Chapter 1

HIGH-LEVEL PROGRAMMING LANGUAGES FOR BIO-MOLECULAR SYSTEMS

Jacob Beal BBN Technologies Cambridge, Massachusetts jakebeal@bbn.com

Andrew Phillips Microsoft Research Cambridge, England andrew.phillips@microsoft.com

Douglas Densmore Boston University Boston, Massachusetts dougd@bu.edu

Yizhi Cai Johns Hopkins University Baltimore, Maryland cai@jhmi.edu

Abstract In electronic computing, high-level languages hide much of the details, allowing non-experts and sometimes even children to program and create systems. High level languages for bio-molecular systems aim to achieve a similar level of abstraction, so that a system might be designed on the basis of the behaviors that are desired, rather than the particulars of the genetic code that will be used to implement these behaviors. The drawback to this sort of high-level approach is that it generally means giving up control over some aspects of the system and having decreased efficiency relative to hand-tuned designs. Different

languages make different tradeoffs in which aspects of design they emphasize and which they automate, so we expect that for biology, there will be no single "right language," just as there is not for electronic computing. Because synthetic biology is a new area, no mature languages have yet emerged. In this chapter, we present an in-depth survey of four representative languages currently in development—GenoCAD, Eugene, GEC, and Proto—as well as a brief overview of other related high-level design tools.

Keywords: Synthetic biology, abstraction, high level languages, GenoCAD, GEC, Proto, Eugene, modeling, design, XOR

1. Overview

A "high-level" programming language is one that abstracts many of the details of how a computation will actually be implemented. The programmer writes down a simple description, capturing the essence of the computation, and this description is automatically expanded to produce a complete implementation that can be executed on the available computational substrate.

On modern digital computers, this process can go through many different layers. Consider, for example, an entry in a Microsoft Excel spreadsheet that adds up a column of figures. The expression itself, something like "= SUM(C1:C10)", is a transparently simple statement in an arithmetic-centric high-level language. Within Excel, this statement is interpreted into a set of calls to various functions within Excel, in the process adding implicit behaviors like error handling. These Excel functions were themselves written in some high-level language, and then compiled into machine code that can execute on the computer where Excel is running, in the process making routine decisions like how to implement each mathematical operation using the resources of the machine's processor. Even that machine code goes through another layer of interpretation, as the processor itself restructures the code to operate more efficiently given the current state of the processor.

The essence of the idea behind high-level languages is this: as an engineering field matures, finding good-enough solutions to sub-problems of design becomes routine. Highly routine problem solving can then be automated, reifying the knowledge of skilled engineers into a piece of software. The software solutions to individual parts of the design process can then be connected together to form a complete tool-chain, translating from high-level descriptions down to working implementations without any need for human intervention. Separating the programmer from the implementation details has three important benefits:

- Accessibility: less knowledge is required to build a system, since much of the required knowledge has been captured in software.
- Scalability: since routine design work is automated, it is possible to build larger and more complex systems, and to re-use the same programs on different platforms.
- **Reliability:** aspects of design that are automated are no longer subject to programmer error; software can also check for common errors in the programmer's high-level design.

On electronic computers, high-level languages have become so successful that few people ever use anything besides a high-level language. In the programming of bio-molecular systems, high-level languages are just beginning to emerge.

For the purposes of this chapter, we will define a high-level language for bio-molecular systems as any system description language where the choice of implementing biological parts may routinely be left unspecified. We will focus primarily on programming languages for *in vivo* biomolecular computation, reviewing four representative languages: Geno-CAD, Eugene, GEC, and Proto in rough order from lower to higher levels of abstraction. To aid comparison and understanding, we apply each language to a simple example problem:

Express green fluorescent protein (GFP) when either of the small-molecule signals aTc or IPTG is present, but not when both are present.

At the end of the chapter, we also review the scope of other related high-level design tools for bio-molecular computing systems.

2. GenoCAD

GenoCAD (www.genocad.org) is one of the earliest CAD tools for synthetic biology, built upon the foundation of formal grammars. In this section, we summarize the basics of grammars, the theoretical foundation underneath GenoCAD and also a brief tutorial on how programs are constructed using the GenoCAD web service.

Formal Language & Syntactic Model

A formal language is a set of (possibly infinite) strings derived from an alphabet Σ , which encodes information for communication purposes. There are several kinds of languages, including natural languages (*e.g.*, English and Chinese), computer languages (*e.g.*, C and HTML), and mathematical languages (*e.g.*, first-order logic). However not all the strings over a language's alphabet actually belong to that language, only those which follow its rules. A grammar is a finite set of rules that specifies the syntax (permissible structure) of a language. A grammar G contains four components:

- A finite set N of non-terminal symbols.
- A finite set Σ of terminal symbols that is disjoint from N.
- A finite set P of rewriting rules, each rule is in the form of $\alpha \to \beta$, where α and β are both strings of symbols, and α contains at least one symbol from N. More formally, a rewriting rule can be represented as $(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$, where * is the Kleene star operation (meaning zero or more copies of the preceding statement) and \cup is the set union operation.
- A distinguished symbol $S \in N$ that is the start symbol.

In the 1950s, Chomsky classified grammatical models into four classes based on the forms of their production rules, which reflect their expressive power [Chomsky, 1956]. In a nutshell, selecting a class of grammatical model as the representation of biological sequences is a tradeoff between the expressivity and the compilation complexity. Since GenoCAD uses a Context-Free Grammar (CFG), we will only give the mathematical definition of CFG. A good general introduction to formal languages and the Chomsky hierarchy may be found in [Sudkamp, 2006].

A Context-Free Grammar allows any production rule of the form $A \to \alpha$. The left-hand side only consists of a single non-terminal symbol A, and the right hand side can be any string α , where $A \in N$, and $\alpha \in (N \cup \Sigma)^*$. The corresponding automaton for a context free grammar is a push-down automaton. The computational complexity to recognize a context free grammar is polynomial.

GenoCAD formalizes many generic design principles of molecule biology in the form of a context free grammar. The biological parts are the terminals, while the devices/systems composing multiple parts are categorized as non-terminals in the grammar. In this review, only a small grammar will be presented: two more comprehensive grammars are published elsewhere [Cai et al., 2007, Cai et al., 2010].

Table 1.1 summarizes the non-terminals and terminals used in this small grammar. S is a special non-terminal which is used as the start symbol of the grammar. *Operon* and *Cistron* are complex devices, which

Table 1.1. GenoCAD small grammar set of terminal and non-terminal symbols.

Non-terminals	Terminals
S	-
Operon	-
Cistron	-
Promoter	prom 1, prom 2, prom 3
RBS	rbs1, rbs2, rbs3
Gene	lacI, tetR, gfp
Terminator	b0012, b0015

Table 1.2. GenoCAD small context free grammar of gene expression.

