# Libmoleculizer 1.1.3 User's Manual

Nathan Addy
The Molecular Sciences Institute

November 9, 2011

# Contents

# Chapter 1

# Introduction

## 1.1 Overview of libmoleculizer

Libmoleculizer is a computer library that takes a set of rules describing a collection of protein and small molecule types as well as a set of descriptions of interactions amongst those types, i.e. different types of protein-protein, protein-small molecule, and enzymatic reactions (protein modifications), and using that information explicitly generates a network of multi-protein species and their reactions that are described by those rules.

The ability to describe networks implicitly using rules can be be very advantageous for modeling biochemical networks. This is because for many common cases, the size of the explicit network grows combinatorially relative to the number of protein/molecule types and interactions types that network possesses. For instance, in a particular model of the alpha mating pathway in S. cerivisiae, defined with fewer than 20 molecule types and fewer than 40 interactions, the explicit network consists of over 350,000 species and over 1,000,000 reactions.

Libmoleculizer is a computer tool that helps manage the complexity of the absolute set of species and reactions in a reaction network, by expanding the typically much smaller and much more comprehensible network of biochemical interactions that users provide. Typically, Libmoleculizer will be used as a component of a larger simulator, which will use libmoleculizer for species and reaction management. For instance, a simulator using libmoleculizer could take in a set of rules, use libmoleculizer to expand the corresponding network, and then use the generated network to run an ODE or stochastic simulation.

Where libmoleculizer goes beyond other software packages of its type however, is in the generation of these networks. Biochemical networks do not have to be expanded all at once. This has two advantages: it allows libmoleculizer to be able to handle networks other packages cannot, such as unbounded species growth, and it also allows for faster operation. In a spatial simulation, if two species collide, the user will not care to know every species and reaction that can occur in the model, they will only want to know whether those two species

react or not. Using libmoleculizer, users do not have to computationally pay for the time to generate huge portions of the network they do not care about.

## 1.2 Manual Overview

**Disclaimer**

At the current moment, this manual is highly incomplete. We expect that in the next couple weeks, the next version of libmoleculizer will be released which will greatly flesh out this user-manual. For the current release, we have decided to include all, including in progress, documentation – however, certain aspects will be very, very roughly sketched out, to the point of being unreadable.

For the time being, chapters marked with a star are considered incomplete.

**Overview**

This manual aims to provide introduction to the functionality and use of the libmoleculizer library.

This manual has three major goals. The first of the manual is to describe what Moleculizer does; this is treated in chapters 2 and **??**. The second objective is to describe the MZR file format in which reaction rules are specified; this is given an overview in chapter 3, and more explicitly in chapter 8. Finally, the libmoleculizer API must be learned, in order to understand how to pass MZR files into libmoleculizer, expand the network, and read out the generated species and reactions. This topic is dealt with in chapters **??**, **??**, and 7.

Finally, although libmoleculizer is a written in standard C++ and C, and packaged using standard methods, we cannot guarantee you won't have problems installing it. Start by reading and following the instructions in the INSTALL file that came with libmoleculizer. There is a 99% chance that will work. However, if you do have any trouble, consult the installation instructions and troubleshooting help available in the chapter 5.

## 1.3 History

Libmoleculizer is the descendant of a prior program called Moleculizer, developed by Dr. Larry Lok at the Molecular Sciences Institute to study the alpha mating pathway in yeast. The standard modeling techniques at the time all involved explicitly enumerating a set of chemical species and reactions. However, it was determined for the alpha signal transduction pathway in yeast that there were so many species and reactions possible, that they could never be practically enumerated. The typical response to this problem had previously been to simplify the system, combining species and reactions, in order to produce a simpler but non-exact simulation of the system involved. This approach was non-optimal, as it required making fundamental assumptions about what was and was not important, prior to investigating system behavior.

The response was to develop a new simulator called Moleculizer[**?**] that took in a description of the basic monomeric proteins and small molecules, as well as a set of rules that described basic interactions between proteins. These rules were used within a simulator implementing Dan Gillespie's first reaction algorithm [**?**]. As it ran, Moleculizer would alternate between executing reaction events in the manner of Gillespie and using the rules to generate enough of the reaction network so that the the next reaction event would always be accurate.

