|[+] by performing a MoClo cloning
   reaction with: loxP|0|1|[+], DVL0|0|1

-> Assemble pA1lacO|1|2*|[-] by performing a MoClo cloning
   reaction with: pA1lacO|1|2*|[-], DVL0|1|2*
```

```
-> Assemble dicRBS|2*|6|[+] by performing a MoClo cloning
   reaction with: dicRBS|2*|6|[+], DVL0|2*|6

-> Assemble cre|6|0|[+] by performing a MoClo cloning
   reaction with: cre|6|0|[+], DVL0|6|0

-> Assemble IR1_IR2_TermT2|0|3|[+] by performing a MoClo
   cloning reaction with: IR1_IR2_TermT2|0|3|[+], DVL0|0|3

-> Assemble loxP|3|1|[-] by performing a MoClo cloning
   reaction with: loxP|3|1|[-], DVL0|3|1

-> Assemble dicRBS|1|2|[+] by performing a MoClo cloning
   reaction with: dicRBS|1|2|[+], DVL0|1|2

-> Assemble gfp|2|4|[+] by performing a MoClo cloning
   reaction with: gfp|2|4|[+], DVL0|2|4

-> Assemble T1|4|8|[+] by performing a MoClo cloning
   reaction with: T1|4|8|[+], DVL0|4|8

PCR loxP with oligos: oligo1 and oligo2 to get part: loxP
   |3|1|[-]
```

```
PCR IR1_IR2_TermT2 with oligos: oligo3 and oligo4 to get
    part: IR1_IR2_TermT2|0|3|[+]

PCR loxP with oligos: oligo5 and oligo6 to get part: loxP
    |0|1|[+]

PCR FRT with oligos: oligo7 and oligo8 to get part: FRT
    |0|2|[+]

PCR cre with oligos: oligo9 and oligo10 to get part: cre
    |6|0|[+]

PCR IR1_IR2_TermT2 with oligos: oligo11 and oligo12 to get
    part: IR1_IR2_TermT2|5|3*|[+]

Anneal oligos: oligo13 and oligo14 to get part: dicRBS
    |2*|6|[+]

PCR T1 with oligos: oligo15 and oligo16 to get part: T1
    |4|8|[+]

PCR gfp with oligos: oligo17 and oligo18 to get part: gfp
    |2|4|[+]
```

```
PCR pBAD with oligos: oligo19 and oligo20 to get part: pBAD
    |2|1|[-]

PCR flpe with oligos: oligo23 and oligo24 to get part: flpe
    |3|5|[+]

PCR FRT with oligos: oligo25 and oligo26 to get part: FRT
    |3*|0|[-]

Anneal oligos: oligo29 and oligo30 to get part: dicRBS
    |1|2|[+]

PCR pTet with oligos: oligo65 and oligo66 to get part: pTet
    |7|0|[+]

PCR pA1lacO with oligos: oligo67 and oligo68 to get part:
    pA1lacO|1|2*|[-]

Anneal oligos: oligo31 and oligo32 to get part: dicRBS
    |1|3|[+]

PCR lacZ with oligos: oligo61 and oligo62 to get vector:
    DVL2|7|8
```

PCR lacZ with oligos: oligo35 and oligo36 to get vector: DVL0|3|1

PCR lacZ with oligos: oligo37 and oligo38 to get vector: DVL0|0|2

PCR lacZ with oligos: oligo39 and oligo40 to get vector: DVL0|3∗|0

PCR lacZ with oligos: oligo33 and oligo34 to get vector: DVL1|7|5

PCR lacZ with oligos: oligo41 and oligo42 to get vector: DVL0|6|0

PCR lacZ with oligos: oligo43 and oligo44 to get vector: DVL0|4|8

PCR lacZ with oligos: oligo45 and oligo46 to get vector: DVL0|2∗|6

PCR lacZ with oligos: oligo49 and oligo50 to get vector: DVL1|6|8