Number	Rule
<i>P</i> 1	$S \rightarrow Operon$
P2	$Operon \rightarrow Operon, Operon$
P3	$Operon \rightarrow Promoter, Cistron, Terminator$
P4	$Cistron \rightarrow RBS, Gene$
P5	$Cistron \rightarrow Cistron, Cistron$
P6	$Terminator \rightarrow Terminator, Terminator$
$P7 \cdots P9$	$Promoter \rightarrow prom1 prom2 prom3$
$P10 \cdots P12$	$RBS \rightarrow rbs1 rbs2 rbs3$
$P13 \cdots P15$	$Gene \rightarrow lacI tetR gfp$
P16, P17	$Terminator \rightarrow b0012 b0015$

are composed of multiple basic parts (terminals). In the category of *Promoter*, there are three terminals, namely *prom1*, *prom2* and *prom3*. Similarly, a ribosome binding site *RBS* can be chosen from *rbs1*, *rbs2* and *rbs3*, while a *Gene* could be *lac1* or *tetR* or *gfp*. Finally, there are two terminals *b*0012 and *b*0015 belong to the non-terminal *Terminator*. Table 1.2 presents a context free grammar for designing gene expression cassettes. The whole grammar can be divided into two sections: rules P1 - P6 transform the structure of a design, while rules P7 - P17 are used to select a particular terminal for each non-terminal category. The design starts with P1, where the start symbol S becomes an expression *Operon*. Multiple *Operons* are allowed by applying rule P2 multiple times: for a design with n cassettes, P2 is applies n - 1 times. Rule P3 specifies the structure of an *Operon* to be a *Promoter*, followed by a *Cistron* and a *Terminator*. A *Cistron* can be broken down by

rule P4 as an RBS and a Gene. Multiple Cistrons and Terminators are allowed in a design by rules P5 and P6, respectively. After the structure of a design is defined, rules P7 - P17 are used to transform each non-terminal to a specific biological part (terminal). For instance, rules P7, P8 and P9 specify prom1, prom2 and prom3 respectively to replace non-terminal Promoter (the '|' sign indicates OR relationship).



Figure 1.1. Grammatical design of a DNA sequence. Panel 1: a parsing tree showing the step-by-step application of rules to generate the sequence (excepting terminal selection). Each step is labeled with the rule applied. Panel 2: Representation of the generated DNA part sequence, using a standard set of synthetic biology icons. Panel 3: The designed DNA sequence.

Figure 1.1 shows how this simple syntactic model can be applied to generate a sequence structurally consistent with the XOR gates developed below in our presentation of Eugene (Figure 1.3) and GEC (Figure 1.4). The design process starts with applying P1 to the start symbol S to transform the design into a single *Operon*. After applying P3 twice, the design becomes three *Operons*. In the next step, rule P3 defines the structure of each *Operon* as a *Promoter*, a *Cistron* and a *Terminator*. In order to express *lacI* and *tetR* under control of the same constitutive promoter, P5 is applied to allow two *Cistrons* in the leftmost *Operon*. Finally, rule P4 breaks down each *Cistron* into an *RBS* followed by a

Gene. Once the structure of the design is decided, a part is selected for each category (Figure 1.1.2) and mapped to a DNA sequence that can be exported for synthesis (Figure 1.1.3).

If we operate the process in Figure 1.1 in reverse, then rather than generating a DNA sequence, we can validate whether a specified DNA sequence is consistent with the syntactic model. This is carried out with an automated process known as "parsing" in computer science. The parser operates in the reverse order of the design process: the Geno-CAD parser takes the DNA sequence (panel 3 in Figure 1.1) as input and breaks it into a series of biological parts (panel 2 in Figure 1.1). It then checks for the existence of at least one rule application tree that can generate this series of parts using the context-free grammar. Realizing that we can build parsers from the syntactic model opens up the possibility of viewing DNA sequences as a programming language. One can make changes to a DNA sequence just like writing source code, and use the parser to check whether the new DNA sequence is still consistent with the syntactic model (which formalizes the biological knowledge).

It should be noted that the syntactic model only checks the structure, but not the meaning of the design. A syntactically correct sentence is not necessarily meaningful. In the context of synthetic biology, this means the syntactic model only controls the order of putting biological parts together to ensure a successful gene expression, but the function of the DNA sequence (*i.e.*, what does this sequence do?) remains unknown. Recently, GenoCAD has been extended to address this area with the introduction of an attribute grammar to develop semantic models of DNA sequences [Cai et al., 2009]. By associating biological attributes with parts, and coupling semantic actions with each production rule, the semantic models are capable of translating a class of DNA sequences to mathematical models that describe the encoded phenotypic behavior.

GenoCAD Web Service

Based on the syntactic model originally described in [Cai et al., 2007], an open-source web application (www.genocad.org) has been implemented. GenoCAD constrains the design space using the underlying syntactic models, and guides the user through the design process in a "point and click" fashion. This has been extended recently with a second syntactic model, designed specifically for BioBrickTM-based constructs [Cai et al., 2010].

The GenoCAD web tool applies these syntactic models to support both design and validation of sequences [Czar et al., 2009] (though at the moment when this chapter was written, the validation section was offline

GenoCAL		•so sign Validate Parts About Log In	Geno CAD		Design Valida	te Parts	About	- none Log in
History Record	Pice Bisson City City Pice RBS City Total Total Pice RBS RBS City Total Pice RBS RBS Total Total	Grammar/Parts Selectio	Validate a Sequence: Exercise of white sequences points in the sequences points and the sequences points and the sequences points and the sequences of white Amendmentary such as sequences of white the sequences of the sequences of the sequences of white the sequences of points are sequences to point the sequences operative along the sequences to point the sequences operative along the sequences of the sequences operative along the sequences of the sequences operative along the sequences to point the sequences operative along the sequences of the sequences operative along the sequences of the sequences of the sequences operative along the sequences of the sequences operative along the sequences operative along the sequences operative along the sequences operative along the sequences operative along the sequences operative along the sequences operative along the sequences operative along the sequences operative along the sequences o	E stal Expression Granmar Path Arts Library E cuts ant St Stappers Access topper access topp	B pression prammer B mean to propose to provide the than the propose to provide the the propose to provide the top provide the top top top top top the comparison of the top top the comparison of the top top the comparison of the top top the top	angst.oct.gopce.ca etropita.egs torou- catogit.cct.co.go ggit.locgit.asgetori tatata.asgego ingoogaaaatorig ingoogaaaatorig gatatorogosco.as gatatorogosco.as gatatorogosco.as gatatorogosco.as	consecution equipation acogrossopace masgeoscore asgeoscore to to a the to a test to a	tgesttooge thtgatgeta tttticatt aggtagoca voittiggtit eggegggata eggtagggata
C GenoC	A D ==	come Veldata Parts Adord Ling Out	f a seguent a suid, for evidence up a sense of hyperech the seguence to a sense of more than the the sense of the sense of more than the the sense of the sense of sense f. Fire sense the transfer for the error message a clustered.	Constant of the sequence	te is valid	G ti O Parts Abo	Upbood • • •	a sequence (*
My Designs My Lit	rafies My Parts Update Profile		My Dealgne My Libraries	Ny Parts Ny Profile				
Which grammar will th Concept grammar (Cal	is library use? . 2007) 4	Describe Your Library: My library	My Designs	Description	Last Wodffeeting	Clone	Delete Vewild	odity
Description, Grammar descr	Sed in Biantomatics 2007		Type 2 Tempiate Typ	e 2 tempiate for toggie switch	9/17/2008 4:44:45 PM	Cone this design	Delete View	_
Preload a library below Base library concept g	anmar 😐	Your Library Name: MyFirstLibrary	Type 1 Tempiate Typ	e 1 template for toggle switch	9/17/2008 4:48:16 PM	Cone this design	Delete View	
Available parts: a01 (00500)	Add Scienced as	w22 (8004d)	p8A0102 Typ	•1	9/17/2008 4:50:31 PM	Cone this design	Delete View	
a10 (Pitet0-1) b01 (Pite-2-ro) b02 (Pite-1)		#03 (R0010) #04 (13453)	19040-03 Tyl		9172008 4 51 08 PM	Core his design	Datete View	
602 (PT-\$1005-FC) b03 (PtetD-1-rc) b04 (PDE-\$1076-rc)	Add All a	#25 (25/034) #66 (80011) #67 (219050)			9172008 4 42 18 PM	Cone his desire	Dateta Vitar	
c02 (80034) c02 (80034)		a08 (Prc-2) a09 (P-10an)	p88(112) 1/2	•1	9172008 4 53 07 PM	Cone this desires	Delete View	
(03 (80032) (04 (A)	<remove selecte<="" th=""><th>d5 (8) 056 (0)</th><th>p887124 7/</th><th>•1</th><th>9172008 4 53 45 PM</th><th>Cone this design</th><th>Delete Vew</th><th></th></remove>	d5 (8) 056 (0)	p887124 7/	•1	9172008 4 53 45 PM	Cone this design	Delete Vew	
c08 (0) c09 (7)	- Remove All	e30 (apene)	p8RT125 Te	•1	9172008 4.54 18 PM	Cone this design	Delete View	
c10 (C) c11 00			Type 3 Tempiate Typ	a 3 template for toggle switch	9172008 4:58:09 PM	Cone this design	Delete View	
d01 (Anc)	2		pTAK106 Tgs	13	9172008 5:00:27 PM	Cone this design	Delete View	
		Save Library Clear	pTAK107 7g	• 3	9172008 5:01:14 PM	Cone this design	Delete View	
			pHE:108 7ys	• 3	\$172008 5:02:05 PM	Cone this design	Delete View	
			pKE110 7g	• 3	\$172008 5:02:41 PM	Cone this design	Delete View	
			pTAK102 w Linker Te	2	919200842108PM	Cone this design	Delete Vew	

