Development of Composition Annotation System in GRITS Toolbox

Author: Masaaki Matsubara

Last update: 12/6/2019

## Purpose

To use glycan compositions for MS annotations.

## Development Plans

* Extends GELATO annotation system
  + To use the same system as GELATO system
* Separates logic part and GUI part
* Extends GELATO plugin for the logic part
* Extends GELATO GUI plugin for the GUI part

## Related plugins

All related plugins are in our SVN system

* **org.grits.toolbox.ms.annotation.glycan.composition**
  + Main plugin for composition annotations (Logic part)
  + Extends GELATO plugin (see below)
  + This plugin will use as a .jar file.
  + Location: trunk/jar/
* **org.grits.toolbox.ms.annotation**
  + GELATO plugin
  + Refactored some MS type related classes to simplify to extend
  + Location: branches/gelatorefactoring/jar/
    - Original location: trunk/jar/
* **org.grits.toolbox.lib**
  + Library plugin containing GELATO
  + Composition plugin will be added
  + Location: trunk/jar/
    - Original location: trunk/jar/
* **org.grits.toolbox.entry.ms.annotation.glycan.composition**
  + Main plugin for composition annotations (GUI part)
  + Extends some classes in GELATO GUI plugin (see below)
    - To start annotation
  + Extends some classes in table viewer plugin (see below)
    - To show annotation results
  + Location: **not created yet**
* **org.grits.toolbox.importer.ms.annotation.glycan.simiansearch**
  + GELATO GUI plugin
  + Modified to use refactored GELATO plugin
  + Location: branches/gelatorefactoring/eclipse/
    - Original location: trunk/eclipse/
* **org.grits.toolbox.entry.ms.annotation.glycan**
  + Table viewer plugin for glycan
  + Location: trunk/eclipse/

## Extending GELATO Annotation System

### Refactoring GELATO plugin to separate MS type specific part

Originally, the main annotation part in GELATO was including the MS type specific part, and thereby it needed to extend to 4 classes to handle each MS type. If we extend each of them, we need to implement the same thing 4 times. Thus, I refactored GELATO plugin to separate the main process into main annotation part and MS type management part.

The refactoring was followed these steps:

1. Refactored logic plugin (org.grits.toolbox.ms.annotation)
   1. Extracted of MS management methods from main annotation class
   2. Created an abstract class only for MS management
   3. Made the abstract class a part of the main annotation class
   4. Extended the abstract class for each MS type
   5. Instantiated the extended class based on MS type and pass it to main annotation class.
2. Compiled logic plugin and putted it into the lib plugin (org.grits.toolbox.lib)
3. Updated GUI plugin (org.grits.toolbox.importer.ms.annotation.glycan.simiansearch) for handling updated logic plugin

These refactorings are already finished and submitted as a branch of SVN system (branches/gelatorefactoring/) and all the tests has been done. These need to be merged into main branch.

### Implementation of Logic Part

The main annotation system is implemented by extending GELATO system.

The difference between our composition system and GELATO is mainly whether GlycoWorkbench is used or not. While GELATO uses GlycoWorkbench at many parts such as glycan object models, mass calculation and glycan fragmentation, our system uses own object models and systems for handling glycan composition and its mass calculation for some reasons (see *Implementations of Composition Object Models* section)

While the most of this composition annotation processes are the same as the GELATO, this system also extends the annotation system in GELATO to use own object models, mass calculation and fragmentation (not to use GlycoWorkbench).

These implementations are in the org.grits.ms.annotation.glycan.composition.annotation package.

* Main class: CompositionStructureAnnotation

*This implementation is almost done but it still need to test at some point.*

### Implementation of GUI Part

*The GUI part is not implemented yet at all.*

#### ToDo List

For future developers.

* Extend GELATO GUI plugin (for starting annotation)
  + Extend NewGeratoHandler.getNewAnalyteStructureAnnotation() method to use CompositionStructureAnnotation class instead of GlycanStructureAnnotation class
  + Create a wizard page for choosing maximum and minimum number of residues for the candidate composition structures.
    - The list of residues can be loaded from ResidueDictionary.
    - We need to create a wizard containing this page instead of the wizard page for choosing glycan databases.
* Extend table viewer plugin (for showing annotation results)
  + Create or extend image provider for composition
    - To support glycan composition image
    - GlycoWorkbench and GlycanBuilder extension can not be used because the sequence format of the composition is totally original (different from GWB sequence).
    - We can use a utility method in GlycanBuilder extension to generate an image of styled text.
      * This sequence format for composition needs to be converted to the styled text format if using the utility method.
  + Extend classes related to the image provider

## Implementations of Composition Object Models

The main logic plugin uses own object models and systems for handling glycan compositions, and its mass calculation and fragmentation.