```
PCR lacZ with oligos: oligo51 and oligo52 to get vector:
    DVL0|2|4

PCR lacZ with oligos: oligo53 and oligo54 to get vector:
    DVL0|1|3

PCR lacZ with oligos: oligo57 and oligo58 to get vector:
    DVL0|0|3

PCR lacZ with oligos: oligo55 and oligo56 to get vector:
    DVL0|1|2*

PCR lacZ with oligos: oligo47 and oligo48 to get vector:
    DVL0|7|0

PCR lacZ with oligos: oligo59 and oligo60 to get vector:
    DVL0|5|3*

*******************************************
```

5.12 Interactive Assembly Summary

Because it is not feasible for a human to design hundreds or thousands of assembly plans manually and even more difficult to produce efficient and low-cost solutions for such sets, a computational tool to automatically determine these solutions is needed. And because assembly planning instructions are necessary for liquid-handling robots and microfluidics to perform high throughput cloning and other automation techniques, the absence of an automated method to inform a robot which steps to take to assemble genetic constructs would severely limit the automation power of a larger tool pipeline.

Raven generates experimentally valid assembly plans, and, although it cannot guarantee success of any one plan or complete target sequence, it can generate new plans on the basis of some specific step failures and efficiency data. While these algorithms have the ability to incorporate feedback of reaction failures and successes to produce better solutions, they do not provide any methodology for predicting the success or failure of specific assembly steps or the construct's function. It is important to note that some standardized cloning protocols cannot be rigidly implemented to clone all constructs owing to inherent complexity of function of the constructs under consideration: some cloning challenges still must be solved by amending standard protocols and thus fall outside the purview of a protocol-agnostic assembly plan.

Finally, formal assembly files can be used to capture assembly information from previously attempted assemblies. The documentation of cloning reaction success and failure and of the path to successful assembly can be accumulated and allow easier reproduction of published work. This is particularly important because this information is often poorly documented, which hinders the ability to build on previous work. Formally documented assembly planning provides a better avenue for tracking this information, and previously attempted assemblies could be studied to develop new heuristics and bring further insight to popular molecular cloning methods.

Raven currently supports only six highly used, well-defined cloning methods, but additional systematic biases and constraints outside the tool's core heuristics can be applied to Raven's solutions by specifying forced, forbidden, recommended and discouraged intermediates and specific cloning vectors. Moreover, the principles of this approach could be expanded and further generalized to nearly to any cloning method, provided common sub-problem scoring required by dynamic programming. The generality of the algorithmic solutions and the breadth of the permitted inputs allow assembly solutions to be adapted to potentially any DNA assembly method because Raven broadly suggests how to reuse DNA libraries to build a set of genetic constructs.

6 Automated Data Analysis and Simulation

The second type of instructions produced by *Phoenix* are instructions that define a set of experimental measurements to make upon a set of strains with their corresponding constructs. This file is a simple CSV file with columns for strain, environmental conditions, time and indication of controls to be used in analysis. The final column in this file is for a cytometry file name that was recorded for the strain, environmental condition and time specified for that row.

This ‘key file’ represents all of the data acquired in one set of functional measurement experiments. A *Phoenix* user receives this key file along with the *Raven* assembly instructions to build and test constructs. These files are protocol-agnostic (i.e. they do not include detailed information on reagent quantity to be added to each reaction and growth conditions), but a set of default protocols will be made available on the *Phoenix* website.

6.1 Data Acquisition

With these instructions and default protocols, a user has a complete plan for building and testing all necessary constructs for that phase in the design hierarchy. While these testing instructions are protocol agnostic to allow for some flexibility in laboratory set-up, minor variations in cloning reaction and testing experiment set-up can have consequences in the experimental outcomes.

While *Raven* can guarantee that its assembly plannings are valid in terms of assembly steps and stages and assigned flanking sequences, it cannot guarantee that any particular plan will be successful. Sometimes it may be necessary for a user to change some aspects of the assembly plan to build all target constructs. One of the key assumptions of the functional testing is that strains with identical genetic information should perform the same in each laboratory setting. If this assumption is true, readings acquired on each flow cytometer must still be normalized. To perform this normalization, we require users to run Speherotech RCP-30-5A 8-peak rainbow calibration beads to normalize data acquired on each machine to an absolute unit for all channels - molecules of equivalent fluorescein (MEFL). Finally, *Phoenix* instructs users to grow cells to saturation in luria broth (LB) from a plated colony and then re-innoculated from that culture into minimal media plus glucose to grow to log-phase growth before measurement. These defined growth conditions reduce variation in measurement.

6.2 Data Analysis

After all constructs are built and tested, the user is responsible for returning the key file indicating which strain, environmental condition, and time each raw data file corresponds to. This file and a folder containing the raw data are interpreted by an analysis script written in *R*, using the *Bioconductor* packages to process the cytometry files. This script first parses the key file to identify unique rows and replicate data rows to determine all unique time points and environmental conditions for each strain. Duplicate rows are considered experimental replicates - three replicates are standard, but there are no limitations on providing more replicates.

Each set of replicates is processed in the following way: First, only channels that are controlled for with a positive control are considered, all other channels except forward and side scatter are discarded. Next, the spectral overlap matrix is calculated using the fluorescent protein positive controls and a negative control. After this step, an elliptical gate is created for forward and side scatter to remove cells that are outliers in size and shape and then all readings with negative values are removed. Next, for the remaining populations, spectral overlap correction is performed with the spectral overlap matrix. After this correction, a clustering algorithm for identifying the largest cluster of readings in each channel is run and the mean values of these clusters for each channel is collected. As a final step, readings from the negative control processed in the same analytics flow are subtracted from all samples to correct for auto-fluorescence.

This processed meta-data is used to create plots in the R console and the data points are exported in a CSV file for input into *Phoenix*. *Phoenix* associates these values to an experiment object associated with each plasmid.

6.3 Simulation for Parameter Estimation and Compositional Designs

This processed meta-data is used to estimate the rate constants needed for each *EXPRESSEE* and *EXPRESSOR*. For *EXPRESSORs*, the steady-state expression measurement is used with the measured degradation rate of its added testing component, *gfp* or *bfp* to fit to a steady-state expression constant.