Figure 1.2. Screenshots of the GenoCAD.org web service showing pages for sequence design (A), sequence validation (B), part library customization (C), and the user's workspace (D).

for development). The design space (Figure 1.2.A) has two distinct sections: on the left hand side is the "History Record" which keeps track of each design step, while the right hand side shows the current design. On top of the right hand side is an icon representation of the current design, which will evolve as the design proceeds. Underneath the icon representation is the main design space, where a user can point and click on a grammar rule to transform the design or to decide on a specific biological part for a category. After the design is finished (*i.e.*, the structure is finalized, and all parts are selected), GenoCAD will offer the user an option to export the DNA sequence being designed.

If a DNA sequence is designed outside GenoCAD, it can be taken into the validation section (Figure 1.2.B) to check whether the composition of biological parts is consistent with GenoCAD grammars. It should be noted that if a sequence fails in the validation, it does not necessarily mean this sequence is non-functional. Rather, it means that the Geno-CAD grammar could not find a parsing tree to generate this sequence, and that sequence requires a closer inspection by human experts. On entering the validation page, a user firsts select a grammar to validate against, then pastes the DNA sequence into GenoCAD. The tool will then interpret the DNA sequence into a series of parts and (if successful) report whether this sequence has a correct structure as defined by the selected grammar.

Finally, users who elect to register an account with the GenoCAD web tool have more privileges in customizing their design space in GenoCAD. A registered user can create new libraries (Figure 1.2.D), add new parts, and save intermediate and final designs for later use (Figure 1.2.D).

3. Eugene

Eugene (www.eugenecad.org) is a human readable, executable specification, which reflects the creation of biological systems by defining, specifying, and combining collections of biological parts. Eugene is inspired by the languages of the Electronic Design Automation (EDA) industry (e.g. Verilog, VHDL) in terms of its ability to provide a biological design netlist (a collection of abstract components and their connections) which can be synthesized (automatically transformed) into collections of physical implementations in a design library.

Eugene bridges the synthetic biology "part" and "device" (composite of multiple basic parts) hierarchy levels by explicitly addressing the components in different levels of the hierarchy. These relationships are explicitly reflected in Eugene's data types: **Device** and **Part** declarations abstract low-level implementation details (captured by **Property** statements), while still providing the capability to capture the lower level information through the encapsulation of specific design information with **Part Instance** objects composed of specific **Properties**. These features address the need for flexibility in biological part and device specification. Moreover, Eugene can directly interface with design tools like Clotho [Densmore et al., 2009] which extract information from repositories of biological parts and encapsulate that information as Eugene "header files". These files define specific instances of Parts and their Properties for a given "design library". These header files are modular and allow changes from one design library to the other with the inclusion of different files without modifying the Eugene Device declarations.

Eugene is also an executable specification since it is an interpreted language. At runtime, the Eugene interpreter can create collections of **Devices** based on conditional execution statements (e.g. **if**) coupled with specific functions to create new **Devices** at runtime. These features address the need for the combinatorial exploration of devices from a wide variety of different biological parts. For example, if a particular Part's Property does not meet a specific threshold, the body of the conditional statement can be used to swap that Part out with one that does meet the requirements.

Finally, Eugene allows for the creation and assertion of design rules. A **Rule** directly applies to the relationship between various **Parts** in a **Device** and provides the validation mechanisms needed to ensure the successful creation of a construct. These rules are not predefined in the language but rather created by the user from a rich set of rule primitives. Such flexibility allows users to define and assert numerous combinations of rules.

Eugene Constructs

The language supports five predefined primitive types. These are **txt**, **num**, **boolean**, **txt**[] (a list of text sequences), and **num**[] (a list of numbers). **Properties** represent characteristics of interest and are defined by primitives and associated with parts. The data type **Part** represents a standard biological part, such as a BioBrickTMin the MIT Registry. **Part** definitions do not construct any parts, but rather specify which parts can be constructed. Declarations of those parts create instances of predefined **Parts** and assigns values to their properties. **Device** statements represent an ordered composite of standard biological parts and/or other devices. Below are examples of these constructs:

```
//Eugene Primitives
txt[] listOfSequences = ["ATG", "TCG", "ATCG"];
txt specificSequence = listOfSequences[2];
num[] listOfNumbers = [2.5, 10, 3.4, 6];
num ten = listOfNumbers[1];
//Eugene Properties
Property Sequence(txt);
Property RelativeStrength(num);
//Eugene Part Declarations
Part Promoter(ID, Sequence, Orientation);
Part ORF(ID, Sequence, Orientation);
Part RBS(ID, Sequence, Orientation);
//Eugene Part Instances
RBS rbs (.Sequence("gatcttaattgcggagacttt"), .Orientation("Forward"));
ORF orf (.Sequence("gatcttaattgcggagacttt"), .Orientation("Forward"));
//Eugene Device
Device BBa_K112234(rbs, orf);
```