Originally, we were going to use GlycoWorkbench system for that because GELATO is using it. However, we need to solve following problems if we use it:

* Some of the glycan residues are not contained in GlycoWorkbench system
  + We need to update GlycoWorkbench to add the new residues
* GlycoWorkbench can not handle the fragmentation of glycan composition
* It is not so easy to extend GlycoWorkbench functionalities because the implementation is quite tricky

So I decided to implement the systems from scratch.

### Implementation of Compositions and Fragments

* Location: under the org.grits.toolbox.ms.annotation.glycan.composition.structure package

#### Composition

A composition is composed of three residue types, Monosaccharides, Substituents, and Reducing ends and implemented as Composition class. The residue types must be loaded from the dictionary file (see Residue Dictionary section). Each of monosaccharide and substituent is stored as key and value, which the key is the residue and the value is its count. A composition can have only one reducing end. The residue type is implemented as a abstract class ResidueType, and MonosaccharideType and SubstituentType classes are subclasses of the ResidueType class. The substituent type and reducing end type are represented by the same class SubstituentType, but these can be distinguished with a member variable isReducingEnd. Basically, all residue types must be loaded from dictionary, but any residue which only have name and mass value can be created using CustomSubstituentType class. This class can be used for user-specific residue.

The composition class has a method computeMass() to calculate its mass value, the mass options can be set using setMassOptions(boolean, PerderivatizationType) method. The first boolean argument is whether the resulting mass is monoisotopic or avarage, and the second enum class PerderivatizationType specifies the type of perderivatization. The perderivatization type will be free end if null is specified and it is default.

#### Fragment of composition

A fragment of composition is composed of the same residue types as the composition and fragment types, and is implemented as CompositionFragment class. The fragment types are classified with 6 cleavage types, A, B, C, X, Y, and Z. Although A and X fragments have a part of the cleaved monosaccharide information because these are cleaved at middle of the ring of monosaccharides, B, C, Y, and Z fragments do not have the cleaved monosaccharide information. Although X, Y, and Z fragments are defined as leaf fragments, and stored as key and value, which the key is the fragment and the value is its count. A, B, and C fragments are defined as root fragment, and it can be only one in a fragment. A and X fragments are implemented as CrossRingFragmentType class and B, C, Y, and Z fragments are implemented as GlycosidicFragmentType class. The fragment types can only be generated from MonosaccharideType using FragmentDictionary. Only one instance for each fragment type is generated and provided by FragmentDictionary.

#### Class hieralchy

* Composition
  + CompositionFragment
* ResidueType
  + MonosaccharideType
    - CrossRingFragmentType (implements IFragmentType)
  + SubstituentType
    - CustomSubstituentType
  + GlycosidicFragmentType (implements IFragmentType)

#### Parser

The compositions and fragments can be exported as string representation using toString() method of Composition and CompositionFlagment. CompositionUtils class has a method parse(String) to parse the strings.

The string of composition is separated into two sections, residue list and a reducing end, and these sections are separated with two hyphens “--”. Each residue in the residue list is separated with a comma “,” each other, and is represented as “{name}:{count}”, which {name} is the name of residue type and {count} is then number of the residue type. The name of custom substituent type is represented as “{name}={mass value}”, which {name} is the name of the substituent and {mass value} is the mass value of the substituent.

The string of fragment contains the string of composition and the fragment residues. While the fragment residue is represented as the same format as the composition residue, the fragment residues must be separated with two hyphens “--” from the other residues. The name of fragment type is represented as “{name}#{cleavage type}”, which {name} is the name of cleaved monosaccharide and {cleavage type} is the name of cleavage type. If the cleavage type is cross ring cleavage, it must have start and end positions of cross ring cleavage, and is represented as “{A or X}\_{start position}\_{end position}”. The name of cleaved monosaccharide of B, C, Y, and Z fragments needs to be “Sugar” because it does not need information of the cleaved monosaccharide.

