Rule Based Constraints for the Construction of Genetic Devices

Douglas Densmore*, Joshua T. Kittleson†, Lesia Bilitchenko‡, Adam Liu§ and J.Christopher Anderson†
*Synthetic Biology Engineering Research Center, Joint BioEnergy Institute, Emeryville, CA 94608
Email: dmdensmore@lbl.gov
†Department of Bioengineering,§Department of EECS, University of California, Berkeley, CA 94720
Email: {jkittles, adam_z_liu, jcanderson}@berkeley.edu
‡Department of Computer Science, Cal Poly Pomona, CA 91768
Email: lbilitchenko@csupomona.edu

***Abstract*— The construction of composite genetic devices from primitive parts is a key activity in synthetic biology. Currently there does not exist a formal method to specify constraints on the construction of these devices. These constraints would help enable an automated design flow from device specification to physical assembly. This paper examines the laboratory creation of variations of a particular genetic device called a phagemid. We illustrate how lessons learned empirically from the non-functional designs can be captured formally as constraints in a newly created domain specific language called "Eugene". These constraints will prevent many faulty constructions automatically in the future saving time and money while increasing design abstraction and productivity.**

I. Introduction

The automated design of electronic circuits has matured to the point where it is often taken for granted. The standard design flow for application specific integrated circuits (ASICs) beginning with a Verilog or VHDL description of the design, proceeding through the logic synthesis process, and resulting in a GDSII layout file is largely a highly optimized, standardized, and completely automated process [1]. This automation fueled a large part of 2008's 249 billion dollar semiconductor industry [2]. All the more impressive are these designs consist of millions of individual components all operating at the nano-scale.

Now imagine the design of a genetic device created from biological material. This device could similarly be described at a high level regarding its functionality, its basic building blocks, and the chemical processes needed to make it a reality. Moreover, as opposed to its electrical counterparts it would not consist of millions of components but only a handful. The automation of these devices however is virtually non-existent. The lack of automation not only makes the design process more tedious but also iterative and error prone. This paper begins to examine at how we can start down the path of automating such designs with the high level specification of genetic devices along with constraints on their construction. The idea is to leverage concepts from electronic design automation and apply them in novel ways to the design of genetic devices.

In this paper we will describe a genetic device called a *phagemid* that was created in a laboratory environment in seven alternate forms. Only one of these functioned correctly. The six non-functional designs represent wasted time and money yet provide valuable insight into the form of constraints which we can use to prevent faulty designs in the future. We capture these constraints in a domain specific language we created called Eugene [3]. We then use Eugene and a design environment called Clotho [4] to show that these newly created constraints will prevent these same types of faulty designs from being created in the future.

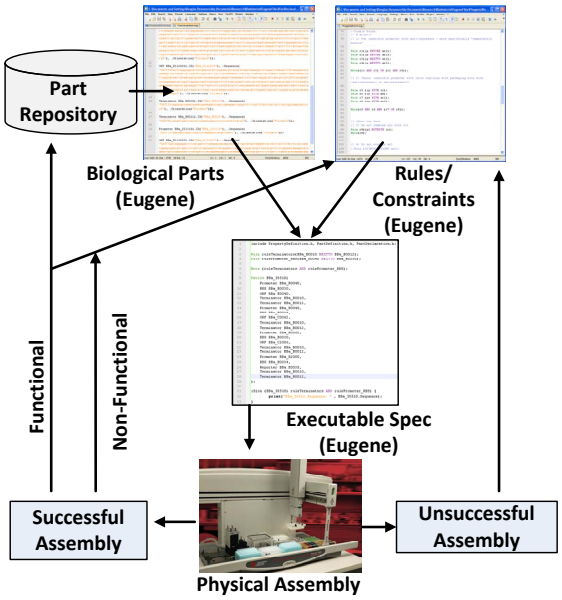

Fig. 1. Constraint Based Genetic Device Design Flow

Figure 1 illustrates the general design process. Biological building blocks (well characterized DNA segments) are captured in a formal specification. In addition, constraints for the "correct-by-construction" composition of these parts are also captured formally. Together these are combined to form an *executable specification* whereby the space of valid designs can be explored. All of this specification can be done with Eugene. Eugene is then passed to Clotho and its tools to create design files for liquid handling robots. Once the designs are physically created, an analysis of the designs is performed (using laboratory assays). The correct parts are fed back to the part repository for future use. Both functional and non-functional designs then contribute to the creation of constraints for future runs. As the rule set grows, the number of non-functional designs will continue to decrease while money and time saved increases.

