AOSB Manual

Release alpha

Philipp Boeing

May 01, 2014

CONTENTS

Contents:

CHAPTER

ONE

TUTORIAL

1.1 Introduction

This tutorial will introduce the main constructs of the AOSB framework and how to use them to build an aspect oriented synthetic biology design. However, it is not written as an in-depth tutorial into thinking about synthetic biology systems in terms of cross-cutting concern separation and aspect modularization. It simply serves as an introduction to the AOSB language defined by the framework.

1.2 Prerequisites

AOSB is a framework for Python 2.7. The tutorial assumes at least basic knowledge of the Python programming language and object oriented programming as well as a conceptual awareness of aspect oriented programming.

The AOSB package must also be added to the Python path.

1.3 Hello World

AOSB allows a genetic circuit, i.e. a synthetic biology "program" to be defined and manipulated in Python. Let us start by creating a simple example. In an empty Python file, first import the AOSB framework:

```
from aosb import *
```

Next, we will define a simple genetic circuit expressing an undefined Protein:

```
class SimpleCircuit(Circuit):
    def mainCircuit(self):
        self.addPart(Promoter)
        self.addPart(RBS)
        self.addPart(CodingRegion(Protein))
        self.addPart(Terminator)
```

Every AOSB design must begin with a circuit, and every circuit needs to define a mainCircuit() method. Similarly to a main method in most modern programming languages, this method defines the beginning of the synthetic biology design.

The Circuit class provides the addPart method, which we are using to add four parts to the design. The order of instruction matters and sets the order in which the parts appear in the circuit.

Lastly, to view the defined genetic circuit, we must "compile" our design using the AOSB.Weaver:

```
compiledDesign = Weaver(SimpleCircuit).output()
print compiledDesign
```

Running the entire file should give this output:

```
Promoter+RBS+CodingRegion(codesFor = Protein)+Terminator
```

1.4 Parts and Molecules

In the Hello World example, we have used two basic building blocks of AOSB: Parts and Molecules.

Every "instruction" in the genetic circuit is a `Part`, added to the execution flow via `addPart()`. Parts follow a strict hierarchy, as seen in the UML diagram.

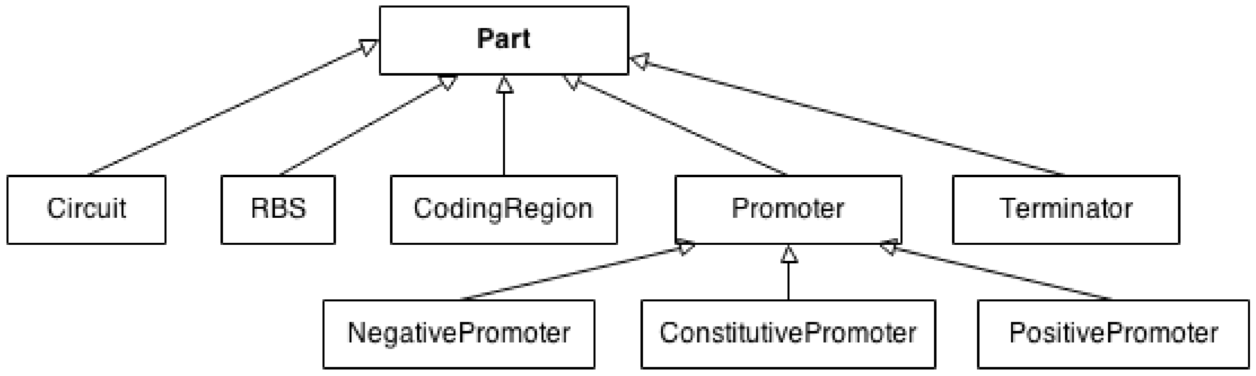

Figure 1.1: Hierarchy of AOSB built in Part classes.

All Parts have the capacity to be connected to a Molecule class. Of the standard parts, `CodingRegion`, `PositivePromoter` and `Negative Promoter` require a Molecule class to be specified. For example, in the Hello World example, the coding region is initialized with *Protein*, a built in sub class of *Molecule*.

Additionally, the Weaver assigns Part attributes `before` and `after`, to refer to the Parts before and after it in the Part list, and a `namespace` attribute, referring to the Circuit or Aspect which added it.

Warning: Molecules are always used as **classes**, not as objects. This is because they represent an interaction of a Part with a **type** of Molecule, not just a specific Molecule.
On the other hand, Parts are always used as objects. However, for syntactic convenience, if a Part requires no parameters to be initialized, `addPart()` accepts Part classes and automatically converts them into objects. This can be seen in the Hello World example: The Promoter, RBS and Terminator Parts are added as classes. This is equivalent to adding them as an object, i.e. `addPart(Promoter())`. The `CodingRegion` class however needs a Molecule to be instantiated, and thus needs to be added as an object.

Parts are a central element of AOSB, because they represent the basic building blocks of the synthetic biology target.

Because each part is a class, new or more concrete Parts and Molecules can be added in the normal Python way, potentially defining additional attributes:

```
# Parts Registry RBS (http://parts.igem.org/Help:Ribosome_Binding_Site)
class BBa_B0030     (RBS):
    sequence = "attaaagaggagaaa"
    bindingEfficiency = 0.6
```

```
# declare that a class of Proteins called exists that are used as markers
class ReporterProtein (Protein):
    pass
```

Creating new, specialized Parts and Molecules will be a frequent occurance in an AOSB program, so the language also provides the `declareNewPart` and `declareNewMolecule` convenience functions:

```
declareNewMolecule("GFP",ReporterProtein)
# after this line, a new class GFP has been exported to the namespace

declareNewPart("GFPCodingRegion",CodingRegion,GFP)
# a new class GFPCodingRegion, subtype of CodingRegion has been created
# the third parameter can specify a Molecule
```

`declareNewMolecule` can accept more than one parent class of type Molecule, so that a Molecule could be tagged by various classes, e.g. ReporterProtein, FluorescentProtein, etc. This will be useful once we start selecting Molecules with point cuts. See the reference on `declareNewPart()` and `declareNewMolecule()` for more information.