Realizing the versatility of this new rule-based approach, a new software project was begun by Nathan Addy at the Molecular Sciences Institute, to adapt the reaction network generating components of Moleculizer into a generic library, called libmoleculizer, that could be used within a variety of simulators and in a variety of contexts.

This manual corresponds to version 1.1.3of that software.

# Chapter 2

# Useful Concepts in libmoleculizer

Hopefully you already have some idea of what the libmoleculizer program is: a computer system that models complicated biochemical pathways (such as signal transduction pathways), by representing chemical species as being composed of one or more indivisible molecule types bound together and reactions between species as being special cases of interactions between interactions between molecules.

This chapter gives an overview of the key concepts needed to understand and use the libmoleculizer system.

## 2.1 Moleculizer creates new species and reactions by expanding species

When Moleculizer starts up, it reads in a collection of rules that defines how a set of biochemical molecules interact; then adds a user specified set of complex-species to an initial list of known complex-species and reactions. Upon initialization, Moleculizer uses the rules to calculate the reactions that can occur between those initial complex-species, and adds the generated reactions to its list of known reactions. Additionally, it adds any products calculated to be generated as products of those reactions into the list of known species, as an *unexpanded* complex-species.

As it runs, Moleculizer has a list of species and reactions it knows about that make up the generated network. However, this generated network usually isn't complete, meaning that that Moleculizer can still generate additional species and reactions using the rules, but has not yet done so.

Each species generated by Moleculizer is either in a state of having been expanded or it has not yet been expanded. Expanded species are species that have been analysed by Moleculizer for matching the rules in the model. When

a species is expanded, its free binding sites are matched against the free binding sites on the already expanded species. If any of the new free binding sites pair up with any of the already expanded complexes, libmoleculizer takes the two species and adds them as the two substrates into a new association reaction. It forms the product by physically binding the two species together and examining the result. If the result is a structurally new species, moleculizer adds it as a new but unexpanded species to the list of species and reactions. Regardless, the reaction is added as a new reaction to its lists.

Transformation reactions work similarly. When a species is expanded, it is checked to see if it contains any of the sub-complexes needed for any of the transformation reactions. If it contains any of them, a new reaction is created, with the matching species as its substrate. The transformation is performed on that species to create the product species. If the product species has not been seen by moleculizer, it is added as a new but unexpanded species. Regardless, the reaction is added as a new reaction.

Thus the process of expanding the reaction network is a matter of taking an initial set of species (by default moleculizer starts with the set of species consisting of a monomeric molecule and any explicitly defined species) and iteratively expanding them one by one.

By expanding them, libmoleculizer checks them for interesting *"features"* that can participate in a reaction, as defined in the rules. If libmoleculizer finds they have features that match rules in the model, perhaps in combination with other, already expanded species, libmoleculizer generates a new reaction, with the newly expanded species as one of the substrates. Depending on circumstances, the product(products) of the reaction may also be new, in which case it is added as a new, unexpanded species to the network.

Expanding the reaction network fully is simply a matter of expanding unexpanded species until there is nothing left to expand.

## 2.2 Biochemical networks consist of chemical species and sets of chemical reactions between them

The libmoleculizer library takes a convenient description of a biochemical reaction network that includes descriptions of the network's constituent proteins as well as *rules* defining protein-protein and enzymatic interactions between those proteins. Libmoleculizer uses this description to generate all or part of the network, emitted as a list of variables (chemical species types) and equations (describing reactions between variables) which can be used in other simulation systems to simulate the network. Emitted Libmoleculizer reaction equations are compatible with all common simulation formalisms, including differential equations, Gillespie solvers, particle-based simulators.

Importantly, libmoleculizer is organized as a library, which means that the ability to convert a human-readable rule-based model into the more conventional species/reaction lists needed to simulate them can be added to any simulator.

These simulators can expose rule-based modeling to their users by using libmoleculizer internally to generate all or part of the network and add the equations generated to their own structures. The equations can either be generated and added all at once, before the simulation begins, or can be generated and added periodically, using an *on-the-fly* generation scheme, in which only enough of the network is generated at any moment to get through the next time steps.