A. Definitions

Before we delve into the details of the genetic device, we provide some definitions for the reader. Due to space constraints we cannot cover all the required biological background. However a key idea in this work is abstraction and treating the following as primitive building blocks by which our genetic device can be created, should be sufficient.

978-1-4244-5309-2/10/$26.00 ©2010 IEEE

Definition 1: Phage - A virus that infects bacteria.
Definition 2: Plasmid - A circular piece of DNA that propagates itself in bacteria.
Definition 3: Phagemid - A plasmid that contains a sufficient complement of phage elements to get itself packaged into viral particles.
Definition 4: Lysogen - Viral DNA that passively propagates inside bacteria.
Definition 5: Lytic Mode - Alternative to lysogeny, where phage generate more viral particles and destroy the host bacteria.
Definition 6: Promoter - A DNA element that causes transcription (DNA to mRNA).
Definition 7: Anti-Repressor - A protein that neutralizes a regulatory protein normally expressed by the phage lysogen, causing the phage to enter the lytic mode.
Definition 8: Inducible Promoter - A promoter that operates only after addition of a specific small molecule.
Definition 9: Lytic Replicon - A protein that recognizes its own sequence and generates linear double stranded DNA molecules that can be packaged into viral particles.
Definition 10: Packaging Site - A protein the recognizes its own sequence, required to package DNA into viral particles.
Definition 11: Regulated Promoter - A promoter that only operates after the phage has entered the lytic mode.
Definition 12: Terminator - DNA element that terminates transcription.

II. Design Overview

Phages offer efficient delivery of large pieces of DNA to bacteria, and can amplify themselves an order of magnitude faster than their hosts. Consequently, subversion of phage function enables the development of improved tools for molecular biology, which improves the ability of scientists to pursue projects such as the development of a tumor destroying bacteria. Here, the goal was to construct a genetic device that both controls entry of a phage lysogen into lytic mode and causes itself to be packaged into phage particles. This is known to minimally require a packaging site, a lytic replicon, and an anti-repressor protein. What was not known was how to organize and control the expression and timing of the component parts. Figure 2 provides an annotated overview of the Phagemid system.

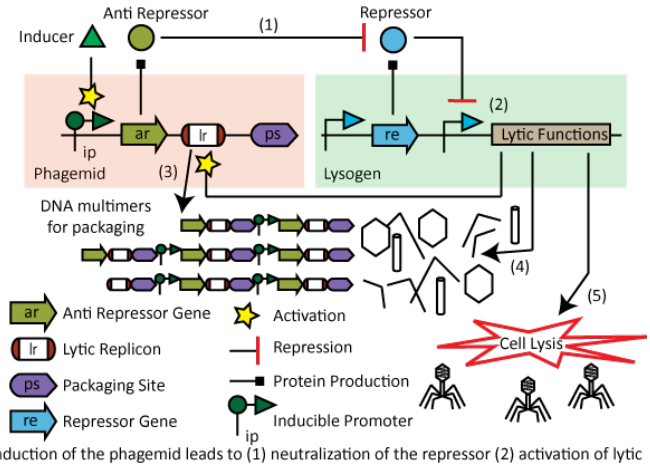

Fig. 2. Phagemid Design Overview

Specification 1 shows how using Eugene we *defined* the primitive properties of interest as well as which collections of properties made up the individual parts. Properties as shown can be text values or numbers along with a name assignment. Parts are assigned names and properties when defined. In our design we had 8 properties and 6 parts.

Specification 1 Property and Part Definitions

```
1: Property sequence(txt); //DNA Sequence
2: Property offStateStr(txt); //Amt. of transcription when off
3: Property onStateStr(txt); //Amt. of transcription when on
4: Property forwardEff(num); //Fwd. termination eff.
5: Property reverseEff(num); //Rev. termination eff.
6: Property toxicity(txt); //Toxicity level
7: Property activity(txt); //Relative effectiveness of protein
8: Property copyNum(num); //DNAs generated or maintained
9: Part AntiRepressor(sequence, toxicity, activity);
10: Part InduciblePromoter(sequence, offStateStr, onStateStr);
11: Part LyticReplicon(sequence, copyNum, toxicity);
12: Part PackagingSite(sequence, toxicity);
13: Part RegPromoter(sequence, offStateStr, onStateStr);
14: Part Terminator(sequence, forwardEff, reverseEff);
```

After the parts and properties are defined, they are *instantiated*. This is the process of assigning actual values to the part properties. This can be done manually or retrieved from a part repository [5] and converted with Clotho into Eugene. Specification 2 illustrates a few sample instantiations. Not shown here are ar2, ar3, rp2, rp3, t2, or t3. These are different instantiations of anti-repressors, regulated promoters, and terminators respectively.