For *EXPRESSEEs*, experimental data is used to determine two or three rate constants (depending on the regulator). The degradation rate of each *EXPRESSEE* and degradation control construct is measured by adding the transcriptional inhibitor, chloramphenicol to the environment at time t = 0 to inhibit protein expression. With protein expression stopped, a time-series measurement for fluorescence can be used to determine the degradation parameters. To fit the regulation constant, a titration for the inducer controlling the expression of the *EXPRESSEE* is performed and the regulation constant is determined with parameter estimation. To determine the small molecule interaction constant for *EXPRESSEEs* that are sensitive to small molecule inducers, another parameter estimation is run for the fluorescence produced by the regulation control as a function of the small molecule present in the environment.

These paramater estimations are performed with an open-source simulation tool called

COPASI. Once the parameter values for each *EXPRESSEE* and *EXPRESSOR* are estimated, kinetic models for compositional designs can be simulated. These simulations are also performed with *COPASI*[53] and the resulting simulation traces are returned to *Phoenix*, where they are evaluated against functional specifications.

6.4 Structural Failure Mode Grammars

The simulated compositional traces are then model-checked against an STL specification and evaluated for robustness. The most robust simulated compositions are selected and subsequently returned to the user for building and testing. Sometimes, this first round of simulation may align very well with experimental results, but other times it may not. In cases where the simulations and experimental results do not align, it is important to have a mechanisms for probing into why the disagreement occurred and to learn from this information to make better selections from future simulations.

The fourth chapter of this thesis discussed how grammars can be applied to validate a structural specification. It is also possible to use grammars in the context of types of structural motifs that are often the source of particular design failures. In *Phoenix*, me and my colleagues defined 10 types of common failure modes and wrote grammars in ANTLR[77] to determine which failure modes exist in any given structure.

When a design's experimental data does not match its simulation, *Phoenix* identifies which failure modes are present and based upon the specific disagreement in traces, instructs the creation of additional constructs to test which failure mode. The results of this experimental inquiry are stored in *Phoenix* as design rules documenting which particular sequence combinations resulted in failures and avoids making similar design selections for future designs.

7 Data Visualization, Documentation and Storage

After experimental data is processed and used for simulation, it is important to store information in a database and effectively visualize the results to a user. In a synthetic synthetic biology workflow, this includes information related to design, DNA assembly and testing for each plasmid. This task involves determining the correct information to store, determining the best tool for storing this information and then determining the most informative data to display back to a user.

7.1 Data Storage and Management

Since biological data is typically large and multi-dimensional, it is important to determine which type of information is important to store in a database and which type of database is the most appropriate. In synthetic biology, there is no existing consensus on which information is essential and which is optional, although some groups have put significant effort into trying to establish these standards[45]. There are also a number of open-source repositories for synthetic biological part information, but there is little agreement on which data standard to adopt.

For my tools, I opted to use *Clotho* as my data storage tool. The reasons for using this tool as opposed to other options was threefold: 1. *Clotho* has a well thought-out core data model, but is intentionally flexible to amendments to the data model. 2. Most of my other tools, including *Phoenix* are Java based like *Clotho*, so there is a smooth data transfer and 3. Since it was developed in the same research group, I can occasionally contribute as a developer to the functionality of the tool and easily tailor functionality with others on the *Phoenix* development team.

After I decided to use *Clotho* as the database tool, it needed to be determined exactly what type of information to store. In general, I opted to store plasmid sequences, feature annotation information, design information, metadata for assembly, testing, and simulation. This is because only the metadata is used for key decision making and can be automatically regenerated from raw data or other metadata. Similarly, images from generated plots are not saved because they can be reproduced automatically. The next challenge was to determine which metadata and sequence information to return to users of my tools.

7.2 Electronic Datasheet Generator for Data Visualization

In other engineering fields such as electrical engineering, researchers use electronic datasheets to accomplish this task for electronic transistors. In recent years, synthetic biologists have thought about employing similar approaches to documenting and representing information related to their plasmids[41] to help biologists evaluate parts for use in compositional designs. Since there currently exist few tools for automatically creating datasheets, me and my colleagues decided to develop a tool to do this. The 'alpha version' of this tool, called *Owl* was tailored towards information contained in the iGEM Registry of Standard Biological parts and could query the Registry to fill out a datasheet partially. The current 'beta version' is more general and can be amended to query arbitrary fields from multiple data sources and organize the information by type of information and uses LaTeXstyle files to typeset the document.

The Registry of Standard Biological Parts (http://parts.igem.org) is the largest open–source registry for synthetic biological parts. It is also the standard registry for the annual iGEM. As iGEM expands, many new entries and entry modifications are submitted each year. Given this already large and rapidly growing registry and other growing registries of synthetic genetic parts (JBEI https://registry.jbei.org/, JGI http://genome.jgi.doe.gov/, SynBERC registry.synberc.org, and BioFAB www.biofab.org), the question of how to best share and store data becomes very important to answer. It is critical that the synthetic biology community forms a concerted effort to share data on genetic parts and other biological components in a standard way.[15]

The Registry purposefully allows flexibility in the entries in both format and content, lending itself to a broad diversity of parts. One of the Registrys strengths is its ability to capture many types of synthetic biological parts, but it lacks consistent formatting and presentation that is required for machine readability and manual comparisons. The lack of a common format can also hinder new users and may impede them from adding useful information to the Registry. Furthermore, as a variety of software tools becomes available for biodesign automation, it is necessary to provide a unified format for a part datasheet so that the tools can leverage data stored in these sheets.

To address these problems, me and my colleagues created an online tool called *Owl* (www.owlcad.org) to generate electronic datasheets automatically, with a common format. This version of *Owl (alpha)* lays the groundwork for the automated generation of datasheets.

7.2.1 An Electronic Datasheet Generator - Alpha Version

Owl (www.owlcad.org) is a web-based tool that generates electronic datasheets for synthetic biological parts. A datasheet provides a quantitative and qualitative description of genetic device behavior that allows an engineer to determine if a part is suitable for a desired use[29]. *Owl* allows users to enter part information either automatically from pre-existing entries on the Registry or manually in the user interface. *Owl* currently uses Synthetic Biology Open Language visual (SBOLv) compliant images for part and device images and can link images from *Pigeon*[23] (www.pigeoncad.org) and *Raven*[14] (www.ravencad.org) onto a datasheet. *Owl* generates HTML pages in a standard format and can be saved as a PDF.