1.5 Aspects

The purpose of AOSB is to allow synthetic biology core and cross-cutting concerns to be modularized. This requires constructs that define weaving rules, which can be used to manipulate the execution flow, i.e. the genetic circuit.

Weaving rules are bundled into "Aspect classes":

```
class EmptyAspect(Aspect):
    def mainAspect(self):
        # advice must be registered here
        pass
```

1.5.1 Design Rules Example

A simple example shows how an Aspect can be used to automatically weave RBS and Terminator parts around Coding Regions:

```
from aosb import *

class CodingGFP(Circuit):
    def mainCircuit(self):
        declareNewMolecule('GFP')
        self.addPart(Promoter)
        self.addPart(CodingRegion(GFP))

class DesignRules(Aspect):
    def mainAspect(self):
        anyCodingRegion = PartSignature('*.CodingRegion+')

        beforeCodingRegion = PointCut(anyCodingRegion,PointCut.BEFORE)
        afterCodingRegion = PointCut(anyCodingRegion,PointCut.AFTER)

        self.addAdvice(beforeCodingRegion,self.insertRBS)
        self.addAdvice(afterCodingRegion,self.insertTerminator)

    def insertRBS(self,context):
        self.addPart(RBS)
    def insertTerminator(self,context):
        self.addPart(Terminator)

compiledDesign = Weaver(CodingGFP,DesignRules).output()
print compiledDesign
```

The output of this program is:

```
Promoter+RBS+CodingRegion(codesFor = GFP)+Terminator
```

In the following sections, the constructs introduced in the example will be covered in detail.

The simple defines the "core concern", GFP coded under an unspecified Promoter, in the CodingGFP Circuit *(lines 3-7)*. The DesignRules Aspect *(lines 9-23)* defines two advice methods, `insertRBS" and ``insertTerinator`. Each Aspect has to implement a `mainAspect()` method, similarly to a Circuit's `mainCircuit()`. During the execution of `mainAspect()`, all advice of the Aspect must be declared. In the example *(lines 16 and 17)*, the two advice methods are attached two the appropriate PointCut and declared using `addAdvice()`.

Lastly, CodingGFP and DesignRules are given as inputs to the Weaver *(lines 25)*.

The Weaver now first constructs the classes and creates a list of all advice, by calling DesignRule's `mainAspect()`. Then, it calls CodingGFP's `mainCircuit()` method. When the CodingRegion is added *(line 7)*, the Weaver is able to match the advice from DesignRules and inserts a call to `insertRBS()` before adding the CodingRegion, and a call to `insertTerminator()` after it.

1.6 Point Cuts

A `PointCut` matches specific join points in the genetic circuit execution flow.

The `Point Cut constructor` requires a PartSignature and an operator of type `PointCut.BEFORE` / `AFTER` or `REPLACE`, which specifies how the matching join point should be manipulated.

1.6.1 Simple Part Signatures

In the Design Rules Example we have seen an example of a simple PartSignature:

```
.. code-block:: python
```

PartSignature('*.CodingRegion+')

This part matches any part of type (or sub-type) `CodingRegion`.

A Part Signature consists of three parts. Firstly, it can specify a namespace of a Part, i.e. the Circuit or Aspect which has added it. Secondly, it can specify the part itself. Thirdly, it can optionally specify a Molecule the part should be connected to. Matching is done on the name of the type.

```
NameSpaceTypeName.PartTypeName(MoleculeName)
```

Additionally, the wild card operator `*` will match any name, and the `+` will make any type or subtype a match. Furthermore, a `!` at the beginning of any element of the signature will invert it.

Examples

```
"Circuit+.BB*(*)"
```

will match any part, which has been added by a Circuit (i.e. not by an Aspect) and whose type name begins with "BB" (e.g. by naming convention, a BioBrick part). The molecule connection is irrelevant, so `(*)` for the molecule element of the signature is equivalent to leavning it out.

```
"Aspect+.Promoter+(TranscriptionFactor+)"
```

will match any Promoter (including subtypes such as NegativePromoter, PositivePromoter, any any user-defined types) that has been added by an Aspect and is regulated by a TranscriptionFactor.

```
"!SimpleAspect.*(!GFP)"
```

will match any Part not being connected to GFP which has not been added by an Aspect called SimpleAspect.

Thus, by constructing a sensible typing hierarchy of Parts and Molecules and by establishing a rigorous naming convention, Point Cuts are able to select groups of Parts based on precise features.

1.6.2 Point Cut Expressions

Additionally, Part Signatures can be connected into expressions, using the `&`, `|` and `%` operators, which stand for boolean-and, boolean-or and concatenation, respectively.

Examples

```
PartSelector('*.NegativePromoter+') | PartSelector('*.PositivePromoter+')
```

will match any part whose (sub)type is either `NegativePromoter` or `PositivePromoter`.

```
PartSelector('*.*(TranscriptionFactor+)') & PartSelector('*.*(ReporterProtein+))
```

will match any part whose molecule is both a TranscriptionFactor and a ReporterProtein.

```
PartSelector('*.RBS+') % PartSelector('*.CodingRegion+')
```

will match any CodingRegion, which has an RBS part immediately before it.

Using these operators, complex Point Cut expressions can be built. All operators are left-associative, so there is no clear precedence between operators. Rather, precedence should be explicitly defined using parentheses.

Additionally, parts of a formula can be inverted using a `PointCutExpressionNot()` node, which does not have an overloaded Python operator:

```
PointCutExpressionNot('*.Terminator+')
```

will match any Part not matched by `'*.Terminator+'`

1.7 Advice

Advice is the code woven into the execution flow. An `Advice` consists of a PointCut, to select the location of where the code should be inserted, and a method of the Aspect.

