EUGENE — A DOMAIN-SPECIFIC LANGUAGE FOR CONSTRAINT-BASED DESIGN OF SYNTHETIC BIOLOGICAL SYSTEMS

Ernst Oberortner and Douglas Densmore

August 22, 2014

Boston University

Department of Electrical and Computer Engineering

Technical Report No. ECE-2014-NN

BOSTON
UNIVERSITY

EUGENE — A DOMAIN-SPECIFIC LANGUAGE FOR CONSTRAINT-BASED DESIGN OF SYNTHETIC BIOLOGICAL SYSTEMS

Ernst Oberortner and Douglas Densmore

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
`www.bu.edu/ece`

August 22, 2014

Technical Report No. ECE-2014-NN

Here comes a special preface page. For example, if the report is restricted, then a suitable note can be included. This page can also be used to indicate to whom the document is dedicated, etc.

Summary

Summary of the report. Maximum 1 page.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Language characteristics

- rule-based design paradigm ⇒ the specification of synthetic biological designs based on biological constraints and rules
- multi-paradigm language ⇒ combines declarative and imperative language features
- declarative features ⇒ the user specifies *WHAT* the design problem is based on rules and constraints; invokes *built-in functions*
- imperative features ⇒ *HOW* to (1) process the results returned by Eugene's declarative features and/or (2) build the *WHAT* question to ask Eugene
- interpreted language ⇒ a given Eugene script is being executed "on-the-fly". That is every statement is executed immediately. The Eugene parser is a one-phase parser, which requires that variables, biological facts, and functions must be declared before being used and/or referred to them.
- strongly typed language ⇒ three primitive types (num, txt, bool), two types of arrays (num and txt), language model for typing the biological facts (primitive and composite).

In this manual, we describe the syntax of the Eugene language using the Extended-Backus-Naur Form (EBNF).[1]

Chapter 2

Language Basics

Eugene's syntax is case sensitive. Language statements are separated by the colon operator (;).

2.1 Comments

Eugene supports two types of comments: single-line comments and multi-line comments.

```
// single line comment

/*
this is a
multi-line
comment
*/
```

2.2 Primitive Types

Eugene supports five primitive types: which can be either a real number (**num**), a string of characters (**txt**), an array of floating point numbers (**num**[]), an array of strings (**txt**[]), or a boolean (**bool**).

```
primitive-type ::= num | txt | num[] | txt[] | bool
```

2.3 Primitive Values / Atoms

Based on the supported primitive types, Eugene supports strings (i.e. character sequences enclosed in double quotas) and floating point numbers. In Eugene, identifiers are unique. That is, no two variables can have the same identifier. Identifiers are also case-sensitive.

```
string ::= " [^"] "
number ::= ( [0-9] )* (. ( [0-9] )* )?
```

2.4 Identifiers

In Eugene, identifiers must be unique. An identifier is a sequence of characters that is constraint as follows:

```
id ::= [a-zA-Z] ( [a-zA-Z0-9_] )*
```

In a Eugene script, the identifiers must be defined before their usage and/or referring to them.

2.5 Declaration of Variables

Variables have a unique identifier (see Section 2.4) and a primitive type (see Section 2.2). Depending on the variable's type, the variable can take only values of the specified type (see Section 2.3). Values can either be constants or expressions. Expressions are calculated during the execution of a Eugene script. Values can be assigned to variables either at the point of the variable's declaration or at later points of execution.

⟨*variable-declaration*⟩ ::= ⟨*primitive-type*⟩ ⟨*id*⟩ ;

⟨*primitive-type*⟩ ::= **txt** | **txt[]** | **num** | **num[]** | **bool**

⟨*id*⟩ ::= [a-zA-Z] ([a-zA-Z0-9_])*

Examples:

- declaration of a variable of type `num` with the identifier `n`:
  `num n;`
- declaration of a string variable with the identifier `s`:
  `txt s;`

2.6 Assignment Statements

Assignment statements assign values to variables and are interpreted from right to left, i.e., the result of the expression specified on the right-hand-side (RHS) is assigned to the variable specified on the left-hand-side (LHS). The LHS and the RHS are separated by the equals operator (`=`).

⟨*assignment*⟩ ::= ⟨*lhs*⟩ = ⟨*expression*⟩ ;

⟨*lhs*⟩ ::= ⟨*variable-declaration*⟩ | ⟨*id*⟩

⟨*expression*⟩ ::= ⟨*multiplication*⟩ | ⟨*division*⟩

⟨*multiplication*⟩ ::= ⟨*addition*⟩ | ⟨*subtraction*⟩ (* ⟨*expression*⟩)?

⟨*division*⟩ ::= ⟨*addition*⟩ | ⟨*subtraction*⟩ (/ ⟨*expression*⟩)?

⟨*addition*⟩ ::= ⟨*atom*⟩ (+ ⟨*expression*⟩)?

⟨*subtraction*⟩ ::= ⟨*atom*⟩ (- ⟨*expression*⟩)?

