Eugene's Enriched Set of Features to Design Synthetic Biological Devices

Haiyao Huang, Ernst Oberortner, Douglas Densmore
Department of Electrical and Computer Engineering
Boston University
{huangh,ernstl,dougd}@bu.edu

Allan Kuchinsky
Agilent Technologies
allan_kuchinsky@agilent.com

ABSTRACT

Eugene is a design language to support synthetic biologist in order to construct large and complex biological devices more accurately. Compared to its original version, Eugene provides now an enriched set of functionalities to specify and constrain synthetic biological devices and their design synthesis. This work highlights (1) the declaration of devices at various abstraction levels, (2) the control-flow management of design synthesis, (3) a design space exploration to generate devices, and (4) the prototyping of functions. Eugene allows synthetic biologists to specify, design, and constrain a large number of biological devices in a few lines of code, without having to specify every single device manually.

1. INTRODUCTION

The most common view in synthetic biology is to view DNA sequences as parts with certain properties to form composite parts, devices, or systems. Design languages that support various level of abstractions can make a synthetic biologist's life easier. Whereas Eugene's initial version [2] has focused on structure and functionality we present in this paper Eugene's new set of facilities, which includes the declaration of devices at various abstraction levels, the management of the control-flow of design synthesis, the automatic generation of devices, and user-defined reusable functions. Compared to other languages in the synthetic biology domain, Eugene offers certain advantages in the areas of flexibility, simple syntax, compatibility with other design tools, and extensibility.

2. EUGENE'S NEW FEATURES

Specifying Devices at Various Levels of Abstraction

Eugene allows synthetic biologist to design abstract, instantiated, and hybrid synthetic devices.

Abstract devices are assembled of part types, such as promoters, ribosome binding sites, or terminators. Instantiated devices are either instances of abstract devices or assembled of various parts, such as *pLac* or *lacI*. If an instantiated device is an instantiates an abstract device, the device's parts are ordered as specified in the abstract device. Hybrid devices are assembled of devices, part types, and parts.

We illustrate in Listing 1 examples of defining an abstract, instantiated, and hybrid inverter. To the best of our knowledge, there exists no language to design such types of synthetic devices.

```
/* Define an abstract Inverter */
Device Abstract_Inverter(
    Promoter, RBS, Repressor, Terminator,
    Promoter, RBS, Reporter, Terminator);

/* Instantiate the abstract Inverter */
Abstract_Inverter Instantiated_Inverter(
    pBad, BBa_J61100, cI, BBa_B0015,
    pCI, RFPc, BBa_B0015);

/* Declare a hybrid Inverter */
Device Hybrid_Inverter(
    Promoter, RBS, cI, BBa_B0015,
    Promoter, RBS, RFPc, BBa_B0015);
```

Listing 1: Declaration of Synthetic Devices

Control-Flow Facilities

Similar to computer programming languages, Eugene offers to its users conditional branches — `if-else` — and loop statements — `for`, `while`, and `do-while`. Conditional branches and loops make it possible to manage the control-flow of design synthesis (see Listing 4). Also, control-flow facilities reduce the lines of redundant code and allow to apply specific constraints various times in case of certain conditions.

Automatic Design Space Exploration

Eugene offers two built-in functions to automatically generate synthesized devices — `permute` and `product`. Though the syntax of both statements is equivalent, both functions generate devices differently. The `product` function changes the assembling parts while maintaining the order of the device's components. The `product` function takes a device and all available parts in the design space, and creates all possible variations of the given device while maintaining the order of the device's components. Given a device composed of n components, and m parts in the design space, the `product` function will generate m^n variations of that device. The `product` function allows, for example, to rapidly generate all instances of an abstract device that adhere to a given set of rules. The `permute` function permutes the order of a device's components. The `permute` function collects all defined parts and creates all possible permutations of them that comply to a given device's structure. For example, the `permute` function permutes the components of an abstract device. Hence, given a device composed of n parts, the permute function will generate $n!$ permutations of the parts.

```
/* Define two Rules */
Rule r01(STARSWITH Promoter);
Rule r02(ENDSWITH Terminator);

/* PRODUCT */
product(Abstract_Inverter, strict, 100);

/* PERMUTE */
permute(Abstract_Inverter, strict);
```