An advice method belongs to the Aspect, and can access and store attributes via the `self` parameter. An advice method must also accept a second Parameter called `context`:

```
def adviceMethod(self,context):
    # self.addPart() could be used here
    pass
```

context is of type PointCutContext, which has a PointCutContext.part attribute, referencing the part that was matched by the PointCut of the advice.

Additionally, context has an isWithin() method, which can be used to search for Circuits or Aspects on the advice call stack. For example, this can be used to make sure that an advice is not recursively executing on itself.

1.7.1 Example: Print Advice Stack

Based on the previous Design Rules Example we can quickly add an aspect to print a stack trace:

```
from aosb import *

class CodingGFP(Circuit):
    def mainCircuit(self):
        declareNewMolecule('GFP')
        self.addPart(Promoter)
        self.addPart(CodingRegion(GFP))

class DesignRules(Aspect):
    def mainAspect(self):
        anyCodingRegion = PartSignature('*.CodingRegion+')

        beforeCodingRegion = PointCut(anyCodingRegion,PointCut.BEFORE)
        afterCodingRegion = PointCut(anyCodingRegion,PointCut.AFTER)

        self.addAdvice(beforeCodingRegion,self.insertRBS)
        self.addAdvice(afterCodingRegion,self.insertTerminator)

    def insertRBS(self,context):
        self.addPart(RBS)

    def insertTerminator(self,context):
        self.addPart(Terminator)

class PrintStack(Aspect):
    def mainAspect(self):
        self.addAdvice(PointCut('*.*',PointCut.AFTER),self.printStack)

    def printStack(self,context):
        print ','.join(str(context.within[i]) for i in range(len(context.within))) + " add: " + str(c


compiledDesign = Weaver(CodingGFP,DesignRules,PrintStack).output()
print compiledDesign
```

This will give the following output:

```
CodingGFP add: Promoter
CodingGFP,<__main__.DesignRules object at 0x7f8f42987a50> add: RBS
CodingGFP,<__main__.DesignRules object at 0x7f8f42987a50> add: Terminator
CodingGFP add: CodingRegion(codesFor = GFP)
Promoter+RBS+CodingRegion(codesFor = GFP)+Terminator
```

CodingGFP has a nicer string representation, which it inherits from Part (see Composites).

1.7.2 before and after Point Cuts

We have already encountered the simpler Before and After Point Cuts. They inject the advice code immediately before or after the join point selected by the Point Cut, as would be expected.

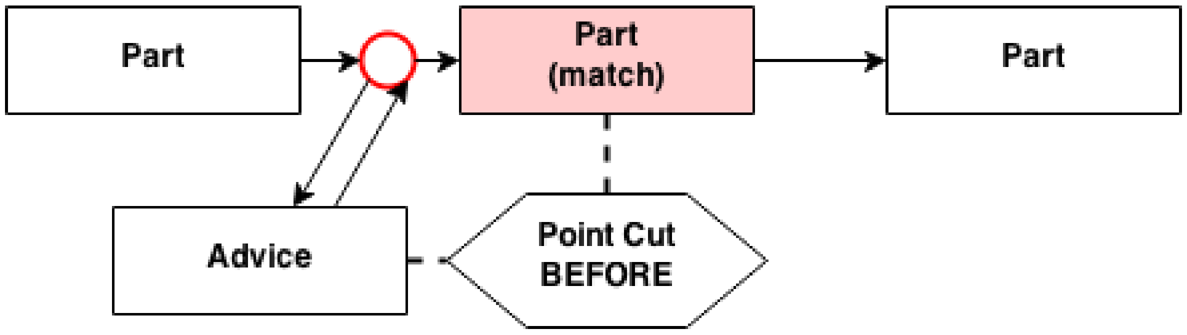

Figure 1.2: Advice weaving at a join point selected by a BEFORE Point Cut

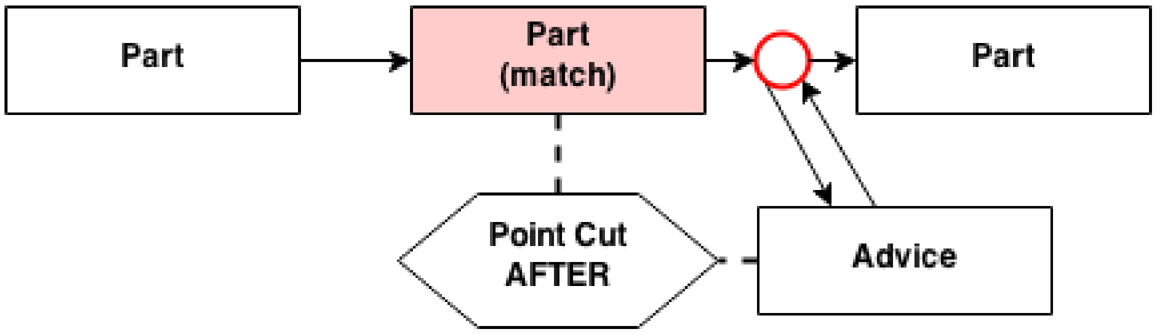

Figure 1.3: Advice weaving at a join point selected by an AFTER Point Cut

1.7.3 replace Point Cuts

1.7.4 Precedence

What if more than one Point Cut matches a given Join Point? Unless a precedence is explicitly set, the behavior is undefined.

`addAdvice()` takes an optional third parameter - an integer between `Advice.MINPRECEDENCE` *(0)* and `Advice.MAXPRECEDENCE` *(100)*

Advice which are added without specifying a precedence, are given minimal precedence by default. Higher precedence Advice is are deemed more important than those of low precedence.

For two Advice sharing a BEFORE Point Cut, this means that the high precedence advice will execute **before** the low precedence one. If both advice have the same precedence, their order is undefined.

For two Advice sharing an AFTER Point Cut, the high precedence advice will execute **after** the low precedence one. If both advice have the same precedence, their order is undefined.

For two Advice sharing a REPLACEMENT Point Cut, the high precedence advice will execute, whereas any lower precedence advice will be ignored.