⟨*atom*⟩ ::= ⟨*id*⟩ | ⟨*number*⟩ | ⟨*string*⟩ | (⟨*expression*⟩)

2.7 Error Reporting

In Eugene v2.0 we differentiate among various types of errors ranging from syntax errors, over unknown identifiers and incompatible types, to inconsistent rules. Any type of error is thrown as a Java `IllegalArgumentException` and reported to the output console. Also, the interpretation of a defective Eugene script stops after printing the error to the console. Error messages have the following format:

```
@Error
Line <line-no> Position <position-in-line>
<Error-Message>
```

2.8 Reserved Words

Eugene has the following reserved words which can not be used as identifiers:

```
Property Part PartType \todo{Type} Device \todo{Template} Rule
```

The following list of reserved words can be specified in lower case too:

```
REPRESSES INDUCES DRIVES MORETHAN CONTAINS EXACTLY SAME_COUNT WITH
THEN BEFORE ALL_BEFORE SOME_BEFORE  AFTER ALL_AFTER SOME_AFTER NEXTTO
ALL_NEXTTO SOME_NEXTTO STARTSWITH ENDSWITH EQUALS ALL_FORWARD
ALL_REVERSE SOME_FORWARD SOME_REVERSE FORWARD REVERSE SAME_ORIENTATION
ALTERNATE_ORIENTATION AND OR NOT IF THEN ELSEIF ELSE FOR FUNCTION
```

keep this list updated

Chapter 3

Declarative Language Features

Eugene v2.0 supports the specification of synthetic biology designs based on biological *Facts* and *Rules*. Additionally, Eugene v2.0 supports the invocation of user-defined *Functions* for the verification and enumeration of rule-compliant synthetic biology designs.

⟨*script*⟩ ::= ⟨*declaration*⟩+

⟨*declaration*⟩ ::= ⟨*fact*⟩ | ⟨*rule*⟩ | ⟨*function*⟩

3.1 Declaration of Biological Facts

Facts can either be specified manually or imported using data exchange standards. Here, we define the Eugene language for the manual specification of facts. We describe Eugene's data exchange facilities in Section 4. Every specification of a fact must be closed with the semi-colon character (;).

⟨*fact*⟩ ::= (⟨*property*⟩ | ⟨*type*⟩ | ⟨*instantiation*⟩ | ⟨*relation*⟩) **;**

In the following, we define the syntax for the specification of properties, types, instantiations, and relations in more detail.

3.1.1 Properties of Biological Facts

Properties represent attributes and characteristics of genetic components, either primitive components or composite components. Each property must have an identifier (see Section 2.4) and a primitive type (see Section 2.2).

⟨*property*⟩ ::= **Property** `id` (⟨*primitive-type*⟩)

Example:

3.1.2 Types

Types are abstract representations of the common structure of genetic elements, such as parts. Eugene v2.0 only supports the definition of types of parts. Types define the properties and structure of genetic elements. For example, a type `Promoter` can defines the common characteristics (i.e. properties) of genetic elements, such as DNA sequence, list of operators, or strength.

⟨*type*⟩ ::= ⟨*part-type*⟩

⟨*part-type*⟩ ::= **PartType** `id` `(` ⟨*list-of-ids*⟩ `)`

⟨*list-of-ids*⟩ ::= `id` (`,` ⟨*list-of-id*⟩)?

For the specification of part types, the identifiers in the `<list-of-ids>` must refer to the identifiers of specified properties (see Section 3.1.1). As defined in Section 2.4, properties must have been defined before specifying a type.

3.1.3 Instantiations

In Eugene, types define the common properties and structure of genetic elements. Hence, Eugene also provides facilities to instantiate types with concrete genetic facts. That is, instances of genetic types have a unique name and property values. For examples, a concrete instance of the type `Promoter` can have a specific DNA sequence (e.g. `ATCG`).

⟨*instance*⟩ ::= `id id` `(` ⟨*list-of-property-values*⟩ `)`

⟨*list-of-property-values*⟩ ::= (⟨*dot-notation*⟩ | ⟨*list-notation*⟩)

⟨*dot-notation*⟩ ::= `.` `id` `(` `primitive-value` `)` (`,` ⟨*dot-notation*⟩)

⟨*list-notation*⟩ ::= ⟨*value*⟩ (`,` ⟨*list-notation*⟩)

The first identifier of an instance definition refers to the type of the instance (see Section 3.1.2). The second identifier is the name of the instance. In Eugene, we support two notations of specifying the property values of instances, the dot (.) notation and the list notation. Using the list notation necessitates to specify the property values in the same order as specified in the type specification. The dot notation offers the possibility to specify the property name of the property to that the specified value must be assigned.

Eugene does check the types of the property values with the type of the property. For example, it is not possible to assign a `num` value to a property of type `txt`.

In Eugene v2.0, we do not support the assignment of property values to undefined properties. Each property must be defined in the type specification.