##### Format

* Composition: {residue list}--{reducing end}
  + {residue list}: {residue},{residue},...,{residue}
  + {reducing end}: {name}
  + {residue}: {name}:{count}
* Fragment:  
  {fragment residue list}--[{residue list}--]({reducing end} or {root fragment residue})
  + {fragment residue list}:  
     {fragment residue},{fragment residue},...,{fragment residue}
  + {fragment residue}: {name}#{cleavage type}:{count}
  + {root fragment residue}: {name}#{cleavage type}
  + {cleavage type}: A\_{start position}\_{end position}, B, C,  
     X\_{start position}\_{end position}, Y, or Z

##### Samples of string representation

* Hex:3,HexNAc:2
* Hex:3,HexNAc:2--redEnd
* Hex#X\_0\_2:1--Hex:2,HexNAc:1--HexNAc#A\_0\_4
* Sugar#Y:1,Sugar#Z:1--Hex:1,HexNAc:1--Sugar#B
* Hex#X\_0\_2:1,Sugar#Z:1--Sugar#C

### Residue Dictionary

For using the residues, we need to load dictionaries for monosaccharides, substituents, and reducing end. The default dictionaries are stored as configuration files named monosaccharid\_type, substituent\_type, and reducing\_end\_type, respectively, in src/conf/residue directory. A dictionary class named ResidueDictionary can load these files using loadDefaultDictionary() method. The class can also load user-specific data but the user must follow formats of our dictionaries.

#### Format

All items in a dictionary must be listed with the orders listed below and separated with tab (\t) each other. The same format is used for substituents and reducing ends but the files must be created individually.

* For monosaccharide

1. *Name*

* The name of the monosaccharide

1. *Synonyms*

* The synonyms of the monosaccharide.
* Multiple synonyms can be specified using a comma “,” as a separator.
* A hyphen “-” if no item.

1. *Carbon length*

* The length of carbon chain of the monosaccharide.
* The maximum carbon number is this value.

1. *Anomeric position*

* The position of anomeric carbon.

1. *Ring size*

* The number of atoms which make ring shape
* 0 if no ring.

1. *Core modifications*

* The core modifications on the monosaccharide.
* Multiple modifications can be specified using a comma “,” as a separator.
* Each modification must be represented as “{position}:{name}”, e.g. “2:d”.
* The unknown position can be specified using question mark “?” instead of the position number.
* Available modifications are the followings: “d” (deoxy), “a” (acid), “h” (hydroxy), “u” (double bond), “lac” (lactone), and “an” (anhydro).
* A hyphen “-” if no item.

1. *Substituents*

* The substituents on the monosaccharide.
* Multiple substituents can be specified using a comma “,” as a separator.
* Each substituent must be represented as “{position}:{name}”, e.g. “2:NAc”.
* The unknown position can be specified using question mark “?” instead of the position number.
* All substituents listed as item must be loaded into the substituent dictionary (see below) before load the monosaccharide dictionary.
* A hyphen “-” if no item.

1. *Description*

* The description of the monosaccharide
* For substituent and reducing end

1. *Name*

* The name of the residue

1. *Synonyms*

* The synonyms of the residue.
* Multiple synonyms can be set specified using a comma “,” as a separator.
* A hyphen “-” if no item.

1. *Molecular composition*

* A string representation of molecular composition.
* “?” if unknown.

1. *Number of methylation*

* The number of methyl groups which the residue have when permethylated.

1. *Droppable when permethylation*

* Yes or true if the residue is lost when permethylated. Otherwise, No or false.

1. *Number of peracetylation*

* The number of methyl groups which the residue have when peracetylated.

1. *Droppable when peracetylation*

* Yes or true if the residue is lost when peracetylated. Otherwise, No or false.

1. *Number of linkages*

* Total number of linkages on the residue.

1. *Is acidic*

* Yes or true if the residue have an acidic part.

1. *Description*

* The description of the residue.

### Mass Calculation (Atomic info)

I re-implemented some object models handling atoms and molecules, which is implemented in GlycoWorkbench, to calculate composition mass. Because the implementation of atoms and molecules is totally the same as GlycoWorkbench, the specification and format are omitted in this document. Please see document of GlycoWorkbench.

The molecular compositions of monosaccharide type and its mass are automatically calculated from its properties (carbon length, core modifications, and substituents), and stored when the monosaccharide type is instantiated.

The substituent type and reducing end type do not need to generate molecular composition because these originally have the molecular composition information.