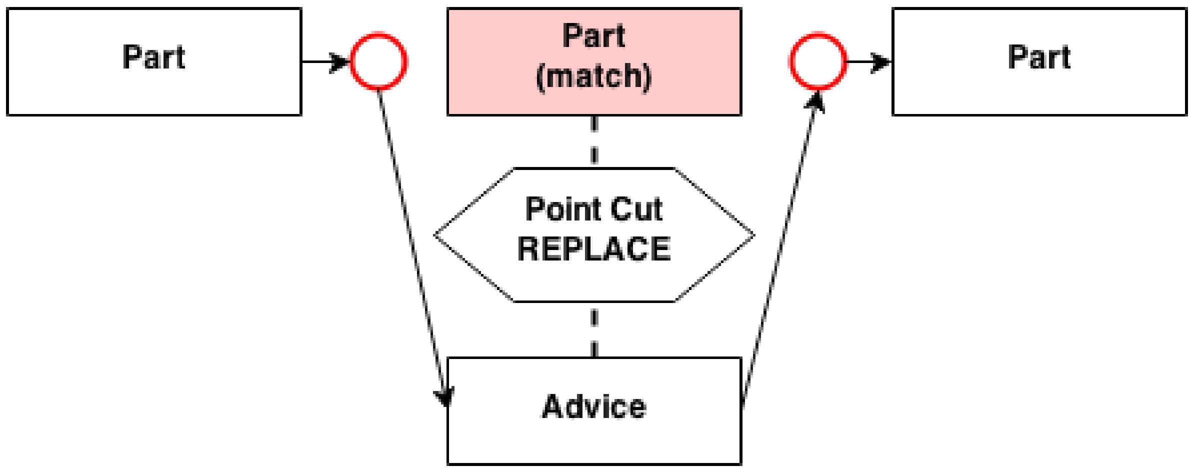

Figure 1.4: Advice on a replace Point Cut is executed instead of the Part.

If many Point Cuts match a Join Point, first the before advice will execute starting from the highest precedence. After the last before advice, the part will be added. Then the lowest precedence after advice will execute until the highest after advice. If a replacement advice occurs at some point, any lower before or after advice will be ignored. However, all after advice of similar or higher precedence will still execute.

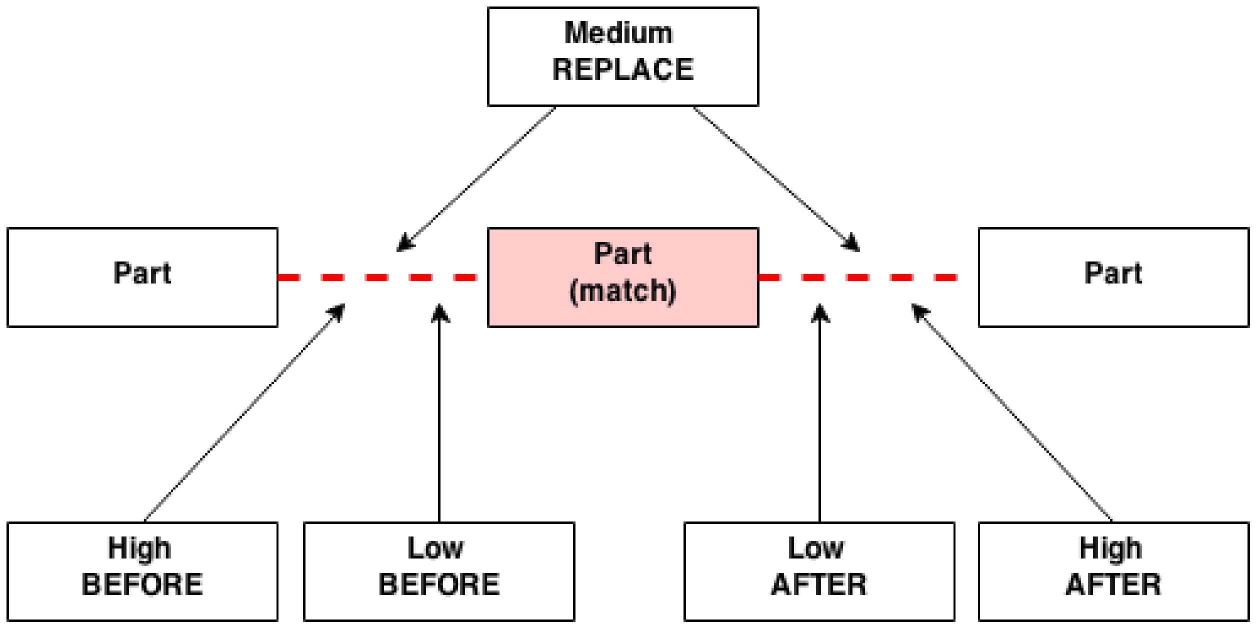

Figure 1.5: In this example, the low advice and the matched Part will not be executed, as they have been replaced by the medium Advice

1.8 Type Advice

Apart from normal Advice, which changes the execution flow, it is also possible to modify the Molecule and Part types using type advice. For example, one Aspect could add sequence information to a whole range of Parts; another could add modelling rules to Parts based on features selected by a `PartSignature`.

Analogously to `addAdvice()`, type advice can be added with

1.8.1 Adding type advice to Parts

Analogously to adding new Advice with `addAdvice()`, Type Advice can be added via `addTypeAdvice()`:

```
def mainAspect(self):
    self.addTypeAdvice(PartSignature('*.*'),True,"isPart")
```

`addTypeAdvice` requires three parameters: A `PartSignature` to select the Part, an attribute to add to the Part type, and a name under which this Part should appear.