Eugene Rules

The specification of rules provides the ability to validate **Device** declarations. **Rule** declarations themselves do not perform the validation. They have to be "noted", "asserted" or used as expressions inside an if-statement to affect program operation. **Rule** declarations are single statements consisting of a left and right operand and one rule operator. The rule operators **BEFORE**, **AFTER**, **WITH**, **NOTWITH**, **NEXTTO**, **NOTCONTAINS**, **CONTAINS**, and **NOTMORETHAN** can be applied to **Part** instances or **Device** instances. These operators also have been defined with specific semantics as well (e.g. their commutative properties). Property values of **Part/Device** instances or primitives in relation with one **Part/Device** can be operators in rule declarations when using the relational operators <, <=, >, >=, !=, ==. These operators are overloaded when evaluating text and the text is compared according to alphabetical precedence. The following are examples of rules in Eugene:

```
Rule r1(rbs BEFORE orf);
Rule r2(rbs WITH promoter);
Rule r3(promoter NEXTTO rbs);
Rule r4(rbs.Sequence != orf.Sequence);
Rule r5(rbsStrong.RelativeStrength > rbsWeak.RelativeStrength);
num relativeStr = rbsStrong.RelativeStrength;
Rule r6(p.RelativeStrength > relativeStr);
Assert(r6); // Strong enforcement of the rule (stop compilation)
Note(r4); // Weak enforcement of the rule (warning)
```

Currently rules must be defined explicitly in the body of the Eugene program or in a header file. However work is in progress to examine ways to associate rules with Parts types and instances as well as generate constraints in response to experimental work done in laboratories which is fed back to Eugene at runtime. In addition, the automated assembly system j5 [Nathan J. Hillson, 2010] uses Eugene constraints as part of its combinatorial exploration of alternate devices.

Eugene Functionality

The use of conditional statements breaks up the flow of execution and allows selected blocks of code to be executed. Eugene supports two kinds of **if** statements to achieve this: rule validating **if** statements and standard **if** statements. The three logical operators **AND**, **OR**, and **NOT** can combine statements of each type but cannot mix them together.

```
Rule r7(rbs BEFORE orf);
if(on (BBa_K112234) r7) {
  Block statement, in case of true evaluation
} else {
  Block statement, in case of false evaluation
}
boolean test = true;
if(test) {
  Assert(r7);
} else {
  Assert(NOT r7);
}
```

The **permute** function automates the specification of many **Devices** that share the same basic structure. Applying **permute** generates a **Device** for every combination of predefined **Parts**, maintaining the **Part** type of each component in the original **Device**. For example, the following code will result in eight devices at the completion of the permute operation.

```
Promoter p1(.Sequence("atc"));
Promoter p2(.Sequence("gcta"));
RBS rbs1(.Sequence("gatct"),.Orientation("Forward"));
RBS rbs2(.Sequence("gatcttaatt"), .Orientation("Forward"));
Device d2(p2, d1, rbs2);
permute(d2);
```

Permute also can be given additional parameters that limit the number of Devices created or force the Devices to adhere to the rules currently defined. The latter provides an intelligent design space exploration process. For example, Permute(d2, 4, strict) will create four Devices which adhere to the rules currently defined while maintaining the overall structure of Device d2.

XOR Design Example

We now show how Eugene can be applied to design our example XOR system, which only produces a green fluorescent protein (GFP) with either aTc or IPTG (but not both) present. Figure 1.3 shows the proposed network of parts and the regulation present. The code snippet below shows what this design would look like in Eugene. Of note is that since Eugene is based around the specification of devices from individual parts, there is not a natural way to express small molecular interactions. These manifest themselves as properties of the parts. Control statements

could then check these properties to create alternate networks reflecting the presence or absence of these molecules. If the proper DNA sequences are provided for all the parts, these interactions themselves would occur naturally in the physical device. The provided design merely captures the topology of the XOR device as an ordered collection of parts.



Figure 1.3. XOR Gate Design Example Designed With Eugene

```
Property sequence(txt);
Property smallMoleculeInteraction(txt);
Property type(num);
//1 - neg regulated by lacI, pos regulated by tetR
//2 - neg regulated by tetR, pos regulated by lacI
Part ConstitutivePromoter(sequence);
Part RegulatedPromoter(sequence, type);
Part ORF(sequence, smallMoleculeInteraction);
ConstitutivePromoter cp("ACGT...");
RegulatedPromoter rpType1("ACGT...", 1);
RegulatedPromoter rpType2("ACGT...", 2);
ORF gfp("ACGT...", "none");
ORF lacI("ACGT...", "aTc");
Device xor(cp, lacI, tetR, rpType1, gfp, rpType2, gfp);
```

Notice that here no rules are actually specified. However, were this design actually given to a downstream tool chain for automated assembly, one would want to create many potential devices in case the provided device either fails to function or assemble. Potential rules could be:

//This places the ConstitutivePromoter before lacI and tetR

```
Rule cpLocation1(cp BEFORE lacI);
Rule cpLocation2(cp BEFORE tetR);
Assert(cpLocation1 AND cpLocation2);
//Ensures that only two RegulatedPromoters are in the system
Rule UniquePromoter1(rpType1 NOTMORETHAN 1);
Rule UniquePromoter2(rpType2 NOTMORETHAN 1);
Assert (UniquePromoter1 AND UniquePromoter2);
```

For the sake of space, all the rules are not listed here but one should specify the relationship between the gfp ORF part and the Regulated-Promoters as well. This would be followed by a Permute(xor, strict)function call which would create a variety of devices (e.g. with the position of the lacI and tetR parts swapped). These devices would then be given to an automated assembly program [Densmore et al., 2010] for downstream use with laboratory automation.

4. GEC

This section describes a programming language for Genetic Engineering of Cells (GEC), initially presented in [Pedersen and Phillips, 2009a] and available at http://research.microsoft.com/gec. The main goal of GEC is to facilitate the design, analysis and implementation of biological devices inside living cells. GEC builds on previous research in the field of synthetic biology, including a registry of standard parts (http://partsregistry.org) together with experimental techniques for combining these parts into higher-level devices. More recently, a range of software tools have been developed for designing and simulating biological devices, as discussed for example in [Purnick and Weiss, 2009, Pedersen and Phillips, 2009a. The main innovation behind GEC is to take the design process a step further, by allowing biological devices to be designed with little or no knowledge of the specific parts available. The user needs only a basic knowledge of the available part types, namely promoters, ribosome bindings sites, protein coding regions and *terminators*. These elementary part types can be composed and the *properties* of the desired parts can be expressed as constraints in the GEC language. Once a biological device has been designed in this way, the GEC compiler automatically determines the set of actual parts that satisfy the design constraints. In most cases, multiple solutions are possible for a given design. GEC can compile each of the solutions to a set of chemical reactions, which can then be simulated or analyzed by the user. The solutions that exhibit the desired behavior can then be synthesized and put to work in living cells. Although there is no guarantee that a solution which produces the desired simulation results will



Figure 1.4. Designing an exclusive OR (XOR) logic gate in GEC. (a) GEC code for the XOR gate, together with its graphical representation, expressed in terms of part types, part properties and logical variables. Note that none of the part identifiers are specified explicitly. The design yields a number of possible solutions. (b) One of the solutions proposed by the GEC tool, expressed as a mapping from logical variables to molecules, together with a list of the part identifiers that make up the design.