In consideration of previous datasheets[15,41] and common assays used to characterize biological systems, *Owl* datasheets are separated into five sections: (1) Basic Information, for part identification and visual representation; (2) Designer Information, for attributing authorship, providing contact information and the date; (3) Design Details, for detailed information about part function; (4) Assembly Information, for describing how the part was made; and (5) Assays, for presenting characterization data. We have provided three sections for assays: restriction mapping, flow cytometry, and a section where users can add their own type of assay.

To demonstrate *Owl*'s ability to represent a diversity of parts, we created several example datasheets for a variety of functionally different parts (datasheets are available online at www.owlcad.org). These datasheets represent several parts created by the CIDAR lab (www.cidarlab.org) as well as examples from literature to demonstrate Owls applicability to a diverse assortment of parts.

Required and Optional Fields

Owl datasheets have required fields based on the Registrys data model (**Figure 26**). This information is meant to represent the minimum information required to define a part with which data can be experimentally associated. Specifically, *Owl* requires that a DNA part be described sufficiently such that an assay could be performed upon it. Thus, all fields needed to describe a parts composition are required. *Owl* datasheets have five required fields, namely: Part Name, Sequence, Part Summary, Author(s), and Date. Completion of all other relevant fields is encouraged but optional.

Automated Population of Fields from Registry Pages

When generating a new datasheet from an existing Registry page, *Owl* parses informa-

Input Fields	**Input Information**
Basic Information	
Part Name*	Part name (e.g., BBa_R0040, pTetR)
Sequence*	DNA sequence for the part
Part Type	Basic or composite part
Pigeon Image	Graphic to visualize the part
Plasmid Map	Graphic to visualize the entire plasmid with the part
Part Summary*	Short written description of the part
Related Parts	List of related Part Names if applicable
Designer Information	
Author(s)*	List all researchers who worked on creating this part
Date*	Date the part was created
Team	iGEM team name; can also be used as Lab Name
Data Collectors	Researchers who collected data for this part
Affiliation	University or company affiliation
Contact	Contact person and email for questions about the part
Design Details	
Type	Descriptive part type (e.g., inducible promoter; NOR gate)
Design Components	Components used in the part (e.g., basic parts within a composite, restriction sites, sequence direction)
Vector	Plasmid backbone the part is cloned into
Additional Comments	Any other comments (e.g., plasmid resistance, required growth conditions)
Assembly Information	
Assembly Method(s)	List any methods used to create this part (e.g., BioBricks, MoClo, Gibson)
Assembly RFC	If available, list the Request For Comments number (e.g., RFC10)
Scars	Assembly scars remaining within or flanking the part once assembled (e.g., BioBricks mixed site)
Assembly Components	Items required for proper assembly (e.g., restriction enzymes, oligonucleotides)
Assembly Graph	Assembly graph or plan used to create this part (e.g., Raven assembly graph)
Chassis	Organism the plasmid containing the part was cloned into (e.g., *E. coli*, *S. cerevisiae*)
Strain	Specific strain of the organism (e.g., DH5-alpha, MG1655)
Additional Comments	Any other comments needed to understand how this part was assembled

* indicates required field

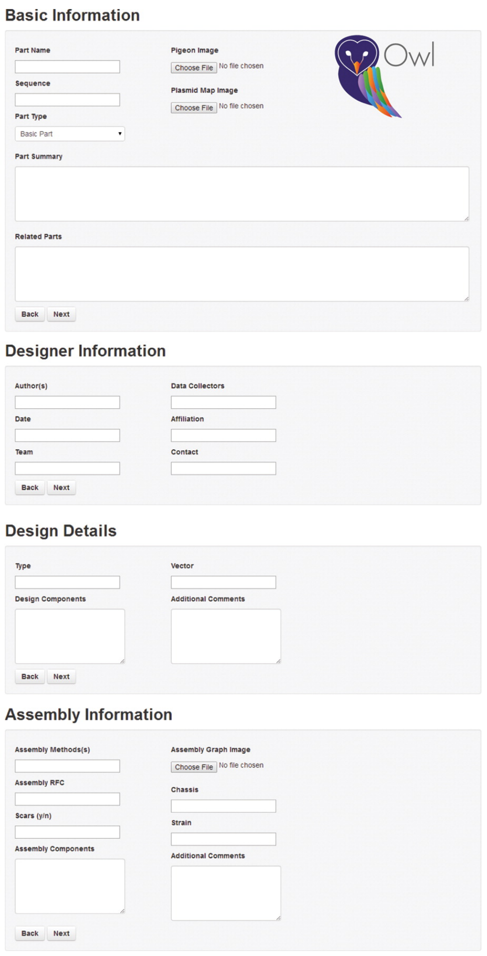


Figure 26: The *Owl* UI sections for Basic Information, Designer Information, Design Details, and Assembly Information. The input fields for each section are outlined on the left with descriptions for the input information for each field. The images on the right show the web interface UI from www.owlcad.org.

tion from the Registrys XML pages and autopopulates fields on the datasheet, namely: Part Name (ex: BBa B0034), Part Description, Part Type (ex: RBS), Date entered, Part Author, and Sequence. The user can then go through each section manually and add to or change the existing information.

While *Owl*'s goal is to automate the creation of datasheets, the task inherently poses the question of what a datasheet for synthetic biological parts should display. The default format of *Owl*'s datasheet may not be ideal for all cases; however, it is flexible and provides fields similar to those described in previous work[15,41] such as identifying information, circuit visualization, author contact information, assembly information, and characterization by gel electrophoresis and flow cytometry. Further considerations include the ability to present four kinds of data proposed in previous work:(4) static behavior, dynamic behavior, compatibility with other devices, and reliability of the device measured by the number of generations the device can uphold desired functionality.

7.2.2 An Electronic Datasheet - Beta Version

The initial release of the Alpha version of *Owl* received two key points of feedback: 1. It is too restrictive of the data types - a user with additional data fields might not fit into the categories defined in the first release. 2. It only searches the Registry of Parts and can only search for a small set of fields.

Although the initial tool was designed to be partially restrictive to ensure that a minimal amount of information was entered on each datasheet, it seemed that we had not struck the right balance between data requirements and flexibility. This was also the reason that only a small number of fields were searchable from the Registry of Parts. It was important that the next version of *Owl* be integrated with additional platforms and registries[45,101] to generate datasheets automatically as a user moves through experimental workflows.

To address these concerns, the *Owl* development team created editable, human-readable ‘‘configuration files" that can be modified for custom fields and wells with a limited set of data types. Since the stylistic formatting of the datasheets may not suit all users, editable style files are be used to automatically typeset datasheets. *Owl* was also extended to search custom fields in existing repositories. Finally, a Java API was created so that *Owl* can be linked more easily with other tools outside of the *Phoenix* environment.

Datasheets produced with the beta version can accomplish all of the same functionalities of the alpha version with the exception of explicit required fields. The thinking here is that other tools that link in with *Owl* can enforce internal null checks and this was not important to include explicitly in the this tool.

7.3 Design Tree Visualization

In large design hierarchies, it is important for a user to visualize where they currently stand in the process and which existing plasmids have data and which do not. In *Phoenix*, this is represented in the user interface as a collapsible tree that represents all of the nodes in a decomposed design. At any one of these nodes, the user can click on nodes or set of nodes to view electronic datasheets produced by *Owl*.

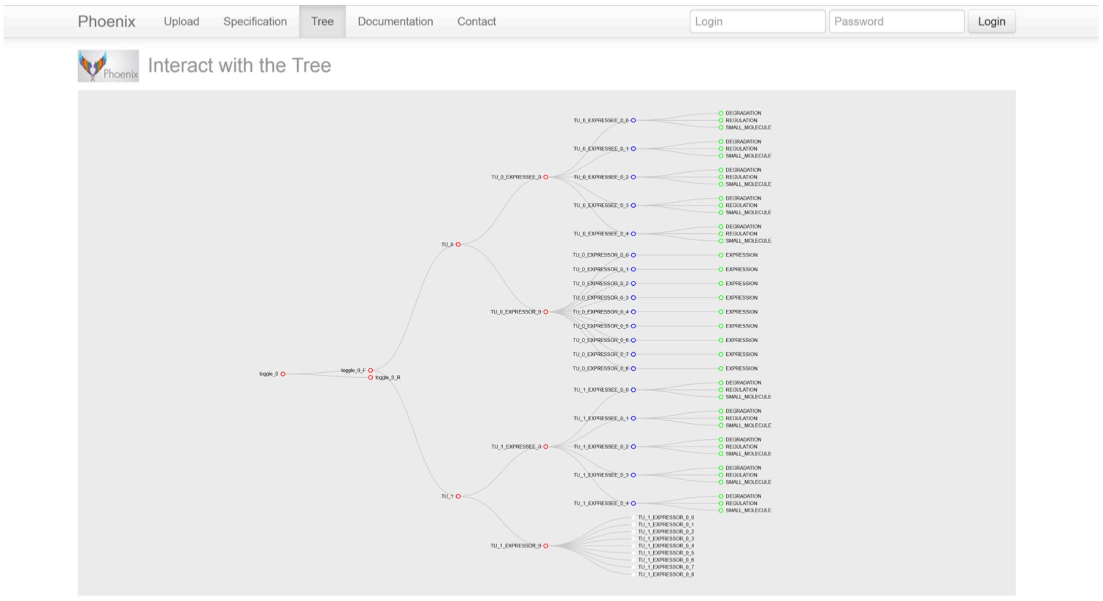


Figure 27: Screenshot of a design tree created by decomposing an input design in *Phoenix*

7.4 Data Visualization Summary

Owl is intended to serve the community by streamlining the creation of electronic datasheets that can be used to exchange important biological part information in a visually intuitive and user-friendly manner. Although other platforms formalize data exchange between machine users[45], there is a need for a consistently structured way to present data to a human user with which they can make decisions. I believe that *Owl* can help fill this gap.

8 Workflow Test Cases and Results

To validate that the overall *Phoenix* tool function, it was important to apply it to a set of example specifications. To do this, we built and tested a set of all *EXPRESSORs* and *EXPRESSEEs* for the functional terminals ('NOT gate', 'toggle switch', and 'oscillator') for our sequence feature library.