If the attribute is a method, it is given two parameters: `self`, referring to the Aspect, and `part`, referring to the Part object:

```
class AnAspect(Aspect):
    def mainAspect(self):
        self.addTypeAdvice(PartSignature('*.*'),self.typeAdviceMethod,"sayHello")

    def typeAdviceMethod(self,part)
        print "Hello from "+str(part)
```

Type Advice can also be added to `Molecules` using the same `addTypeAdvice()` method. This requires changing the first parameter to a `MoleculeSignature`:

```
def mainAspect(self):
    self.addTypeAdvice(MoleculeSignature('TranscriptionFactor+'),True,"isTranscriptionFactor")
```

The `MoleculeSignature` is simply the element of the PartSignature within the parantheses.

1.8.2 Type Advice Example

Here is a simple example to add a method to Promoter types to print if they are regulated or not.

```
from aosb import *

class TwoPromoters(Circuit):
    def mainCircuit(self):
        self.addPart(Promoter)
        self.addPart(NegativePromoter(Protein))

class PromoterTypeAdvice(Aspect):
    def mainAspect(self):
        # define three signatures
        nonRegulatedPromoterSignature = PartSignature('*.Promoter+()')
        regulatedPromoterSignature = PartSignature('*.Promoter+(Molecule+)')
        anyPromoterSignature = PartSignature('*.Promoter+')

        # add Boolean Attribute "isRegulated" to two different types of Promoters
        self.addTypeAdvice(nonRegulatedPromoterSignature,False,"isRegulated")
        self.addTypeAdvice(regulatedPromoterSignature,True,"isRegulated")

        # add a type advice method
        self.addTypeAdvice(anyPromoterSignature,self.printIsRegulated,"printIsRegulated")

    def printIsRegulated(self,part):
        if part.isRegulated == False:
            print str(part)+" is not regulated by a Molecule."
        else:
            print str(part)+" is regulated by a Molecule."

compiledDesign = Weaver(TwoPromoters,PromoterTypeAdvice).output()
for part in compiledDesign.partList:
    part.printIsRegulated()
```

The output will be:

```
Promoter is not regulated by a Molecule.
NegativePromoter(regulatedBy = Protein) is regulated by a Molecule.
```

1.9 Weaver Output Advice

The last item to cover is the `WeaverOutput` class, a container object holding the results of the Weaver compilation and returned by `Weaver.output()`:

```
compiledDesign = Weaver(ACircuit,AnAspect,AnotherAspect).output()
```

By default, `WeaverOutput` has two attributes. `partList`, holding the list of the parts in order of execution, and `moleculeList` a list of all types of Molecules used in the system. The class also overloads `__str__` to provide a nice string output of the compiles design, which we have used in previous examples.

An Aspect can also declare output advice to the `WeaverOutput`. For example, this allows an Aspect implementing a rule-based modelling concern to define additional `WeaverOutput` methods, such as `printModel()`.

The mechanism is very similar to Type Advice:

```
def mainAspect(self):
    addWeaverOutput(self.newOutputMethod)

def newOutputMethod(self,weaverOutput):
    pass
```

`addWeaverOutput()` has only one parameter which must refer to a method of the Aspect. This weaver output advice method must accept two parameters, `self` to refer to the Aspect, and `weaverOutput` to refer to the `WeaverOutput` object. Unlike `addTypeAdvice()`, `addWeaverOutput` requires no name for the new attribute. Instead, the new method for `WeaverOutput` will automatically have the same name.

1.9.1 Example: Print number of parts

```
from aosb import *

class CodingGFP(Circuit):
    def mainCircuit(self):
        declareNewMolecule('GFP')
        self.addPart(Promoter)
        self.addPart(CodingRegion(GFP))

class DesignRules(Aspect):
    def mainAspect(self):
        anyCodingRegion = PartSignature('*.CodingRegion+')

        beforeCodingRegion = PointCut(anyCodingRegion,PointCut.BEFORE)
        afterCodingRegion = PointCut(anyCodingRegion,PointCut.AFTER)

        self.addAdvice(beforeCodingRegion,self.insertRBS)
        self.addAdvice(afterCodingRegion,self.insertTerminator)

    def insertRBS(self,context):
        self.addPart(RBS)

    def insertTerminator(self,context):
        self.addPart(Terminator)

class NumberOfPartsAspect(Aspect):
    def mainAspect(self):
        self.addWeaverOutput(self.printNumberOfParts)

    def printNumberOfParts(self,weaverOutput):
        print "The design has "+str(len(weaverOutput.partList))+" parts."

compiledDesign = Weaver(CodingGFP,DesignRules,NumberOfPartsAspect).output()
compiledDesign.printNumberOfParts()
```

CHAPTER

TWO

AOSB REFERENCE

exception `aosb.core.`**`AspectValueError`**

Bases: `exceptions.ValueError`

An Aspect class was expected, but something else given

`args`

`message`

exception `aosb.core.`**`CircuitValueError`**

Bases: `exceptions.ValueError`

A Circuit class was expected, but something else given

`args`

`message`

exception `aosb.core.`**`InvalidAdviceMethodError`**

Bases: `exceptions.ValueError`

method with 2 parameters expected, but something else given

`args`

`message`

exception `aosb.core.`**`InvalidPointCutError`**

Bases: `exceptions.ValueError`

object of type PointCut expected, but something else given

`args`

`message`

exception `aosb.core.`**`InvalidPointCutExpressionError`**

Bases: `exceptions.ValueError`

There is an error in a Point Cut expression

`args`

`message`

exception `aosb.core.`**`InvalidPointCutOperatorError`**

Bases: `exceptions.ValueError`

An unknown or illegal operator was used for the point cut

`args`

`message`

exception `aosb.core.`**`InvalidSignatureError`**

Bases: `exceptions.ValueError`

A signature (PointCut / Part / Molecule) is incorrectly formatted or typed

`args`

`message`

exception `aosb.core.`**`InvalidSymbolNameError`**

Bases: `exceptions.ValueError`

A symbol name (string) was not correctly formatted

`args`

`message`

exception `aosb.core.`**`InvalidWeaverOutputMethodError`**

Bases: `exceptions.ValueError`

with 2 parameters expected, but something else given

`args`

`message`

exception `aosb.core.`**`MoleculeValueError`**

Bases: `exceptions.ValueError`

Molecule was expected, but something else was given

`args`

`message`

exception `aosb.core.`**`PartInitializationError`**

Bases: `exceptions.Exception`

Part was expected, but something else was given

`args`

`message`

exception `aosb.core.`**`PartValueError`**

Bases: `exceptions.ValueError`

Part was expected, but something else was given

`args`

`message`

exception `aosb.core.`**`PrecedenceOutOfRangeError`**

Bases: `exceptions.ValueError`

An unknown or illegal operator was used for the point cut

`args`

`message`

exception `aosb.core.`**`SymbolExistsWarning`**

Bases: `exceptions.UserWarning`

A Part or Molecule declaration uses an existing name

args

message

class `aosb.core.`**`Advice`**(*pointcut*, *adviceMethod*, *precedence=0*)