function correctly inside a living cell, analyzing the design on a computer is an effective way to rapidly detect design errors prior to building the physical device—a process which can take several days and for which even small errors can prove very costly. We illustrate the design approach of the GEC language on a simple exclusive OR (XOR) logic gate (Figure 1.4 on page 15). The system is specified as a collection of three transcriptional units, where each unit consists of a sequences of part types. The first transcriptional unit consists of a promoter (prom), a ribosome binding site (rbs) a protein coding region (pcr), followed by another ribosome binding site and protein coding region, followed by a terminator (ter):

prom; rbs; pcr; rbs; pcr; ter

Additional constraints on part types are specified in the form of part properties. In the first transcriptional unit, the prom < con(RT) > denotes a promoter with a constitutive transcription rate RT, the part pcr < codes(PA) > denotes a protein coding region that codes for protein PA, and the part pcr < codes(PB) > denotes a protein coding region that codes for protein that codes for protein PB:

```
prom<con(RT)>; rbs; pcr<codes(PA)>; rbs; pcr<codes(PB)>; ter
```

The transcription rate RT and the proteins PA and PB start with an upper case letter, which means that they are *logical variables* representing an unknown rate and unknown proteins. Although the values of these variables are not known in advance, the GEC compiler takes into account the full set of design constraints in order to find suitable values that satisfy the desired properties. For example, the property RT > 0.1 states that the constitutive transcription rate of the promoter must be above a certain threshold. In the second transcriptional unit the part prom<neg(PA), pos(PB)> denotes a promoter region that is negatively regulated by protein PA and positively regulated by protein PB:

prom<neg(PA),pos(PB)>; rbs; pcr<codes(gfp)>; ter

This places additional constraints on the proteins PA and PB, which must act as a positive and negative regulator, respectively. The third transcriptional unit places further constraints on the proteins PA and PB, which must act as both positive and negative regulators simultaneously:

prom<neg(PB),pos(PA)>; rbs; pcr<codes(gfp)>; ter

Note that the protein gfp starts with a lower case letter, meaning that it represents a known protein.

In order to map logical variables and design constraints to physical parts, GEC includes a database of parts. Each of the parts in the database is associated with a part identifier together with zero or more part properties. A subset of a GEC parts database is shown in Table 1.3 on page 17. The part properties are also associated with rate constants, which are used to simulate the design solutions. For example,

Table 1.3. A subset of the GEC parts database, which can be defined and extended by the user. Each of the parts in the database is associated with a part identifier together with zero or more part properties.

ID	Type	Properties
e0040	pcr	codes(gfp, 0.01)
c0012	pcr	codes(lacI, 0.01)
c0040	pcr	codes(tet R, 0.01)
b0034	rbs	rate(0.1)
b0015	ter	
r0051	prom	neg(cl, 1.0, 0.5, 0.00005), con(0.12)
r0040	prom	neg(tet R, 1.0, 0.5, 0.00005), con(0.09)
rU1	prom	neg(tetR, 1.0, 0.01, 0.0), pos(lacI, 1.0, 0.5, 0.1), con(0.0)
rU2	prom	neg(lacI, 1.0, 0.01, 0.0), pos(tetR, 1.0, 0.5, 0.1), con(0.0)
rU3	prom	neg(tetR, 1.0, 0.5, 0.0), pos(lacI, 1.0, 0.5, 0.1), con(0.0)

Table 1.4. A subset of the GEC reactions database, which can be defined and extended by the user.

Reactants	rate	Products
lacI + iptg	1.0	lacI-iptg
tetR + aTc	1.0	tetR-aTc
$_{ m iptg}$	1.0	c[iptg]
aTc	1.0	c[aTc]

the database entry (c0040 \mapsto pcr,codes(tetR, 0.01)) denotes a protein coding region c0040, which codes for the protein tetR with degradation rate 0.01. The entry (r0051 \mapsto prom,neg(cI,1.0,0.5, 0.00005),con(0.12)) denotes a promoter r0051 that is negatively regulated by the protein cI, which binds to the promoter at rate 1.0 and unbinds at rate 0.5, where the repressed transcription rate is 0.00005 and the constitutive transcription rate is 0.12.

The design of the XOR gate in Figure 1.4 on page 15 also includes interactions between proteins and transport reactions across the cell membrane. The following constraints require that the protein A binds to PA and forms a complex PA-A, and that the protein B binds to PB and forms a complex PB-B. A vertical bar is used to separate multiple constraints:

$$PA + A \rightarrow PA-A \mid PB + B \rightarrow PB-B$$

This effectively specifies that the inputs A and B to the XOR gate can inhibit the activity of the transcription factors PA and PB by forming inert complexes with these transcription factors. Finally, the following properties require that both A and B are able to cross the cell wall:

 $A \rightarrow c[A] | B \rightarrow c[B]$

These properties are essential in order for the input signals of the XOR gate to be read by the cell. In order to map these reaction constraints to physical parts, the GEC system includes a database of reactions. Each of the reactions in the database consists of a set of reactants, a set of products and a corresponding reaction rate. A subset of a GEC reactions database is shown in Table 1.4 on page 17. For example, the reaction (lacI + iptg \rightarrow {1.0} lacI-iptg) denotes the formation of a complex between lacI and iptg. In many cases accurate rate information for these reactions is missing, and approximate rate constants are used instead.

The above design constraints for the XOR gate are solved by the GEC compiler in order to find an appropriate solution. For example, the first protein coding region of the first transcriptional unit must produce a protein PA that can both inhibit the promoter of the second transcriptional unit, activate the promoter of the third transcriptional unit and also form a complex with a compound that is capable of crossing the cell membrane. In the general case multiple solutions are possible for a given design. One of the possible solutions is shown in Figure 1.4 on page 15. The solution maps the inputs A and B to iptg and aTc respectively, and the transcription factors PA and PB to lacI and tetR, respectively. The corresponding part identifiers are also listed, which denote specific nucleotide sequences that could potentially be inserted inside a bacterium in order to program an XOR gate.

The main characteristic of the XOR gate is that green fluorescent protein (GFP) is only produced when one of the input signals A or B is present, but not both. When the user compiles the XOR gate design in GEC, they are presented with a set of possible solutions that satisfy the design constraints. The user can then simulate each of the solutions in order to choose the most desirable one. The design can be further refined by specifying that certain rates such as transcription, translation or transcription factor binding must lie within a specified range. This helps to reduce the initial set of possible solutions. In the case of the XOR design, one of the solutions represents a condition whereby GFP is produced even in absence of both inputs A and B. This occurs because the rate of repression of one of the promoters by transcription factor PA is less than its rate of activation by transcription factor PB, meaning that activation out-competes inhibition. This unwanted solution can be eliminated by adding the constraint that the inhibitor transcription factors bind more tightly than the activator transcription factors.

In order to simulate a given design, GEC automatically compiles the design to a set of chemical reactions, using the rates associated with the part properties and reactions in the GEC databases. The set of reactions for the XOR gate design is summarized in Figure 1.7 on page 21. Additional details about the compilation to reactions are provided in [Pedersen and Phillips, 2009a], and a screen shot of the tool is shown in Figure 1.6 on page 20.

In this section we have illustrated the design of genetic devices in GEC using a simple XOR gate as an example. In order to effectively design more complex devices, however, further work is needed to characterize the properties of individual parts. At present only a few parts are well-characterized and many reaction rates are unknown, so the part and reaction databases described here do not yet exist on a large scale. As one potential consumer of such databases, GEC may help guide how these they are designed and populated with information about biological devices.