Specification 2 Sample Part Instantiation

```
1: AntiRepressor ar1("CTA...", "medium", "high");
2: InduciblePromoter ip("CAA...", "very low", "high");
3: LyticReplicon lr("TAC...", 100, "high");
4: PackagingSite ps("CCG...", "none");
5: RegPromoter rp1("GCC...", "medium", "unknown");
6: Terminator t1("TGA...", 99, 99);
```

After the parts were instantiated they were assembled into devices. The following sections introduce the seven devices we investigated. They are classified by how they actually performed in the laboratory environment. All devices are shown in Figure 3.

A. Failed to Assemble

The first category of devices consists of four devices that failed to assemble. Two of these devices, d1 and d2, both contained the same regulated promoter controlling the lytic replicon. The physical DNA for both of these devices could not be obtained in the lab because the regulated promoter proved to be always active instead of regulated as expected, and the lytic replicon proved to be toxic when always activated. The other two devices, d3 and d4, contained the same anti-repressor (but one different from that used in d1 and d2). Both of these devices were never constructed in the lab because that anti-repressor proved to be highly toxic and significantly impaired growth of bacterial cells regardless of the regulation used.

Specification 3 Devices That Failed To Assemble

```
1: Device d1(ip, ar1, ps, t1, rp1, lr, t2);
2: Device d2(rp1, lr, t2, rp2, ps, t1, ip, ar1, t3);
3: Device d3(rp3, ps, t1, ip, ar3, lr, t2);
4: Device d4(ip, ar3, t3, rp2, ps, lr, t2);
```

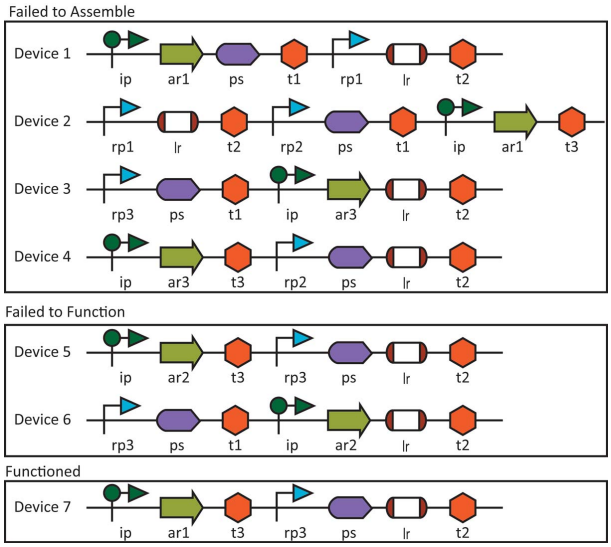

Fig. 3. Phagemid Device Overview and Classification

B. Failed to Function

The second category of devices consists of two devices that failed to function. d5 and d6 were both successfully constructed; however, they were unable to cause the phage to transition from lysogenic mode to lytic mode. As these devices both employ the same anti-repressor, it can be surmised that that anti-repressor does not have sufficient activity to work properly in these devices.

Specification 4 Devices That Failed To Function

```
Device d5(ip, ar2, t3, rp3, ps, lr, t2);
Device d6(rp3, ps, t1, ip, ar2, lr, t2);
```

C. Functioned

The final category of devices contains one device that functioned as desired. d7 caused lysogens to transition into lytic mode, and showed evidence of packaging into viral particles. Therefore, d7 must generate the necessary components at the appropriate time, without otherwise harming the cell.

Specification 5 Functioning Device

```
Device d7(ip, ar1, t3, rp3, ps, lr, t2);
```

III. Rule and Constraint Specification

Once the status of the devices had been determined, constraints were created to prevent undesirable device construction. There are five potential modes of failure that would prompt the creation of rule sets.