8.1 Unit testing of 'Classic' Networks

A unit test is a unit of work done upon a system to individually and independently scrutinize a single assumption about the behavior of that unit of work. In a genetic unit test, we define the single assumptions to be the enzymatic model for each node and the enzymatic parameters corresponding to each individual sequence. Thus, *Phoenix* adds testing components to perform unit tests to measure these parameters. In *Phoenix*, there are two general classes of unit tests - unit tests for *EXPRESSORs* and *EXPRESSEs* and unit tests for functional elements and complex function created with multiple functional elements.

Unit tests for *EXPRESSORs* and *EXPRESSEEs* are performed for creating the base characterization data for fitting enzymatic model parameters. Since *EXPRESSORs* are measured to estimate their expression parameter, an expression unit test is performed. Expression unit tests require steady-state fluorescence expression tests and a degradation unit test for each fluorescent protein expressed by the *EXPRESSOR*. Degradation unit tests are performed by adding chloramphenicol to liquid culture at time t = 0 to inhibit expression via transcriptional inhibition and measuring fluorescence over time. *EXPRESSEEs* necessitate unit tests for degradation, regulation, and (when appropriate) small molecule interaction. Regulation unit tests require inducing the expression of an *EXPRESSEE* with an inducible upstream promoter to measuring how the expression of a transciptional unit controlled by the promoter the *EXPRESSEE* regulates changes as a function of the amount of the *EXPRESSEE* present. We design these tests by building an additional regulation control plasmid and co-transforming it with the *EXPRESSEE* plasmid. These regulation control plasmids are built separately to save cloning cost and minimize failure mode opportunities in measurement. Small molecule unit tests are performed for *EXPRESSEEs* that are known to interact with a specific small molecule. For this test, a high quantity of the *EXPRESSEE* is induced and a titration curve of the small molecule is performed and measured over time.

For our library of 8 promoter-regulator pairs, 6 RBSs, 10 fluorescent proteins, 8 terminator and 2 vectors, I used *Phoenix* to decompose the designs and return the set of *EXPRESSORs* and *EXPRESSEEs* for a ‘forward-strand only’ linear design architecture.

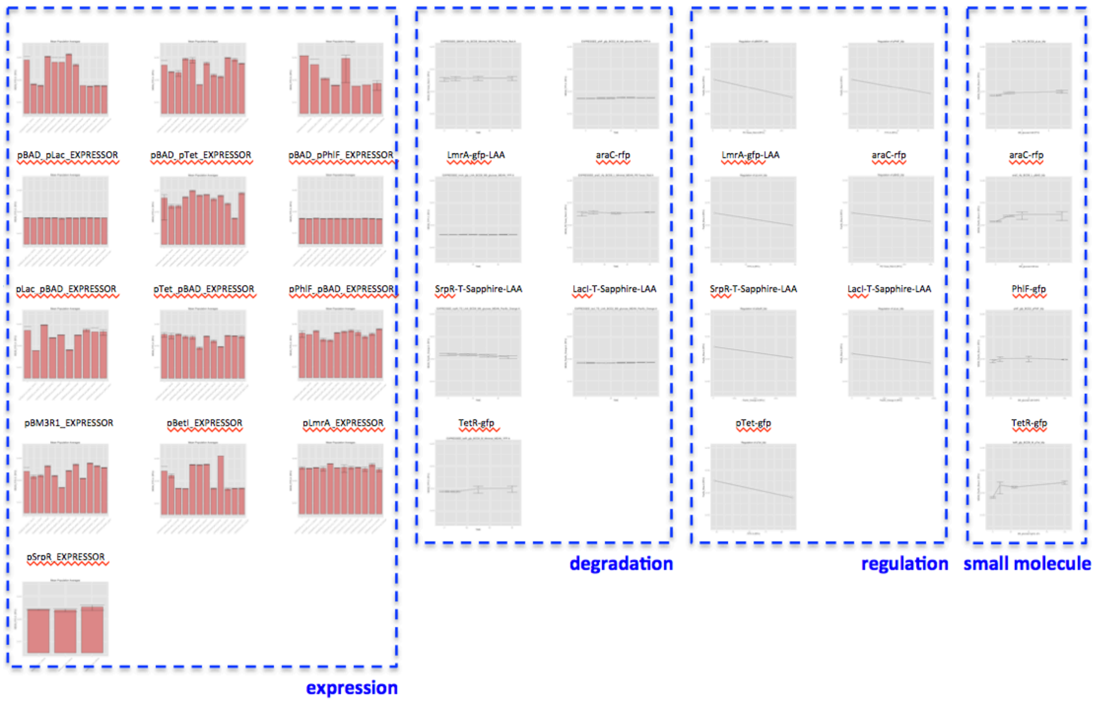


Figure 28: Preliminary data from expression tests for *EXPRESSORs* and degradation, regulation, and small molecule tests for *EXPRESSEEs*.

For the *EXPRESSORs*, I observed variable sequence and function outcomes, which agreed with our team’s findings in the multiplex assembly experiments described in Chapter

5.

Since the peptide linker sequences used in the *EXPRESSEEs* between the regulator and fluorescent protein had not yet been validated, I had to determine if these linker peptides interfered with the function of either the regulator or the fluorescent protein. It proved to be the case that the linker peptide we employed in the *Phoenix* workflow worked for nearly all *EXPRESSEEs*, so my colleagues and I advanced to measuring the *EXPRESSEEs* for their degradation, regulation and small molecule constants. These measurements resulted in a range of parameter values (data not shown in this document), so our group concluded that we had a rich design space of *EXPRESSORs* and *EXPRESSEEs* and used the fitted parameters to combinatorially simulate our specification targets.

Unit tests for compositional designs are based upon the functional specification as opposed to expert knowledge of the structural elements. For these cases, a test is done for time and performance bounds of the specification. Unit tests are performed at each node in the design tree for which there is a compositional design.

8.2 Building Networks via 'Brute Force'

To determine to what degree *Phoenix*'s computational methods improve outcomes, it was important to compare the success of these part assignments against a set of constructs that use the same assembly method and part arrangements, but with randomly assigned parts. To do this, I used the same ARGNs for the linear design architecture for the 'NOT gate', 'toggle switch' and 'oscillator' and determined all possible partially-assigned constructs where I only selected the promoter-regulator pairs to remain orthogonal where appropriate. Using these valid, partially-assigned GRNs, I multiplexed the the remaining components of the GRN to build randomly-assigned networks. The results of these tests are not presented in this document, but will be used as a baseline to compare against *Phoenix*-designed networks in future work.

8.3 Unit-testing Failure Modes

Since the initial set of assignments were all in the same linear arrangements, they were all subject to the same failure mode - transcriptional read-through. To test the wider scope of defined failure modes, we built and tested ‘NOT gates’, ‘toggle switches’ and ‘oscillators’ with non-linear arrangements.

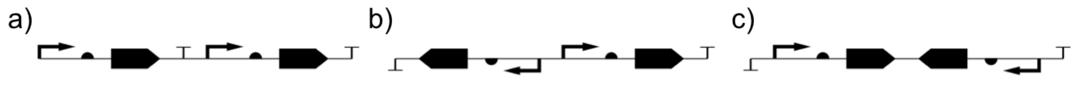


Figure 29: Arrangements for two-transcriptional unit constructs that present different common failure modes. (a) The linear arrangement is sensitive to transcriptional red-through from the first transcriptional unit into the second. (b) The alternating-strand transcriptional unit architecture introduce a supercoiling failure mode created by adjacent promoters. (c) This arrangement exposes the failure mode of transcriptional interference where polymerases on either strand might collide during transcription and affect expression.

For two-transcriptional-unit constructs such as these, there are a total of 12 valid arrangements based upon our structural grammar. To test more of these arrangements, we selected three more different types of architectures and used these as input to build alternative sets of *EXPRESSORs* that expose different types of failure modes. The outcomes of these functional tests demonstrated that different structural part arrangements with the same sequences yielded different expression rates, but the data for these experiments are not detailed in this document.

9 Project Summary and Impact

9.1 Project Summary

This thesis explores questions of what types of project should be built with synthetic biology and describes a set of software tools and workflows for accomplishing these project goals for genetic regulatory networks in *E. coli*. I described a project in which we probe a variety of projects for risk assessment and biosecurity concerns and then described another project that examined a project that poses problems of risk and biosecurity in detail. In the rest of the chapters, I discussed my technical work in building two software tools to solve specific sub-problems in the software tool space, namely *Raven* for creating assembly plans and *Owl* for creating electronic datasheets. The rest of this thesis describes the creation of *Phoenix*, a tool that ties together many existing software tools for specific tasks of a synthetic biology workflow in addition to providing algorithmic solutions to a number of unsolved sub-problems in the workflow.

9.2 Impact of Work

Phoenix represents one of the first closed-loop workflow tools in the synthetic biology space to automate a majority of design decisions in genetic regulatory network creation via experimental instructions and data entry. In contrast to other existing software suites, *Phoenix* is not designed as an open canvas where a user has flexibility to create nearly any type of genetic network, but still has to make all design decisions manually. Rather, the ultimate goal of *Phoenix* is to advance the goals of the bio-design automation community and work towards powerful computer-aided design that simplifies the design process for a user and enables higher level of design abstraction.