5. Proto

Proto is a truly high-level language for synthetic biology, in the sense that a programmer specifies the computation they wish to execute, but the implementation of that computation as a genetic regulatory network is entirely automated. This greatly increases the power of the programmer, at the cost of programs that typically consume more resources than hand-tuned systems. The same sort of optimization techniques that apply to conventional processors, however, can be applied to the genetic



Figure 1.5. Simulation of gfp concentration over time for an exclusive OR gate in GEC, with four combinations of inputs. The simulation uses the chemical reactions of Figure 1.7 on page 21, which were automatically generated from the chosen solution of Figure 1.4 on page 15. The solution exhibits the desired behavior and is a candidate for synthesis.



Figure 1.6. Screen shot of the GEC tool in action. The GEC program is entered on the left as a collection of part types, part properties and logical variables. The design is then compiled to a set of solutions, which can be individually selected. A given solution can then be simulated by the tool in order to observe the expected evolution of the molecular species over time.

regulatory networks generated by Proto, making this a reasonable approach to designing complex synthetic biology systems.

Amorphous Medium and Proto

The original focus of Proto [Beal and Bachrach, 2006] was not synthetic biology, and synthetic biology is still not its primary focus. Rather, it was designed for programming spatial computers—potentially large



Figure 1.7. Network of reactions generated from the design of Figure 1.4 on page 15. The graphical representation on the left was also generated by the GEC tool, and is equivalent to the textual representation on the right.



Figure 1.8. An amorphous medium is a manifold where every point is a general computational device that knows its neighbors' recent past state.

aggregates of locally communicating computing devices distributed to fill a physical space, such as sensor networks, robotic swarms, smart materials, or FPGAs. A colony of cells is also a spatial computer albeit one that may have billions or trillions of devices, rather than the paltry dozens in many sensor networks. Proto's continuous space-time abstraction lets it scale gracefully to such large numbers and its functional dataflow semantics match well with genetic regulatory networks, particularly for describing the spatial differentiation necessary to construct complex multicellular systems like biofilms or tissues.

Proto's approach to the challenges of spatial computing is to focus not on the network of devices, but on the continuous space that they occupy, using the *amorphous medium* abstraction. An amorphous medium [Beal, 2004] is a manifold with a general computational device at every point, where each device knows the recent past state of all other devices in its neighborhood (Figure 1.8). While an amorphous medium cannot, of course, be constructed, it can be approximated on the discrete network of a spatial computer.

Proto uses the amorphous medium abstraction to factor programming a spatial computer into three loosely coupled subproblems: global descriptions of programs, compilation from global to local execution on an amorphous medium, and discrete approximation of an amorphous medium by a real network.

Proto is a functional language that is interpreted to produce a dataflow graph of operations on fields. This program is then evaluated against a manifold to produce a field with values that evolve over time. Proto uses four families of operations: point-wise operations like + that involve neither space nor time, restriction operations that limit execution to a subspace, feedback operations that establish state and evolve it in continuous time, and neighborhood operations that compute over neighbor state and space-time measures and summarize the computed values in the neighborhood with a set operation like integral or minimum.

With appropriate operators, compilation and discrete approximation are straightforward. Thus, Proto makes it easy for a programmer to carry out complicated spatial computations using simple geometric programs that are robust to changes in the network and self-scale to networks with different shape, diameter, density of nodes, and execution and communication properties [Bachrach et al., 2007].

For example, Weiss' band detector [Basu et al., 2005] uses diffusing AHL to detect intermediate distance from a high aTc concentration. This can be implemented using the Proto program:

```
(def band-detect (signal lo hi)
  (and (> signal lo) (< signal hi))))
(let ((signal (diffuse (aTc) 0.8 0.05)))
  (green (band-detect signal 0.2 1)))
```

where \mathbf{aTc} is a function for sensing aTc and green is an actuator that sets the level of GFP expression. Figure 1.9 shows the Proto band detector program interpreted to produce a dataflow graph, then evaluated against an irregularly shaped space. Executing the Proto band detector in simulation produces results equivalent to Weiss's band detector. Figure 1.10 compares execution on a network of 2000 simulated wireless devices distributed randomly through a 100 by 100 unit region with a 10 unit communication radius to Weiss' original results.

Motif-based compilation and optimization

Given a library of devices and standards to compile to, Proto programs can be transformed into genetic regulatory network designs by a process of motif-based compilation. The resulting design can then be optimized using adapted forms of standard code optimization techniques.

The basis of this compilation are associations of each Proto primitive to be compiled with a genetic regulatory network fragment. These are declared as annotations on primitives. For example, the logical **not** operator is associated with a biological inverter motif by the statement shown in Figure 1.11. The first line declares the **not** operator as a primitive that takes a boolean as input and returns a boolean as output. The second line annotates this declaration with a description of a genetic regulatory region—in this case, a strong promoter repressed by whatever protein will represent the **not** operator's input, followed by coding regions for the proteins representing its outputs (each of which is implicitly headed by the necessary ribosome binding site), then finally a terminator.

Motifs can include many other elements as well. For example, a motif can specify particular chemicals to be used, as in the case of the **green** actuator shown in Figure 1.12, whose green fluorescence side effect is implemented by the inclusion of a GFP coding region in the motif. Motifs can also include chemical reactions, as in the case of the **IPTG** sensor



Figure 1.9. A Proto program specifies a dataflow graph of operations on fields. When evaluated on a space, each operation produces a field of values over that space. Here the band detector program is shown evaluated on an irregularly shaped space, with scalar fields grey (lighter is less) and boolean fields colored (**true** is red, **false** is blue). The actuation produced by **green** is shown inside that operation.

shown in Figure 1.13, which uses repression of LacI to detect the presence of the small-molecule signal IPTG. They may even declare internal signaling variables, to be filled in by the compiler, as in the case of the **and** operator shown in Figure 1.14, which implements a non-brancing logical AND using inverter input to a NOR gate.

In order to transform a Proto dataflow computation into an abstract genetic regulatory network, the compiler maps each operator to its associated motif and maps each dataflow edge and internal motif variable to a regulatory protein. These motifs and proteins are then linked together, using the structure of the dataflow graph, to form an abstract genetic regulatory network. The particular choice of chemicals and sequences to implement this network is not fully determined, but left for a later stage of compilation, such as might be provided by systems like GEC [Pedersen and Phillips, 2009b] or Eugene [Berkeley Software 2009 iGem Team, 2010]. An initial set of target chemical rate constants for the network (to be modified as the implementation is determined) are



Figure 1.10. Examples of the Weiss lab band detector in use (a, reprinted by permission from Macmillan Publishers Ltd: Nature ([Basu et al., 2005]), copyright (2005)). The circular regions in the center are active sender bacteria, while the fuzzy areas around them are receiver bacteria expressing fluorescent protein. A Proto implementation produces equivalent results (b) on a network of 2000 simulated devices.





Figure 1.12. Motif declaration for green fluorescence actuator.

filled in from the motifs where specified and filed in by a default set-point in the standards family where not specified.

Consider, for example, the following declaration and use of logical XOR to implement our example program:

(def xor (a b) (or (and a (not b)) (and b (not a))))



Figure 1.13. Motif declaration for IPTG sensor. An aTc sensor uses the same motif, except that aTc replaces IPTG and TetR replaces LacI.



Figure 1.14. Motif declaration for a non-branching logical AND operator. A logical OR uses the same motif, except that all repressors are switched to activators and promoters have low base activity.