Bases: `object`

Container for Advice

Attributes:

precedence : int - High precedence advice have execution priority over low precedence
- MINPRECEDENCE <= precedence <= MAXPRECEDENCE is the valid range

pointcut : PointCut

adviceMethod : method - The method to be executed at the advice
- must have 2 parameters: self and PointCutContext

MAXPRECEDENCE = 100

MINPRECEDENCE = 0

class `aosb.core.`**`Aspect`**

Bases: `object`

Abstract class for an Aspect | Will generally be used to represent a design's cross-cutting concerns

Notes:

Any child must implement mainAspect method

Attributes

weaver: Weaver
- The weaver which compiles the aspect

adviceList - A list of all advice this aspect declares

typeAdviceList - A list of all type advice this aspect declares

weaverOutputList - A list of all type advice for the WeaverOutput

addAdvice(*pointcut*, *adviceMethod*, *precedence=0*)

Declare a new advice in the aspect

Args:

pointcut - The pointcut for the advice

adviceMethod - The method to be executed at the pointcut
- should be a method bound to this aspect, with second parameter
- expecting a PointCutContext object

precedence : integer (optional) - set precedence of advice,
- see notes on Advice.precedence

Notes:

addAdvice constructs an Advice object. Further information thus
can be found there.

addPart (*part*)
Used to add parts to the design by passing them on to the AOSB Weaver

Args:
part: The part to be added to the circuit

addTypeAdvice (*signature*, *typeaddition*, *name*)
Declare a type advice in the aspect

Args:
signature - The signature for the type advice
typeaddition - The attribute / method to be added to the type
name: str - The name for the typeaddition in the new type

Notes:
addTypeAdvice constructs an TypeAdvice object.
Further information thus can be found there.

addWeaverOutput (*outputmethod*)
Declare a new weaver output target

Args:
outputMethod - The method to be added to the WeaverOutput

Raises:
InvalidWeaverOutputMethodError -
If outputmethod is not a method or has wrong number of parameters
(needs to accept self and a WeaverOutput reference)

getAdviceList ()
Returns adviceList

getTypeAdviceList ()
Returns typeAdviceList

getWeaverOutputList ()
Returns weaverOutputList

mainAspect ()
Entry point for an aspect, analogous to “main” in a program | Needs to be implemented by any sub class. | mainAspect will be called by the AOSB Weaver.

setWeaver(*weaver*)
Internal - Should not be used outside of the framework. Sets the Weaver Object of this Circuit

Args:
weaver: A weaver object that will be used

class aosb.core.**Circuit**
Bases: aosb.core.Part

Abstract class for a genetic part circuit.

Will generally be used to represent a design's core concerns
Additionally, a circuit can be used to represent composite parts

Attributes:
weaver: Weaver
The weaver which compiles the circuit

addPart(*part*)
Used to add parts to the circuit by passing them on to the AOSB Weaver

Args:
part: The part to be added to the circuit

mainCircuit()
Entry point for a circuit, analogous to "main" in a program

Needs to be implemented by any sub class.
mainCircuit will be called by the AOSB Weaver.

setWeaver(*weaver*)
Internal - Should not be used outside of the framework.

Sets the Weaver Object of this Circuit

Args:
weaver: A weaver object that will be used

after = None

before = None

moleculeConnection = None

namespace = None

weaver = None

class aosb.core.**CodingRegion**(*codesFor*)

Bases: aosb.core.Part

Class for parts that are Coding Regions

getCodesFor()

after = None

before = None

moleculeConnection = None

namespace = None

class aosb.core.**ConstitutivePromoter**

Bases: aosb.core.Promoter

Class for Promoters that are Constitutive Promoters

after = None

before = None

moleculeConnection = None

namespace = None

class aosb.core.**Molecule**

Bases: object

Superclass for all molecules.

All molecules are used as classes, not objects. Any agents such as Proteins, Transcription Factors, etc. need to be subtypes of Molecules

class aosb.core.**MoleculeSignature**(*signature*)

Bases: aosb.core.PartSignatureElement

A PartSignatureElement that is a Molecule Signature | Overloads some methods, since MoleculeSignature is used by the user for | Molecule Type Advice - unlike PartSignatureElement, which is internal

match(*obj*)

see PointCutExpressionNode definition

ANY = 1

CLASSONLY = 3

SUBCLASS = 2

class aosb.core.**NegativePromoter**(*regulatedBy*)

Bases: aosb.core.Promoter

Class for Promoters that are Negative Promoters

getRegulatedBy()

after = None

before = None

moleculeConnection = None

namespace = None

class `aosb.core.`**`Part`**

Bases: `object`

Abstract Superclass for all Parts

Parts are the atomistic instructions in the genetic circuit execution flow

Attributes:

before : Part
: The part that immediately precedes this one in the execution flow

after : Part
: The part that immediately follows this one in the execution flow

moleculeConnection : class(Molecule)
: Each part may have a logical link to a molecule

namespace : Circuit or Aspect
: Each part belongs to a namespace, i.e. the circuit or aspect that has added it to the execution flow

`after` **= None**

`before` **= None**

`moleculeConnection` **= None**

`namespace` **= None**

class `aosb.core.`**`PartSignature`**(*signature*)

Bases: `aosb.core.PointCutExpressionNode`

A PartSignature, used in PointCut expressions or directly for Type Advice

Attributes:

namespace : PartSignatureElement - The namespace part of the signature

part : PartSignatureElement - The "part" part of the signature

molcule : PartSignatureElement - The molecule part of the signature

nomolecule : Boolean -
: If the PartSignature explicitly should not match parts with molecules

`match`(*part*)
: see PointCutExpressionNode definition

`numberOfMatchingParts`(*part*)
: returns the number of parts the expression matches | If an expression uses concatenation, then it might match the current part and a number of preceding parts.