It is well known that creating synthetic genetic regulatory networks is no easy task and some would agree that many aspects of manual synthetic biology methodologies have already reached or will soon reach their limits - it will not be possible to solve significantly more complex problems without computational tools and computer-aided design. The central goal of this thesis is to make significant progress on this front and open the door to larger design goals while instituting a philosophy of careful consideration of which projects to pursue.

10 Methods

10.1 Computational

Back-end algorithms in all of the tools in this thesis were implemented in Java. The *Raven*, *Owl*, and *Phoenix* UIs were implemented in Javascript using jQuery and Bootstrap libraries. Scripts for recommended, discouraged and required parts in *Raven* are implemented in *Eugene*[24]. Design tree graphs in *Phoenix* are generated using d3.js libraries and automatically-generated construct glyphs are made from scripts to pigeoncad.org[23]. Analytics scripts are implemented *R* and rely heavily on the *Bioconductor* packages.

10.2 Experimental

10.2.1 Materials

Cloning enzymes and buffers ordered from New England Biolabs, Ipswich, MA, USA and Promega Corporation, Madison, WI, USA. Epoch kits and protocols used to extract and prepare DNA. Oligonucleotides synthesized by IDT. DIC counter plasmid templates acquired from Timothy Lu. All reactions performed in an Eppendorf Mastercycler ep thermocylcer (Eppendorf North America, Westbury, NY, USA).

10.2.2 MoClo DNA Assembly

10.2.3 Cloning Destination Vectors

The lacZα fragment was PCR amplified from a lacZα-containing cloning vector (pMJS2AF, donated by Michael Smanski) and subsequently cloned into three backbones, depending on the MoClo level: level 0 used pSB1A2, level 1 used pSB1K3, and level 2 used pSB1A2. DNA containing the lacZα fragment was used as template for PCR reactions. PCR reactions with 5X Phusion HF Buffer, 100 uM dNTPs, Phusion DNA Polymerase, 5% DMSO, 1 mM MgCl2 (New England Biolabs, Ipswich, MA, USA), and sterile diH2O. Reactions were performed using the following parameters: one denaturation step at 95C for 5 min, followed by 30 extension cycles ($95\,^{\circ}\mathrm{C}$ 20 sec., $61\,^{\circ}\mathrm{C}$ 20 sec., $72\,^{\circ}\mathrm{C}$ 15 sec.), a final 5 min extension step at $72\,^{\circ}\mathrm{C}$ and then incubation at $4\,^{\circ}\mathrm{C}$. PCR products over 100bp were purified using either the QIAquick PCR Purification Kit (Qiagen Inc., Valencia, CA, USA) or GenCatch PCR Purification Kit (Epoch Life Sciences, Sugar Land, TX, USA) according to the manufacturer's protocol. PCR products and pSB1K3 and pSB1A2 vectors were digested with SpeI enzyme (NEB) according to the manufacturer's protocol using up to 500 ng DNA. Restriction digestions were purified using the QIAquick PCR Purification Kit (Qiagen) following the manufacturer's protocol. Ligation reactions were performed with T4 DNA ligase (NEB) following the manufacturer's protocol with a 3:1 insert part to vector backbone ratio.

10.2.4 PCR Amplifying Level 0 Parts

Level 0 target sequences over 32 bps were PCR amplified from the DIC 3-Counter multiple-inducer (Donated by Timothy Lu). Level 0 sequences for parts smaller than 34 bps, annealing oligonucleotides were designed and annealed with the following parameters: $95\,^{\circ}\mathrm{C}$ for 3 min, then 55 x (-1 $^{\circ}\mathrm{C}$ every 30 sec.), then incubation at $4\,^{\circ}\mathrm{C}$. (see Supplementary Raven files for part sequences). Reactions were performed using the following parameters: one denaturation step at $95\,^{\circ}\mathrm{C}$ for 5 min, followed by 30 extension cycles ($95\,^{\circ}\mathrm{C}$ 20 sec., $61\,^{\circ}\mathrm{C}$ 20 sec., $72\,^{\circ}\mathrm{C}$ 30 sec.), a final 5 min. extension step at $72\,^{\circ}\mathrm{C}$ and then held at $4\,^{\circ}\mathrm{C}$.

10.2.5 MoClo Cloning Protocol

Each MoClo reaction had the following contents: 40 fmol of each DNA component (DNA PCR Product or previously made MoClo DNA Parts, and the appropriate Destination Vector), BsaI or BbsI (BsaI for Level 1, BbsI for Level 0 and Level 2; NEB), high concentration T4 DNA ligase (C M1794, Promega, Madison, WI, USA), 1 X T4 DNA Ligase Buffer (Promega), and sterile, diH2O. Reactions performed using the following parameters: 25-35 cycles ($37\,^{\circ}\mathrm{C}$ 1.5 min., $16\,^{\circ}\mathrm{C}$ 3 min.), followed by $50\,^{\circ}\mathrm{C}$ for 5 minutes and $80\,^{\circ}\mathrm{C}$ for 10 minutes and then held at $4\,^{\circ}\mathrm{C}$ until transformed. Level 0 reactions were done for 25 cycles while Level 1 and 2 reactions were done for 25-30 cycles. Transformations into Alpha Select Gold Efficiency E. coli cells (Bioline USA Inc., Taunton, MA, USA), DH5α-Z1, and epi300 competent E. coli. Transformations were heat shocked at $42\,^{\circ}\mathrm{C}$ for 45 sec. and recovered in SOC media for 1 hour at $37\,^{\circ}\mathrm{C}$, 300rpm.

10.2.6 Primer Design

Primers for MoClo[99] assembly designed in the following format for parts larger than 24 bp: NN - BpiI recognition site - NN - 1234 - part - 5768 - NN - BpiI recognition site - NN. Forward primers: 5′ NN - GAAGAC - NN - [Overhang Sequence] - [first 24 bp of part] 3′. Reverse primers: 5′ [last 24bps of gene] - [Overhang Sequence] - NN - GTCTTC - NN 3′. For parts smaller than 24bp, annealing primers were designed that adhere to the preceding format.

References

[1] (1925). *Geneva Protocol*. United Nations.

[2] (1972). *Convention on the Prohibition of the Development, Production and Stockpiling of Bacteriological (Biological) and Toxin Weapons and on their Destruction*. United Nations.

[3] (2002). *NIH Guidelines for Research Involving Recombinant DNA Molecules*. National Institutes of Health.

[4] (2003). *Cartegena Protocol on Biosafety to the Convention on Biological Diversity*. United Nations.

[5] (2004). *Laboratory Biosafety Manual*. World Health Organization, 3rd edition.

[6] (2009). *Regulation of Genetically Engineered Animals Containing Heritable Recombinant DNA Constructs: Final Guidance*. Center for Veterinary Medicine (CVM), Food and Drug Administration, US Department of Human Health and Human Services, FDA, Rockville, MD.

[7] (2012). *Guidelines for Transfers of Sensitive Chemical and Biological Materials*. Australia Group.

[8] (2012). *The Regulation of Synthetic Biology: A Guide to United States and European Union Regulations, Rules, and Guidelines*, National Science Foundation, Arlington, VA. Synthetic Biology Engineering Research Center (Synberc).

[9] (2013). *United States Government Policy for Institutional Oversight of Life Sciences Dual use Research of Concern, Draft Notice*. U.S. Science and Technology Office.

[10] (2014). *Creating a Research Agenda for the Ecological Implications of Synthetic Biology*, MIT Center for International Studies, Cambridge, MA and Wordrow Wilson International Center for Scholars, Washington, DC.

[11] Anderson, J. C., Clarke, E. J., Arkin, A. P., and Voigt, C. A. (2006). Environmentally controlled invasion of cancer cells by engineered bacteria. *Journal of Molecular Biology*, 355(4):619--627.

[12] Anderson, J. C., Voigt, C. A., and Arkin, A. P. (2007). Environmental signal integration by a modular and gate. *Molecular Systems Biology*, 3(133):133.

[13] Appleton, E. et al. (2014a). Owl: Electronic datasheet generator. *ACS Synthetic Biology*, 3(12):966--968.

[14] Appleton, E., Tao, J., Haddock, T., and Densmore, D. (2014b). Interactive assembly algorithms for molecular cloning. *Nature methods*, 11(6):657--662.

[15] Arkin, A. (2008). Setting the standard in synthetic biology. *Nature biotechnology*, 26:771--774.

[16] Baltimore, B., Berg, P., Botchan, M., Carroll, D., Charo, R. A., Church, G., Corn, J. E., Daley, G. Q., Doudna, J. A., Fenner, M., et al. (2015). A prudent path forward for genomic engineering and germline gene modification. *Science*, 348(6230):36--38.

[17] Basu, S., Gerchman, Y., Collins, C. H., Arnold, F. H., and Weiss, R. (2005). A synthetic multicellular system for programmed pattern formation. *Nature*, 434(7037):1130--1134.

[18] Batt, G., Ropers, D., de Jong, H., Geiselmann, J., Mateescu, R., Page, M., and Schneider, D. (2005). Validation of qualitative models of genetic regulatory networks by model checking: analysis of the nutritional stress response in escherichia coli. *Bioinformatics*, 21.