Figure 1.15. A Proto data flow computation implementing the XOR example program.

(green (xor (aTc) (IPTG)))

This program should create cells that fluoresce green when precisely one of IPTG or aTc is present at high concentration.

This program is first interpreted to produce the dataflow computation shown in Figure 1.15. Each operator is then mapped to the motifs specified by the declarations shown above. The dataflow edges are assigned to arbitrary regulatory proteins A, B, etc. The consuming motifs set the type of protein, such that A and B are activators, C is a repressor, etc.



Figure 1.16. A Proto dataflow computation is compiled to an abstract genetic regulatory network in two stages. First, each operator is mapped to a motif and each dataflow edge is mapped to a regulatory protein. These elements are then linked together, using the structure of the dataflow graph, to form an abstract genetic regulatory network.

We now have a genetic regulatory network design that implements our high-level computation, though as yet it is still unoptimized and may be extremely inefficient. As we have demonstrated in [Beal and Bachrach, 2008], standard code optimization techniques such as copy propagation, dead code elimination, and common subexpression elimination, can be adapted to operate on genetic regulatory networks.



Figure 1.17. Optimized genetic regulatory network for XOR example.

For example, copy propagation tests whether a protein is being used only to copy a value; if so, the original input may be used directly rather than the copy. In this case of this XOR program, copy propagation changes the input of the GFP-expressing element from A to J. This then leaves protein A not regulating anything. Similarly, copy propagation switches the regulation of J from B to K and from F to L.

Dead code elimination deletes proteins that are not regulating anything, network elements with no products, and proteins that can never be expressed. Since protein A is no longer regulating anything, it is deleted, along with all of the protein coding sequences that can produce it. Since A was the only product of one of the network elements regulated by J, that whole network element is deleted. Likewise, B and Fand their producing elements are deleted by dead code elimination.

Another example of optimization is double negative elimination, which looks for sequences of two inverters and snips them out of the network. In the case of this XOR program, this results in changing the production of E to production of K, since E's only use is to repress D, which in turn represses K. Similarly production of I is changed to production of L. This leaves I and E produced nowhere and D and H unable to be expressed, so dead code elimination deletes another piece of unneeded genetic regulatory network.

These optimizations and more are all applied automatically by the compiler, eventually resulting in the network shown in Figure 1.17. All told, the complexity of the generated network is reduced by approximately 50% in every measure of complexity: from 15 to 8 proteins, from 18 to 9 network elements, and from 7 to 4 stages of propagation delay.

We thus see that high-level computations specified in Proto can be automatically transformed into an abstract genetic regulatory network through a strategy of motif-based compilation. The resulting genetic regulatory network can be optimized using adapted forms of standard code optimization techniques, and might then be mapped onto particular parts from a database using lower-level languages like Eugene or GEC. Although the network is more complex than a hand-optimized design like those encoded in the other tools above, stronger optimizations will likely be able to continue to close the gap, as they have for electronic computers.

6. Other High-Level Design Tools for Biological Computation

We have chosen to focus this chapter on high-level programming languages for *in vivo* bio-molecular computation, where the metaphor of cell as computer holds most strongly. There are a number of related areas outside of this scope, however, in which high-level design tools for bio-molecular systems have been developed, which we now briefly survey.

Macroizing CAD Tools. A number of synthetic biology design tools, such as TinkerCell [Chandran et al., 2009] and SynBioSS [Hill et al., 2008], use biological rules to aid the programmer in designing reaction networks. For example, SynBioSS (the Synthetic Biology Software Suite) is a software suite for the generation, storage, and quantitative simulation of synthetic biological networks. One component of this software suite, called SynBioSS Designer, uses biological rules to create a reaction network given a series of biological parts, such as promoters and ribosome binding sites, and the spatial and temporal connectivity of these parts.

These systems also frequently include the ability to abstract a portion of the network being designed. This type of "macroization" is a step toward a high-level language: the details of the abstracted portion are hidden and it can be given a name that describes its overall function. The programmer must still be aware of the details, however, since the set of parts in the abstracted sub-network are fixed and can interfere with other portions of the design.

Specialized Automated Design Tools. Complementary to Macroizing CAD tools are specialized automated design tools, which might be thought of as limited high-level languages. An example is the boolean circuit design tool recently described in [Marchisio and Stelling, 2010]. Given a truth table mapping inputs to desired outputs, this tool applies the Karnaugh map method from electronic circuit design to find a minimal set of boolean formulas, then maps these formulas onto a library of established bio-molecular boolean gates.

Cell-free Bio-Molecular Computation. A number of biomolecular computation systems have been constructed to operate in cellfree *in vitro* environments, and the design challenges for many of these systems are being addressed with high-level design tools. For example, the VERB compiler [Shea et al., 2009] transforms circuit designs written in Verilog into a biochemical reaction network, and CAD tools have been written to generate DNA origami structures [Rothemund, 2005].

Bio-Inspired Languages. There are a number of biologicallyinspired languages that have been designed to mimic the behavior of engineered biological systems. For the most part, these are at a level of abstraction too high to currently be able to map to a bio-molecular systems implementation, though Weiss' Microbial Colony Language [Weiss, 2001] is close. Many of these languages are focused on pattern formation, such as the Origami Shape Language [Nagpal, 2001], which develops geometric structure through folding, the Growing Point Language [Coore, 1999], which develops topological structure through tropisms. Yet others either model high-level biological development without connection to the details necessary to implement it, as in the case of L-systems [Prusinkiewicz and Lindenmayer, 1990] and MGS [Giavitto et al., 2002], or use biological metaphors for decidedly non-biological programming, as in the case of membrane computing [Paun, 2002].

Modeling languages. Biological modeling languages such as Antimony [Smith et al., 2009], ProMoT [Mirschel et al., 2009], iBioSim [Myers et al., 2009] and little b [Mallavarapu et al., 2009] raise the level of abstraction for constructing models of bio-molecular reactions, but do not directly address the problem of designing computations. For example, Antimony is a modular model definition language that allows scientists to define and use reaction networks. It is designed to be human-writable and acts as an extension to other tools by translating the model to SBML [Finney et al., 2006]. Antimony models composable DNA parts and also allows reaction networks to be abstracted and parameterized, but does not provide any design automation for its user.

7. Summary

In this chapter, we have examined four high-level languages for the design of bio-molecular computing systems. Although the philosophy and the level of abstraction varies between systems, all are fulfilling the same basic goal of hiding complexity from the programmer. Each thus allows a programmer to specify the computing system they wish to create without the full details of how it will be implemented, then automatically generates the remaining details.

At present, none of the available high-level languages can be considered mature. They are, however, an important and rapidly developing research area. Major challenges in the near future for this area include:

- Development of concise high-level abstractions that map well to efficient bio-molecular implementations of a broad range of goals.
- Enhancing the range and quality of automation.
- Integration with other simulation, design, and assembly tools to form complete tool-chains.
- Transitioning from research software to production quality software.

Assuming that progress continues in these areas, however, the advent of high-level programming languages for bio-molecular systems is likely to fundamentally transform the field, much as they have done in computer science, by enabling much more complex bio-molecular systems to be designed more reliably by a vastly larger number of practitioners.