3.1.4 Composite Biological Facts

3.1.5 Relations

3.2 Design Templates

3.3 Rules and Constraints

In Eugene, rules are conditions which can only take the values **true** or **false**.

3.3.1 Structural Constraints

integrate table from miniEugene paper

3.3.2 Relational Expressions

Relational expressions are rules of the following form:

⟨*rel-exp*⟩ ::= ⟨*exp*⟩ ⟨*rel-operator*⟩ ⟨*exp*⟩

⟨*exp*⟩ ::= ⟨*exp-operand*⟩ (⟨*exp-operator*⟩ ⟨*exp-operand*⟩)*

⟨*exp-operand*⟩ ::= ⟨*property*⟩ | ⟨*constant*⟩

⟨*property*⟩ ::= `id` (⟨*index*⟩)* (`.` `id` (⟨*index*⟩)*)* (`.` ⟨*function*⟩)?

⟨*constant*⟩ ::= `number` | `string`

⟨*exp-operator*⟩ ::= + | - | * | /

⟨*index*⟩ ::= [`number`]

⟨*function*⟩ ::= **size** | **length** | ...

⟨*rel-operator*⟩ ::= < | <= | == | != | > | >=

Examples:

- Select all promoters whose strength is greater than 5:
  `Promoter.strength > 5`
- Select all pairs of repressors and promoters that have the following relation:
  `Repressor.repress == Promoter.name`
- Select all pairs of promoters and RBSs that have following relation:
  `Promoter.sequence.size + RBS.sequence.size <= 500`
- Select all devices on that the following equation holds **true**.
  `Device[0].sequence.size <= Device[1].sequence.size`
- find and provide more examples

3.4 Containers for Biological Facts and Rules

Eugene v2.0 provides two types of containers for biological facts and rules:

- **Collections** are unordered groups of biological data.
- **Arrays** are ordered groups of biological data.

Eugene v2.0 also supports nested container types. That is, collections can contain collections and arrays, as well as arrays can contain other arrays and collections.

3.4.1 Collections

3.4.2 Arrays

Chapter 4

Data Exchange Capabilities

Eugene v2.0 supports the following types of data exchange facilities:

- INCLUDE of Eugene scripts
- IMPORT of Eugene containers
- IMPORT of biological data from the iGEM partsregistry[1]
- IMPORT and EXPORT of biological data from Genbank files[2]
- IMPORT and EXPORT of biological data compliant with the Synthetic Biology Open Language (SBOL) standard[3]
- VISUALIZATION of designs compliant with the SBOL Visual graphical notation[4] using Pigeon[2]

4.1 Inclusion of Eugene Scripts

Eugene v2.0 supports to include Eugene scripts into a Eugene script via the `include` keyword. The syntax is defined as follows:

⟨*include-statement*⟩ ::= **include** `string`

The string argument must specify the relative location of the included Eugene script to the actual Eugene script. If the specified Eugene script does not exist or is not a well-formed Eugene script, then an error is reported.
Examples:

- Select all promoters whose strength is greater than 5:

[1] http://parts.igem.org
[2] http://www.ncbi.nlm.nih.gov/genbank/
[3] http://www.sbolstandard.org/
[4] http://www.sbolstandard.org/visual

Note! In Eugene v2.0 we do not check for cyclic inclusions of Eugene script. For example, if the Eugene script A includes Eugene script B and Eugene script B includes Eugene script A, then an endless loop is the result.

4.2 Data Import of Eugene Containers

4.3 Data Import from the iGEM Partsregistry

4.4 GenBank Data Import and Export

4.5 SBOL Data Import and Export

4.6 SBOL Visual Compliant Design Visualization

Chapter 5

Imperative Language Features

Imperative languages features serve for the specification of the control-flow of design synthesis. In Eugene, imperative language features enable to

- control the execution of statements if certain conditions are satisfied (Section 5.1),
- iterate over the elements of a Eugene container, which are either collections or arrays (Section 5.2),
- group recurring blocks of statements into reusable functions (Section 5.3), and
- execute blocks of statements multiple times while a certain condition is satisfied (Section 5.4).

Besides user-defined functions, Eugene also provides various built-in functions which can be invoked within a Eugene script (see 5.5).

5.1 Conditional Branches

5.2 Iterators

5.3 Function Prototypes

5.4 Loops

5.5 Built-in Functions

5.5.1 Design Verification

Built-in functions:

assert

note

5.5.2 Design Enumeration

Built-in functions:

product

permute

If a Eugene script contains a function prototype with an equivalent signature as one built-in function, then the built-in function is overwritten by the user-prototyped function.

Chapter 6

Algorithms

6.1 Enumeration of rule-compliant designs

6.2 Verification of designs and rules

Appendices

Appendix A

Railroad Diagrams

Bibliography

[1] Standard, E. S. S. *http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf*

[2] Bhatia, S.; Densmore, D. *ACS Synthetic Biology* **2013**, *2*, 348–350.