1) Devices must contain a sufficient set of biological functions to accomplish a particular goal. The absence of one or more of these functions leads to partial or complete abrogation of the desired functionality, so rules can be created to ensure that all of the necessary types of parts are present.
2) A particular part may prove to not function as expected, or at all, so rules can be created to prevent their incorporation into devices.
3) A particular part may prove to be toxic when expressed at too high a level or at the wrong time.
4) Parts may be mis-regulated, being expressed at either the wrong time or in the wrong quantity, preventing overall device effectiveness. Rules can be created to ensure that toxicity is avoided and function optimized by properly regulating expression.
5) Parts may interfere with each other, and rules can be generated to ensure that they appear only in allowable combinations.

In Eugene constraints are specified as *rules*. Rule operands can be part instances, numbers, or text values. Rule operators include compositional requirements (BEFORE, AFTER, WITH, NEXTTO, NOTCONTAINS, NOTMORETHAN) and comparison (<, ==, etc). Rules themselves are enforced with Note statements which can be created with boolean operations (AND, OR, NOT). In the event that the Note evaluates to "false" a the offending device is flagged.

In Specification 6 there are five constraint types illustrated. The first three are constraints that could be specified *a-priori*. The last two are the results of *empirical* evidence. The first set requires that there be no more than one of the same terminator in a device. The second set requires that a inducible promoter be immediately before an anti-repressor. Set 3 requires that an inducible promoter, lytic replicon, packaging site, and one type of anti-repressor be present in a device. Set 4 specifies that a specific regulated promoter not be with the lytic replicon. Finally, set 5 prevents the second and third anti-repressor from being used at all.

Specification 6 Constraint Based Rules

```
//Constraint Set 1 - a priori
Rule r1a(t1 NOTMORETHAN 1);
Rule r1b(t2 NOTMORETHAN 1);
Rule r1c(t3 NOTMORETHAN 1);
Note (r1a AND r1b AND r1c);

//Constraint Set 2 - a priori
Rule r2a(ip BEFORE ar1);
Rule r2b(ip BEFORE ar2);
Rule r2c(ip BEFORE ar3);
Rule r2d(ip NEXTTO ar1);
Rule r2e(ip NEXTTO ar2);
Rule r2f(ip NEXTTO ar3);
Note((r2a AND r2d) OR (r2b AND r2e)
  OR (r2c AND r2f));

//Constraint Set 3 - a priori
Rule r3a (ip WITH lr);
Rule r3b (lr WITH ps);
Rule r3c (ps WITH ar1);
Rule r3d (ps WITH ar2);
Rule r3e (ps WITH ar3);
Note(r3a AND r3b AND (r3c OR r3d OR r3e));

//Constraint Set 4 - empirical
Rule r4a(rp1 NOTWITH lr);
Note(r4a);

//Constraint Set 5 - empirical
Rule r5a(NOTCONTAINS ar2);
Rule r5b(NOTCONTAINS ar3);
Note(r5a AND r5b);
```

Specification 7 illustrates constraints on text values of properties. In particular, in this case the activity of anti-repressors must be "medium", "high", or "very high" for devices to be considered valid.

Specification 7 Constraint Based Rules Cont.

```
Rule r6a(ar1.activity == "medium");
Rule r6b(ar2.activity == "medium");
Rule r6c(ar3.activity == "medium");
Rule r6d(ar1.activity == "high");
Rule r6e(ar2.activity == "high");
Rule r6f(ar3.activity == "high");
Rule r6g(ar1.activity == "very high");
Rule r6h(ar2.activity == "very high");
Rule r6i(ar3.activity == "very high");
Rule ar1Con(NOTCONTAINS ar1);
Rule ar2Con(NOTCONTAINS ar2);
Rule ar3Con(NOTCONTAINS ar3);
Note((r6a OR r6d OR r6g) OR ar1Con);
Note((r6b OR r6e OR r6h) OR ar2Con);
Note((r6c OR r6f OR r6i) OR ar3Con);
```

IV. Performance

To examine how effective our constraints can be, we examined how they would have pruned the design space of 45 designs under consideration originally. Figure 4 illustrates that simply with the addition of rule r4a, the 45 designs are reduced to 20. The application of two more rules then reduces the final designs to only four candidates. This is a 91% reduction in overall design count. Examining the final designs experimentally revealed that these devices do indeed function (three functioned moderately well, while one functioned very well).