References

- [Bachrach et al., 2007] Bachrach, Jonathan, Beal, Jacob, and Fujiwara, Takeshi (2007). Continuous space-time semantics allow adaptive program execution. In *IEEE SASO 2007*.
- [Basu et al., 2005] Basu, Subhayu, Gerchman, Yoram, Collins, Cynthia H., Arnold, Frances H., and Weiss, Ron (2005). A synthetic multicellular systems for programmed pattern formation. *Nature*, 434:1130–1134.
- [Beal, 2004] Beal, Jacob (2004). Programming an amorphous computational medium. In Unconventional Programming Paradigms International Workshop.
- [Beal and Bachrach, 2006] Beal, Jacob and Bachrach, Jonathan (2006). Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, pages 10–19.
- [Beal and Bachrach, 2008] Beal, Jacob and Bachrach, Jonathan (2008). Cells are plausible targets for high-level spatial languages. In *Spatial Computing Workshop*.
- [Berkeley Software 2009 iGem Team, 2010] Berkeley Software 2009 iGem Team (October 2009, Retrieved May 10, 2010.). Eugene. http://2009.igem.org/Team:Berkeley_Software/Eugene.
- [Cai et al., 2007] Cai, Yizhi, Hartnett, Brian, Gustafsson, Claes, and Peccoud, Jean (2007). A syntactic model to design and verify synthetic genetic constructs derived from standard biological parts. *Bioinformatics*, 23(20):2760–7.
- [Cai et al., 2009] Cai, Yizhi, Lux, Matthew W, Adam, Laura, and Peccoud, Jean (2009). Modeling structure-function relationships in synthetic dna sequences using attribute grammars. *PLoS Comput Biol*, 5(10):e1000529.

- [Cai et al., 2010] Cai, Yizhi, Wilson, Mandy L, and Peccoud, Jean (2010). Genocad for igem: a grammatical approach to the design of standard-compliant constructs. *Nucleic Acids Res*, 38(8):2637–44.
- [Chandran et al., 2009] Chandran, Deepak, Bergmann, Frank, and Sauro, Herbert (2009). Tinkercell: modular cad tool for synthetic biology. *Journal of Biological Engineering*, 3(1):19.
- [Chomsky, 1956] Chomsky, N. (1956). Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124.
- [Coore, 1999] Coore, Daniel (1999). Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer. PhD thesis, MIT.
- [Czar et al., 2009] Czar, Michael J, Cai, Yizhi, and Peccoud, Jean (2009). Writing dna with genocad. *Nucleic Acids Res*, 37(Web Server issue):W40–7.
- [Densmore et al., 2010] Densmore, Douglas, Hsiau, Timothy H-C, Kittleson, Joshua T, DeLoache, Will, Batten, Christopher, and Anderson, J Christopher (2010). Algorithms for automated dna assembly. *Nucleic Acids Res*, 38(8):2607–16.
- [Densmore et al., 2009] Densmore, Douglas, Van Devender, Anne, Johnson, Matthew, and Sritanyaratana, Nade (2009). A platform-based design environment for synthetic biological systems. In TAPIA '09: The Fifth Richard Tapia Celebration of Diversity in Computing Conference, pages 24–29, New York, NY, USA. ACM.
- [Finney et al., 2006] Finney, Andrew, Hucka, Michael, Bornstein, Benjamin J., Keating, Sarah M., Shapiro, Bruce M., Matthews, Joanne, Kovitz, Benjamin K., Schilstra, Maria J., Funahashi, Akira, Doyle, John, and Kitano, Hiroaki (2006). Software infrastructure for effective communication and reuse of computational models. In Szallasi, Zoltan, Stelling, Jörg, and Periwal, Vipul, editors, System Modeling in Cell Biology: From Concepts to Nuts and Bolts. MIT Press.
- [Giavitto et al., 2002] Giavitto, Jean-Louis, Godin, Christophe, Michel, Olivier, and zemyslaw Prusinkiewicz, Pr (2002). Computational models for integrative and developmental biology. Technical Report 72-2002, Univerite d'Evry, LaMI.
- [Hill et al., 2008] Hill, Anthony D., Tomshine, Jonathan R., Weeding, Emma M. B., Sotiropoulos, Vassilios, and Kaznessis, Yiannis N.

(2008). Synbioss: the synthetic biology modeling suite. *Bioinformatics*, 24(21):2551–2553.

- [Mallavarapu et al., 2009] Mallavarapu, Aneil, Thomson, Matthew, Ullian, Benjamin, and Gunawardena, Jeremy (2009). Programming with models: modularity and abstraction provide powerful capabilities for systems biology. *Journal of The Royal Society Interface*, 6(32):257– 270.
- [Marchisio and Stelling, 2010] Marchisio, Mario Andrea and Stelling, Jorg (2010). Automatic design of digital synthetic gene circuits. In 2nd International Workshop on Bio-Design Automation.
- [Mirschel et al., 2009] Mirschel, S., Steinmetz, K., Rempel, M., Ginkel, M., and Gilles, E. D. (2009). Promot: Modular modeling for systems biology. *Bioinformatics*, 25(5):687–689.
- [Myers et al., 2009] Myers, C.J., Barker, N., Jones, K., Kuwahara, H., Madsen, C., and Nguyen, N.P. (2009). ibiosim: a tool for the analysis and design of genetic circuits. *Bioinformatics*, 25:2848–9.
- [Nagpal, 2001] Nagpal, Radhika (2001). Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics. PhD thesis, MIT.
- [Nathan J. Hillson, 2010] Nathan J. Hillson (Retrieved September 28, 2010.). j5 automated dna assembly software. http://jbeiexwebapp.lbl.gov/j5.
- [Paun, 2002] Paun, G. (2002). Membrane computing: An introduction. Springer.
- [Pedersen and Phillips, 2009a] Pedersen, Michael and Phillips, Andrew (2009a). Towards programming languages for genetic engineering of living cells. J R Soc Interface, 6 Suppl 4:S437–S450.
- [Pedersen and Phillips, 2009b] Pedersen, Michael and Phillips, Andrew (2009b). Towards programming languages for genetic engineering of living cells. Journal of the Royal Society Interface.
- [Prusinkiewicz and Lindenmayer, 1990] Prusinkiewicz, Przemysław and Lindenmayer, Aristid (1990). The Algorithmic Beauty of Plants. Springer-Verlag, New York.
- [Purnick and Weiss, 2009] Purnick, Priscilla E M and Weiss, Ron (2009). The second wave of synthetic biology: from modules to systems. *Nat Rev Mol Cell Biol*, 10(6):410–422.

- [Rothemund, 2005] Rothemund, Paul W. K. (2005). Design of dna origami. In International Conference on Computer-Aided Design (IC-CAD).
- [Shea et al., 2009] Shea, Adam, Fett, Brian, Riedel, Marc, and Parhi, Keshab (2009). Writing and compiling code into biochemistry. In International Conference on Computer-Aided Design.
- [Smith et al., 2009] Smith, Lucian P., Bergmann, Frank T., Chandran, Deepak, and Sauro, Herbert M. (2009). Antimony: a modular model definition language. *Bioinformatics*, 25(18):2452–54.
- [Sudkamp, 2006] Sudkamp, Thomas A (2006). Languages and machines: an introduction to the theory of computer science. Pearson Addison-Wesley, Boston, 3rd ed edition.
- [Weiss, 2001] Weiss, Ron (2001). Cellular Computation and Communications using Engineered Genetic Regulatory Networks. PhD thesis, MIT.