[19] Bayer, T. S. and Smolke, C. D. (2005). Programmable ligand-controlled riboregulators of eukaryotic gene expression. *Nature biotechnology*, 23(3):337--343.

[20] Beal, J., Lu, T., and Weiss, R. (2011). Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks. *PLoS ONE*, 6(8):e22490.

[21] Beal, J., Weiss, R., Densmore, D., Adler, A., Appleton, E., Babb, J., Bhatia, S., Davidsohn, N., Haddock, T., Loyall, J., Schantz, R., Vasilev, V., and Yaman, F. (2012). An end-to-end workflow for engineering of biological networks from high-level specifications. *ACS Synthetic Biology*, 1:317--331.

[22] Benedict, M., D'Abbs, P., Dobson, S., Gottlieb, M., Harrington, L., Higgs, S., James, A., James, S., Knols, B., Lavery, J., et al. (2008). Guidance for contained field trials of vector mosquitoes engineered to contain a gene drive system: recommendations of a scientific working group. *Vector-Borne and Zoonotic Diseases*, 8(2):127--166.

[23] Bhatia, S. and Densmore, D. (2013). Pigeon: a design visualizer for synthetic biology. *ACS synthetic biology*, 2(6):348--350.

[24] Bilitchenko, L., Liu, A., Cheung, S., Weeding, E., Xia, B., Leguia, M., Anderson, J. C., and Densmore, D. (2011). Eugene: A domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PLoS ONE*, 6(4):e18882.

[25] Blakes, J., Raz, O., Feige, U., Bacardit, J., Widera, P., Ben-Yehezkel, T., Shapiro, E., and Krasnogor, N. (2014). Heuristic for maximizing dna reuse in synthetic dna library assembly. *ACS Synthetic Biology*, 3(8):529--542.

[26] Bonnet, J., Subsoontorn, and Endy, D. (2012). Rewritable digital storage in live cells via engineered control or recombination directionality. *Proceedings of the National Academy of Sciences of the United States of America*, 109(23):8884--8889.

[27] Bonnet, J., Yin, P., Ortiz, M. E., Subsoontorn, P., and Endy, D. (2013). Amplifying genetic logic gates. *Science*, 340(6132):599--603.

[28] Burt, A. (2003). Site-specific selfish genes as tools for the control and genetic engineering of natural populations. *Proceedings of the Royal Society of London B: Biological Sciences*, 270(1518):921--928.

[29] Canton, B., Labno, A., and Endy, D. (2008). Refinement and standardization of synthetic biological parts and devices. *Nature biotechnology*, 26(7):787--793.

[30] Cao, Y., Lu, H.-M., and Liang, J. (2010). Probability landscape of heritable and robust epigenetic state of lysogeny in phage lambda. *Proceedings of the National Academy of Sciences*, 107(43):18445--18450.

[31] Chandran, D., Bergmann, F., and Sauro, H. (2009). Tinkercell: modular cad tool for synthetic biology. *Journal of Biological Engineering*, 3(1):19.

[32] Cohen, S. N., Chang, A. C., Boyer, H. W., and Helling, R. B. (1973). Construction of biologically functional bacterial plasmids in vitro. *Proceedings of the National Academy of Sciences*, 70(11):3240--3244.

[33] Cong, L., Ran, F. A., Cox, D., Lin, S., Barretto, R., Habib, N., Hsu, P. D., Wu, X., Jiang, W., Marraffini, L. A., et al. (2013). Multiplex genome engineering using crispr/cas systems. *Science*, 339(6121):819--823.

[34] Czar, M. J., Cai, Y., and Peccoud, J. (2009). Writing dna with genocad. *Nucleic Acids Research*, 37(Web Server issue):W40--W47.

[35] Danino, T., Mondragon-Palomino, O., Tsimring, L., and Hasty, J. (2009). A synchronized quorum of genetic clocks. *Nature*, 463(7279):326--330.

[36] Densmore, D. (2012). Bio-design automation: Nobody said it would be easy. *ACS synthetic biology*, 1(8):296--296.

[37] Densmore, D., Hsiau, T. H. C., Kittleson, J. T., DeLoache, W., Batten, C., and Anderson, J. C. (2010). Algorithms for automated dna assembly. *Nucleic Acids Research*, 38(8):2607--2616.

[38] Ellis, B., Haaland, P., Hahne, F., Meur, N. L., Gopalakrishnan, N., and Spidlen, J. (2014). *flowCore: Basic structures for flow cytometry data*. R package version 1.32.2.

[39] Ellis, T., Wang, X., and Collins, J. (2009). Diversity-based, model-guided construction of synthetic gene networks with predicted functions. *Nature biotechnology*, 27(5):465--471.

[40] Elowitz, M. and Leibler, S. (2000). A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767):335--338.

[41] Endy, D. (2005). Foundations for engineering biology. *Nature*, 438(7067):449--453.

[42] Engler, C., Kandzia, R., and Marillonnet, S. (2008). A one pot, one step, precision cloning method with high throughput capability. *PLoS ONE*, 3(11):e3647.

[43] Esvelt, K. M., Smidler, A. L., Catteruccia, F., and Church, G. M. (2014). Concerning rna-guided gene drives for the alteration of wild populations. *Elife*, 3:e03401.

[44] Friedland, A. E., Lu, T. K., Wang, X., Shi, D., Church, G., and Collins, J. J. (2009). Synthetic gene networks that count. *Science*, 324:1199 – 1202.

[45] Galdzicki, M. et al. (2014). The synthetic biology open language (sbol) provides a community standard for communicating designs in synthetic biology. *Nature biotechnology*, 32(6):545--550.

[46] Galdzicki, M., Wilson, M. L., Rodriguez, C. A., Adam, L., Adler, A., Anderson, J. C., Beal, J., Chandran, D., Densmore, D., Drory, O. A., Endy, D., Gennari, J. H., Grünberg, R., Ham, T. S., Kuchinsky, A., Lux, M. W., Madsen, C., Misirli, G., Myers, C. J., Peccoud, J., Plahar, H., Pocock, M. R., Roehner, N., Smith, T. F., Stan, G.-B., Villalobos, A., Wipat, A., , and Sauro, H. M. (2011). Synthetic Biology Open Language (SBOL) Version 1.0.0. RFC 85.

[47] Gardner, T. S., Cantor, C. R., and Collins, J. J. (2000). Construction of a genetic toggle switch in escherichia coli. *Nature*, 403(6767):339--342.

[48] Gibson, D. G., Young, L., Chuang, R.-Y., Venter, J. C., Hutchison, C. A., and Smith, H. O. (2009). Enzymatic assembly of DNA molecules up to several hundred kilobases. *Nature methods*, 6(5):343--345.

[49] Guet, C. C., Elowitz, M. B., Hsing, W., and Leibler, S. (2002). Combinatorial synthesis of genetic networks. *Science*, 296(5572):1466--1470.

[50] Hayden, E. C. (2015). Synthetic biologists seek standards for nascent field. *Nature*, 520(7546):141--142.

[51] Herzenberg, L. A., Tung, J., Moore, W. A., Herzenberg, L. A., and Parks, D. R. (2006). Interpreting flow cytometry data: a guide for the perplexed. *Nature immunology*, 7(7):681--685.

[52] Hillson, N., Rosengarten, R. D., and Keasling, J. (2011). j5 dna assembly design automation software. *ACS Synthetic Biology*, 1:14--21.

[53] Hoops, S., Sahle, S., Gauges, R., Lee, C., Pahle, J., Simus, N., Singhal, M., Xu, L., Mendes, P., and Kummer, U. (2006). Copasia complex pathway simulator. *Bioinformatics*, 22(24):3067--3074.

[54] Isaacs, F. J., Dwyer, D. J., Ding, C., Pervouchine, D. D., Cantor, C. R., and Collins, J. J. (2004). Engineered riboregulators enable post-transcriptional control of gene expression. *Nature biotechnology*, 22(7):841--847.

[55] Kahl, L. J. and Endy, D. (2013). A survey of enabling technologies in synthetic biology. *Journal of biological engineering*, 7(1):1--19.

[56] Keating, S. and Le Novère, N. (2013). Supporting sbml as a model exchange format in software applications. In Schneider, M. V., editor, *In Silico Systems Biology*, volume 1021 of *Methods in Molecular Biology*, pages 201--225. Humana Press.

[57] Kelly, J. R., Rubin, A. J., Davis, J. H., Ajo-Franklin, C. M., Cumbers, J., Czar, M. J., de Mora, K., Glieberman, A. L., Monie, D. D., and Endy, D. (2009). Measuring the activity of biobrick promoters using an in vivo reference standard. *Journal of Biological Engineering*, 3(4).

[58] Khalil, A. S. and Collins, J. J. (2010). Synthetic biology: applications come of age. *Nature Reviews Genetics*, 11(5):367--379.

[59] Kuiken, T., Dana, G., Oye, K., and Rejeski, D. (2014). Shaping ecological risk research for synthetic biology. *Journal of Environmental Studies and Sciences*, 4(3):191--199.

[60] Kuwahara, H., Madsen, C., Mura, I., Myers, C., Tejeda, A., and Winstead, C. (2010). Effecient stochastic simulation to analyze targeted properties of biological systems, stochastic control. InTech.

[61] Lanphier, E., Urnov, F., Haecker, S. E., Werner, M., and Smolenski, J. (2015). Don't edit the human germ line. *Nature*, 519(7544):410.

[62] Li, M. Z. and Elledge, S. J. (2007). Harnessing homologous recombination in vitro to generate recombinant dna via slic. *Nature methods*, 4(3):251--256.