Args:

part - The part at which matching starts

Returns:

integer - how many parts were matched

class `aosb.core.`**`PartSignatureElement`**(*signature*)

Bases: `object`

A building block of a Part Signature

Attributes:

qualifier : ANY / SUBCLASS / CLASSONLY
    Whether the element should match precisely, all subclasses or uses a wildcard
element : str - The string of the element (without a qualifier)
inverse : boolean - Whether the element has been negated

`match`(*obj*)

see PointCutExpressionNode definition

`ANY` **= 1**

`CLASSONLY` **= 3**

`SUBCLASS` **= 2**

class `aosb.core.`**`PointCut`**(*signature*, *operator*)

Bases: `object`

A PointCut to select Join Points in the genetic parts execution flow

Attributes:

operator : BEFORE / AFTER / REPLACE
    - The operator for this PointCut
signature : PartSignature

`checkAndSetOperator`(*operator*)

Set the internal operator attribute, if the parameter is a valid operator

Args:

operator - The operator to be confirmed

Raises:

InvalidPointCutOperatorError - if the operator is invalid
    (can be dependent on the signature)

`match`(*partObject*)

see PointCutExpressionNode definition

`AFTER` **= 22**

`BEFORE` **= 11**

REPLACE = 33

class aosb.core.**PointCutContext** (*within*, *part*)

Bases: `object`

Container for context at a PointCut

Attributes:

within - A stack of the circuits / aspects that within which the PointCut was matched

part - The part matched by the PointCut

isWithin (*obj*)

Checks if the parameter (a circuit or aspect) is on the within stack | *Args:* | obj : Circuit / Aspect - The object that is to be found on the stack

Returns:

Boolean - true if obj is on the within stack

class aosb.core.**PointCutExpressionAnd** (*left*, *right*)

Bases: `aosb.core.PointCutExpressionOperator`

A PointCutExpressionOperator which is an And

expressionUses (*nodeType*)

Confirms if a certain type of node is used in the expression

Args:

nodeType: The type of the node whose existence is to be confirmed

Returns:

boolean - Whether or not the node exists in the formula

match (*part*)

see PointCutExpressionNode definition

numberOfMatchingParts (*part*)

left = None

right = None

class aosb.core.**PointCutExpressionConcatenate** (*left*, *right*)

Bases: `aosb.core.PointCutExpressionOperator`

A PointCutExpressionOperator which is an Concatenation

expressionUses (*nodeType*)

Confirms if a certain type of node is used in the expression

Args:

nodeType: The type of the node whose existence is to be confirmed

Returns:

boolean - Whether or not the node exists in the formula

`match` (*part*)

see PointCutExpressionNode definition

`numberOfMatchingParts` (*part*)

see PointCutExpressionNode definition

This child overrides it, since a concatenation operator is a node in the expression at which more than one part can be matched.

`left` = None

`right` = None

class `aosb.core.`**`PointCutExpressionNode`**

Bases: `object`

Abstract superclass for all Nodes in a Point Cut Expression Tree

A complex expression for a Point Cut, using operators such as & (and),

| (or) or % (concatenation) is represented as a tree of nodes

`match` (*part*)

Does a part match this (sub)-expression? An abstract method, must be implemented by each child

Args:

part - The part to be matched

Returns:

boolean - Whether or not the part was matched

`numberOfMatchingParts` (*part*)

returns the number of parts the expression matches | If an expression uses concatenation, then it might match the current part and a number of preceding parts.

Args:

part - The part at which matching starts

Returns:

integer - how many parts were matched

class `aosb.core.`**`PointCutExpressionNot`**(*pointcutexpression*)

Bases: `aosb.core.PointCutExpressionOperator`

A PointCutExpressionOperator which is a Not

A special case, only uses one child, acts as the inverse operator

`expressionUses`(*nodeType*)

Confirms if a certain type of node is used in the expression

Args:

nodeType: The type of the node whose existence is to be confirmed

Returns:

boolean - Whether or not the node exists in the formula

`match`(*part*)

see PointCutExpressionNode definition

`numberOfMatchingParts`(*part*)

returns the number of parts the expression matches | If an expression uses concatenation, then it might match the current part and a number of preceding parts.

Args:

part - The part at which matching starts

Returns:

integer - how many parts were matched

`left` **= None**

`right` **= None**

class `aosb.core.`**`PointCutExpressionOperator`**(*left*, *right*)

Bases: `aosb.core.PointCutExpressionNode`

Abstract superclass of an PointCutExpression Node which is an operator

Attributes:

left - The first child of the operator

right - The second child of the operator

`expressionUses`(*nodeType*)

Confirms if a certain type of node is used in the expression

Args:

nodeType: The type of the node whose existence is to be confirmed

Returns:

boolean - Whether or not the node exists in the formula

match(*part*)

Does a part match this (sub)-expression? An abstract method, must be implemented by each child

Args:

part - The part to be matched

Returns:

boolean - Whether or not the part was matched

numberOfMatchingParts(*part*)

returns the number of parts the expression matches | If an expression uses concatenation, then it might match the current part and a number of preceding parts.