Constraints also provide a cost and time savings as well. Both the cost and design time associated with building the devices depends on the number of junctions between parts. The median number of junctions for devices in this work is 6. Each junction costs approximately $3 to make (due primarily to DNA purification columns and restriction enzymes), and takes about 10 minutes of design time (20 junctions amortized across 200 minutes). Thus, the time and costs drop from $810 and 2700 minutes to $72 and 240 minutes by going from 45 constructs to 4. This is also illustrated on Figure 4.

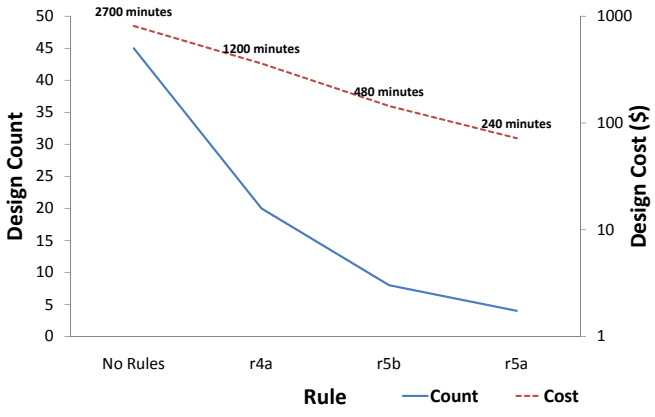

Fig. 4. Design Space and Cost Reduction Via Constraint Application

Finally Algorithm 8 illustrates how the *permute* function in Eugene operates by replacing part instances in a device with other available,

Algorithm 8 Device Permutation

```
//Generate permutations of device D_N for assembly
permute(D_N)
for all P_i ∈ D_N do
  for all p_n ∈ P_i do
    p_n → p_{n+1}
    Store(D_{N+1})
    CreateAssembly(D_{N+1})
  end for
end for
```

defined part instances in the same part family. The run time of this algorithm in O(|D|*|P|) where |D| is the number of parts in a device (the size of the device) and |P| is the number of individual instances of the part. In our design the max value of |D| was 9 and the max value of |P| was 3 so the runtime was negligible. Even for very large values we expect fast computational runtimes which will be insignificant compared to the physical assembly time.

V. Conclusion

We have briefly illustrated how lessons learned from a small set of devices created experimentally can be captured formally as constraints in the Eugene language. These constraints when applied to larger design sets have the potential to not only greatly prune the overall design space but most importantly lead to functioning designs. This reduction in design count saves a substantial amount of time and money for the designer. The construction of rules is flexible and modular which allows them to be reused and the computational runtime of the software will be insignificant.

Future work includes expanding the designs examined and looking for ways to automatically generate rules by analyzing the results of assembly to detect patterns and other anomalies in the designs. Another potentially useful mode for such a tool is in pinpointing which properties should be probed to most reduce the set of potential devices by considering which rules can be applied if experimental data for properties is obtained. For example, if the strength of all promoters is known and there are many rules governing expression levels, many orders of magnitude fewer parts may need to be considered if the designer is able to apply these rules. It then becomes an optimization problem as to which properties an experimentalist should measure. Ultimately this would involve a tradeoff between effort in property determination and desired design space reduction.

Acknowledgment

The authors would like to thank Joanna Chen, Richard Mar, Thien Nguyen, Nina Revko, and Bing Xia for helping to develop tools related to Eugene's development. In addition, discussions with Cesar Rodriguez and Emma Weeding were very useful in our initial discussions of Eugene.

References

[1] D. G. Chinnery and K. Keutzer, "Closing the gap between asic and custom: an asic perspective," in *DAC '00: Proceedings of the 37th Annual Design Automation Conference*. New York, NY, USA: ACM, 2000, pp. 637–642.

[2] Semiconductor Industry Association, "Industry fact sheet," http://www.sia-online.org/cs/industry_resources/industry_fact_sheet, 2009.

[3] Berkeley Software iGEM Team, "Eugene domain specific language overview," http://2009.igem.org/Team:Berkeley_Software/Eugene, 2009.

[4] D. Densmore, A. Van Devender, M. Johnson, and N. Sritanyaratana, "A platform-based design environment for synthetic biological systems," in *TAPIA '09: The Fifth Richard Tapia Celebration of Diversity in Computing Conference*. New York, NY, USA: ACM, 2009, pp. 24–29.

[5] MIT, "Registry of standard biological parts," http://partsregistry.org, 2009.