Listing 2: product and permute

In Listing 2 we define two rules and exemplify the functions' utilization. Both functions can take up to three arguments and return a list of the generated devices. Only the first argument — the input device — is required whereas the second and third arguments are optional. For the second parameter, which specifies the level of rule enforcement, two options can be specified: `strict` and `flexible`. The `strict` option only generates devices that obey the specified rules, while the `flexible` option, which is default, generates every possible device and labels them if they violate a rule. The third parameter is an integer number which limits the number of the generated devices. If the Eugene user calls the `product` or `permute` with a capacity smaller than the total number of possible variations, it will generate a random subset of the that size.

Function Prototyping

Eugene offers a rich set of built-in functions that are not described in this paper due to space restrictions. However, for synthetic biologists it is important to defining their own functions and parameters. Hence, Eugene offers facilities to extend the repertoire of functions. In Listing 3 we present an example of creating a function that returns the number of promoters in a given device.

```
// function defintion
function num countPromoters(Device d) {
    num nrOfPromoters = 0;
    for(num i=0; i<d.size(); i++) {
        if(d[i] INSTANCEOF Promoter) {
           nrOfPromoters++;
        }
    }
    return nrOfPromoters;
}

// call the function
num nr = countPromoters(Abstract_Inverter);
```

Listing 3: Function Prototyping

3. AN EXAMPLE OF USING EUGENE'S NEW FEATURES

The example in Listing 4 focuses on the replacement of an inverter's promoters whose strength is lower than a given threshold. First, we use the `product` function in order to generate all instantiated inverters from the design space whose structure equals to given `Abstract_Inverter` device. Next, we iterate over all generated devices and each device's components, to check if the current component is a `Promoter`, and if its strength is lower then the threshold `T`. If so, we replace the current promoter with a new promoter from the design space by calling the defined `getPromoter()` function. In the `getPromoter` function, we iterate over all promoters returned by Eugene's `getAllPromoters()` function, and return the first promoter with a strength higher then the given threshold `T`.

```
// generate all instances of an Inverter
Device[] arrDevices = product(
    Abstract_Inverter);

// evaluate all generated Inverters
num T = 6.2;
for(num i=0; i<arrDevices.size(); i++)  {
  inverter = arrDevices[i].
  for(num k=0; k<inverter.size(); k++) {
    if(inverter[k] INSTANCEOF Promoter AND
       inverter[k].strength < T) {
        // replace the current promoter
        // with the new promoter returned
        // by the getPromoter function
        inverter[k] = getPromoter(T);
      }
    }
  }
}

// define a function
function Promoter getPromoter(num T) {
  // find a promoter in the design space
  // whose strength is higher then
  // the given threshold T
  for(Promoter prom : getAllPromoters()) {
    if(prom.strength > T) {
      return prom;
    }
  }
}
```

Listing 4: Using Eugene's New Features to Replace a Device's Promoters

4. CONCLUSION AND FUTURE WORK

In this paper, we exemplified the new features of Eugene, namely to (1) design of devices at various levels of abstraction, (2) specify the control-flow of design synthesis, and (3) to generate devices automatically, and (4) to specify user-defined functions. We are currently working on the integration with Synthetic Biology Open Language (SBOL), making it easier to exchange synthetic biological between tools. Furthermore, we are planing to release Eugene with a user-friendly IDE for the International Genetically Engineered Machine (iGEM) 2012 competition [1], allowing the iGEM teams to evaluate Eugene's features and usability. In the future, we want to provide facilities to specify families of parts and devices, enhance the specification of rules, as well as to query characterization data of parts and devices. We believe that the Eugene language is a great step towards a full support of synthetic biologists in order to design and build complex and efficient synthetic biological systems automatically.

5. REFERENCES

[1] International Genetically Engineered Machine (iGEM) Foundation. `http://igem.org`.

[2] Bilitchenko, L. *et al.* Eugene: A domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PLoS ONE 6*, 4 (2011).