[63] Linshiz, G., Stawski, N., Goyal, G., Bi, C., Poust, S., Sharma, M., Mutalik, V., Keasling, J. D., and Hillson, N. J. (2014). Pr-pr: Cross-platform laboratory automation system. *ACS synthetic biology*, 3(8):515--524.

[64] Lou, C., Stanton, B., Chen, Y.-J., Munsky, B., and Voigt, C. A. (2012). Ribozyme-based insulator parts buffer synthetic circuits from genetic context. *Nature biotechnology*, 30(11):1137--1141.

[65] Lucks, J. B., Qi, L., Mutalik, V. K., Wang, D., and Arkin, A. P. (2011). Versatile rna-sensing transcriptional regulators for engineering genetic networks. *Proceedings of the National Academy of Sciences*, 108(21):8617--8622.

[66] MacDonald, J. T., Barnes, C., Kitney, R. I., Freemont, P. S., and Stan, G.-B. V. (2011). Computational design approaches and tools for synthetic biology. *Integrative biology quantitative biosciences from nano to macro*, 3(2):97--108.

[67] Mali, P., Yang, L., Esvelt, K. M., Aach, J., Guell, M., DiCarlo, J. E., Norville, J. E., and Church, G. M. (2013). Rna-guided human genome engineering via cas9. *Science*, 339(6121):823--826.

[68] Marchisio, M. A. and Stelling, J. (2009). Computational design tools for synthetic biology. *Current Opinion in Biotechnology*, 20(4):479--485.

[69] Moon, T. S., Lou, C., Tamsir, A., Stanton, B. C., and Voigt, C. A. (2012). Genetic programs constructed from layered logic gates in single cells. *Nature*, 491:249--253.

[70] Mouchet, M. A., Villeger, S., Mason, N. W., and Mouillot, D. (2010). Functional diversity measures: an overview of their redundancy and their ability to discriminate community assembly rules. *Functional Ecology*, 24(4):867--876.

[71] Mutalik, V. K., Guimaraes, J. C., Cambray, G., Lam, C., Christoffersen, M. J., Mai, Q.-A., Tran, A. B., Paull, M., Keasling, J. D., Arkin, A. P., et al. (2013). Precise and reliable gene expression via standard transcription and translation initiation elements. *Nature methods*, 10(4):354--360.

[72] Myers, C. J. (2009). *Engineering Genetic Circuits*. Chapman and Hall.

[73] Oberortner, E., Bhatia, S., Lindgren, E., and Densmore, D. (2014). A rule-based design specification language for synthetic biology. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 11(3):25.

[74] Oberortner, E. and Densmore, D. (2015). Web-based software tool for constraint-based design specification of synthetic biological systems. *ACS Synthetic Biology*, 4(6):757--760.

[75] Olson, E. J., Hartsough, L. A., Landry, B. P., Shroff, R., and Tabor, J. J. (2014). Characterizing bacterial gene circuit dynamics with optically programmed gene expression signals. *Nature methods*, 11(4):449--455.

[76] Oye, K. A., Esvelt, K., Appleton, E., Catteruccia, F., Church, G., Kuiken, T., Lightfoot, S. B., McNamara, J., Smidler, A., and Collins, J. P. (2014). Regulating gene drives. *Science*, 345(6197):626--628.

[77] Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789--810.

[78] Pedersen, M. and Phillips, A. (2009). Towards programming languages for genetic engineering of living cells. *Journal of the Royal Society Interface the Royal Society*, 6(4):S437--S450.

[79] Perfetto, S. P., Chattopadhyay, P. K., and Roederer, M. (2004). Seventeen-colour flow cytometry: unravelling the immune system. *Nature reviews immunology*, 4(8):648--655.

[80] Pontikos, N. (2013). *flowBeads: Analysis of flow bead data.* R package version 1.4.0.

[81] Quan, J. and Tian, J. (2009). Circular polymerase extension cloning of complex gene libraries and pathways. *PLoS ONE*, 4(7):e6441.

[82] Reeves, R. G., Denton, J. A., Santucci, F., Bryk, J., and Reed, F. A. (2012). Scientific standards and the regulation of genetically modified insects. *PLoS Negl Trop Dis*, 6(1):e1502.

[83] Ro, D.-K., Paradise, E. M., Ouellet, M., Fisher, K. J., Newman, K. L., Ndungu, J. M., Ho, K. A., Eachus, R. A., Ham, T. S., Kirby, J., Chang, M. C. Y., Withers, S. T., Shiba, Y., Sarpong, R., and Keasling, J. D. (2006). Production of the antimalarial drug precursor artemisinic acid in engineered yeast. *Nature*, 440(10.1038):940--943.

[84] Rodrigo, G. and Jaramillo, A. (2013). Autobiocad: Full biodesign automation of genetic circuits. *ACS Synthetic Biology*, 2(5):230--236.

[85] Roederer, M. (2001). Spectral compensation for flow cytometry: visualization artifacts, limitations, and caveats. *Cytometry*, 45(3):194--205.

[86] Roederer, M., De Rosa, S., Gerstein, R., Anderson, M., Bigos, M., Stovel, R., Nozaki, T., Parks, D., Herzenberg, L., and Herzenberg, L. (1997). 8 color, 10-parameter flow cytometry to elucidate complex leukocyte heterogeneity. *Cytometry*, 29(4):328--339.

[87] Saiki, R. K., Gelfand, D. H., Stoffel, S., Scharf, S. J., Higuchi, R., Horn, G. T., Mullis, K. B., and Erlich, H. A. (1988). Primer-directed enzymatic amplification of dna with a thermostable dna polymerase. *Science*, 239(4839):487--491.

[88] Salis, H. M., Mirsky, E. A., and Voigt, C. A. (2009). Automated design of synthetic ribosome binding sites to control protein expression. *Nature biotechnology*, 27(10):946--950.

[89] Sarrion-Perdigones, A., Falconi, E. E., Zandalinas, S. I., Juarez, P., Fernandez-del Carmen, A., Granell, A., and Orzaez, D. (2011). Goldenbraid: An iterative cloning system for standardized assembly of reusable genetic modules. *PLoS ONE*, 6(7):e21622.

[90] Shetty, R. P., Endy, D., and Knight, T. F. (2008). Engineering biobrick vectors from biobrick parts. *Journal of Biological Engineering*, 2(5):1--12.

[91] Siuti, P., Yazbek, J., and Lu, T. K. (2013). Synthetic circuits integrating logic and memory in living cells. *Nature biotechnology*, 31(5):448--452.

[92] Smanski, M. J., Bhatia, S., Zhao, D., Park, Y., Woodruff, L. B., Giannoukos, G., Ciulla, D., Busby, M., Calderon, J., Nicol, R., et al. (2014). Functional optimization of gene clusters by combinatorial design and assembly. *Nature biotechnology*.

[93] Smith, H. O. and Welcox, K. (1970). A restriction enzyme from hemophilus influenzae: I. purification and general properties. *Journal of molecular biology*, 51(2):379--391.

[94] Smolke, C. D. (2009). Building outside of the box: igem and the biobricks foundation. *Nature biotechnology*, 27(12):1099--1102.

[95] Stricker, J., Cookson, S., Bennett, M. R., Mather, W. H., Tsimring, L. S., and Hasty, J. (2008). A fast, robust and tunable synthetic gene oscillator. *Nature*, 456(7221):516--519.

[96] Tabor, J. J., Salis, H. M., Simpson, Z. B., Chevalier, A. A., Levskaya, Marcotte, E. M., Voigt, C. A., and Ellington, A. D. (2009). A synthetic genetic edge detection program. *Cell*, 137:1272--1281.

[97] Tamsir, A., Tabor, J. J., and Voigt, C. A. (2011). Robust multicellular computing using genetically encoded NOR gates and chemical 'wires'. *Nature*, 469:212--215.

[98] Watson, J. D., Crick, F. H., et al. (1953). Molecular structure of nucleic acids. *Nature*, 171(4356):737--738.

[99] Weber, E., Engler, C., Gruetzner, R., Werner, S., and Marillonet, S. (2010). A modular cloning system for standardized assembly of multigene constructs. *PloS ONE*, 6(2):e16765.

[100] Weiss, B. and Richardson, C. C. (1967). Enzymatic breakage and joining of deoxyribonucleic acid, i. repair of single-strand breaks in dna by an enzyme system from escherichia coli infected with t4 bacteriophage. *Proceedings of the National Academy of Sciences of the United States of America*, 57(4):1021.

[101] Xia, B., Bhatia, S., Bubenheim, B., Dadgar, M., Densmore, D., and Anderson, J. C. (2011). Developer's and user's guide to clotho v2.0 a software platform for the creation of synthetic biological systems. *Methods in Enzymology*, 498:97--135.

[102] Yaman, F., Bhatia, S., Adler, A., Densmore, D., and Beal, J. (2012). Automated selection of synthetic biology parts for genetic regulatory networks. *ACS synthetic biology*, 1(8):332--344.

[103] Yordanov, B., Appleton, E., Ganguly, R., Gol, E., Carr, S., Bhatia, S., Haddock, T., Belta, C., and Densmore, D. (2012). Experimentally driven verification of synthetic biological circuits. In *Design and Test in Europe*, Dresden, Germany.

[104] Yordanov, B. and Belta, C. (2011). A formal verification approch to the design of synthetic gene networks. In *Proceedings of the 50th IEEE Conference on Decision and Control (CDC)*.

EVAN M. APPLETON

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]