Args:

part - The part at which matching starts

Returns:

integer - how many parts were matched

left = None

right = None

class aosb.core.**PointCutExpressionOr**(*left*, *right*)

Bases: aosb.core.PointCutExpressionOperator

A PointCutExpressionOperator which is an Or

expressionUses(*nodeType*)

Confirms if a certain type of node is used in the expression

Args:

nodeType: The type of the node whose existence is to be confirmed

Returns:

boolean - Whether or not the node exists in the formula

match(*part*)

see PointCutExpressionNode definition

numberOfMatchingParts(*part*)

left = None

right = None

class aosb.core.**PositivePromoter**(*regulatedBy*)

Bases: aosb.core.Promoter

Class for Promoters that are Positive Promoters

getRegulatedBy()

after = None

before = None

moleculeConnection = None

namespace = None

class aosb.core.**Promoter**

Bases: aosb.core.Part

Class for Parts that are Promoters

after = None

before = None

moleculeConnection = None

namespace = None

class aosb.core.**Protein**

Bases: aosb.core.Molecule

Class for Molecules that are Proteins

class aosb.core.**RBS**

Bases: aosb.core.Part

Class for parts that are Ribosome BindingSites

after = None

before = None

moleculeConnection = None

namespace = None

class aosb.core.**Terminator**

Bases: aosb.core.Part

Class for parts that are Terminators

after = None

before = None

moleculeConnection = None

namespace = None

class aosb.core.**TypeAdvice**(*signature*, *typeaddition*, *name*, *aspect*)

Bases: object

Container for TypeAdvice

Attributes:

signature : PartSignature or MoleculeSignature
typeaddition : method or attribute to be added to the type
name : The name the new typeaddition should have in the new type
aspect : Aspect which declares this TypeAdvice

`isMoleculeAdvice``()`
Returns True if TypeAdvice is for Molecule, False otherwise

`isPartAdvice``()`
Returns True if TypeAdvice is for Part, False otherwise

class `aosb.core.`**`Weaver`**`(`*circuit*, **aspects*`)`
Bases: `object`

The "compiler" that weaves core concerns (circuits) and cross-cutting concerns (aspects) and creates a woven execution flow of parts

Attributes:
partList - The current list of parts
moleculeList - The current list of molecules
beforeAndReplaceAdviceList - List of all before and replace advice to be woven
afterAdviceList - List of all after advice to be woven
partTypeAdviceList - List of all part type advice
moleculeTypeAdviceList - List of all molecule type advice
circuit - The main circuit
aspects - The list of all aspects to be woven
weaverOutput : WeaverOutput - The "compiled" result

class **`WeaverOutput`**
Bases: `object`

Container for the woven result of the Weaver

Attributes:
partList - finished ordered list of parts in the design
moleculeList - list of all molecules in the design

`Weaver.`**`addPart`**`(`*callingObject*, *part*`)`
Called by circuit or aspect to add a part in the execution flow

Args:
callingObject - the circuit or aspect calling this
part - The part supposed to be added

`Weaver.`**`addPartTypeAdvice`**`(`*part*`)`
Internal - adds type advice to a part

`Weaver.`**`constructMoleculeListAndAddTypeAdvice`**`()`
Internal - constructs list of Molecules in design and adds their type advice

`Weaver.`**`output`**`()`
Returns the weaverOutput Element

`Weaver.`**`readAspectsConstructAdviceLists`**`()`
Internal - initializes aspects and constructs all advice lists

`Weaver.`**`runAfterAdvice`**`(`*callingObject*, *part*, *precedence=0*`)`
Internal - runs After Advice of a part

Args:

part - The part for which the advice matches
callingObject - the circuit or advice which added the part
precedence - Only run advice with a precedence greater or equal this
Used if replacement advice has been executed

`Weaver.`**`runBeforeAndReplaceAdvice`**`(`*callingObject*, *part*`)`
Internal - runs Before and Replace Advice of a part

Args:

part - The part for which the advice matches
callingObject - the circuit or advice which added the part

Returns:

"continue"
True - If part should still be added
False - If part has been replaced and remaining after advice executed

`Weaver.`**`sortAdviceList`**`()`
Internal - sorts the advice lists by precedence

`aosb.core.`**`checkAndSetMolecule`**`(`*molecule*`)`
checks if parameter is a class of type Molecule and returns it

Args:

molecule - A potential Molecule

Raises:

MoleculeValue - If molecule is not a class of type Molecule

Returns:

A class of type Molecule

`aosb.core.`**`checkBaseClassesMatch`**`(`*bases*, *typename*`)`

Recursively check if the name of the types in bases (or parents) are equal to typename

Args:

bases - A tuple of types

typename : str - A name of a type

Returns:

True if any of the names of types in bases or any of their parent bases equals typename,
False otherwise

`aosb.core.`**`checkIfTypeReturnInstance`**`(`*possibleType*`)`

if the parameter is a type, try to return an instance

Args:

possibleType - a parameter which may be an instance or a type

Raises:

PartInitializationError - if the parameter is a type which can not be constructed

Returns:

An instance of the type of possibleType

`aosb.core.`**`declareNewMolecule`**`(`*classname*, **parents*`)`

Returns a new Molecule type and exports it to the caller's namespace

Args:

classname: The name for the new type

*parents: 0 or more Molecule super classes

Returns:

The new Part type

Raises:

MoleculeValueError: If any parent is not a Molecule

InvalidSymbolNameError: If classname is not a valid name for a symbol

Warnings:

SymbolExistsWarning: If classname already exists in the namespace `aosb.core.`**`declareNewPart`**(*classname*, *parent=<class 'aosb.core.Part'>*, *moleculeConnection=None*)

Returns a new Part type and exports it to the caller's namespace

Args

classname : string

The name for the new type

parent : Part

super class for the new type

moleculeConnection : Molecule

optional, if the new Part type should be connected to a Molecule type

Returns

The new Part type

Raises

PartValueError -If the parent is not a Part

InvalidSymbolNameError - If classname is not a valid name for a symbol

Warnings

SymbolExistsWarning - If classname already exists in the namespace

PYTHON MODULE INDEX

INDEX