This invites the question of how libmoleculizer actually generates reaction networks from systems of rules. Generally speaking, once a rules file is loaded into libmoleculizer, libmoleculizer creates an initial list of species and an empty list of reactions. The initial list of species consists of all the monomeric protein species for all proteins described in the network, as well as any other species explicitly defined by the user. Libmoleculizer also records each of the species in the initial species list as *unexpanded*. At this initial moment, libmoleculizer has not yet generated any reactions.

In an iterative process, either automatically-directed (used in pre-generative schemes) or user-directed (as used in on-the-fly generation), libmoleculizer goes through the list of species it knows about and *expands* them one at a time. What this means is that libmoleculizer takes that species and compares it against each of the rules, checking for the correct *features* needed by that rule. In the case of binary reaction rules, the species is checked against both the rules as well as features that have already been found in other already expanded features. When the expanding species matches a rule (or a rule and a second species, in the case of binary reaction rules). The matching species is taken, possibly along with its already expanded mate, and the transformation defined in the transformation is performed upon the species ( for instance, a kind of enzymatic reaction may look for molecules that have been singly phosphorylated; the transformation could turn the singly phosphorylated modification to a doubly phosphorylated one). Libmoleculizer then examines the product(s). If any of them are new - not in the list of species libmoleculizer knows about - they are added there as a newly generated, unexpanded species. The reaction is added as a newly generated reaction to the system. As libmoleculizer expands species, the network grows; if and when there are no more species to generate the entire network has been generated and libmoleculizer halts.

At any time during this process, either the complete lists of species/reactions or partial lists of recently created species/reactions can be read out by client programs. Additionally, libmoleculizer also records and manages other facts about the network, such as keeping track of sets of species that have certain structural properties, which can also be used in client programs to add features for their users.

## 2.3 Biochemical networks consist of chemical species and sets of chemical reactions between them

Libmoleculizer takes in models that represent an abstraction of a biochemical network and explicitly generates all or a part of that network. What this means is that libmoleculizer generates an explicit list of species along with all reactions those species can participate in.

This list of species and reaction can be sent to another simulator, as a list of variable names and reaction equations between them, and can be used to simulate the portion of the network that was expanded. This is what is meant when we say libmoleculizer expands biochemical reaction networks: it uses a set of rules to generate lists of species identities and of reactions between species.

## 2.4 Species are complexes of one or more indivisible units bound together

Species in libmoleculizer are constructed by combining together one or more indivisible units called *molecules*. Molecules have binding sites, and are joined together by creating bindings between their binding sites.

When one or more molecules joined together, a structurally unique *complex species* is formed, which has a unique identity as a biochemical species in that reaction network.

## 2.5 Molecules represent indivisible units of reaction

As discussed previously, molecules are the indivisible components of complex-species. Reactions can have structural effects like binding two unbound molecules together, breaking bonds between molecules, or exchanging one molecule in a complex species for another, but molecules cannot be be subdivided or combined.

### 2.5.1 Molecules have a structure consisting of binding sites and modification sites

Moleculizer recognizes two types of molecules. The first type are the *modifiable-molecules*. These molecules may have any number of unique binding sites, each of which may participate in binding this molecule to other molecules. Modifiable-molecules may also have any number of unique modification sites, which can each be associated with different post-translational modifications e.g phosphorylation, methylation, etc.

The second type of Moleculizer molecules are the *small-molecules*. These are molecules that can participate in reactions but which only have a single

non-differentiable binding site (which is considered to be structurally indistinguishable from the small-molecule itself) and no modification sites.

Typically, modifiable-molecules are appropriate structures for modeling individual proteins whereas small-molecules are appropriate for representing small molecules such as GTP and ATP.

## 2.6 Association and dissociation reactions between complex species are caused by the associations and dissociations on the binding sites belonging to the molecules that make up that complex

In the libmoleculizer system, all dimerization reactions between complex species result from an association reaction between two binding sites on two types of molecules, one belonging to each of the two complex species.

These association reactions between molecules are recorded in the network model as an *"association-reaction"* rule. As libmoleculizer generates new complex species by expanding the network, when it notices pairs of species with free binding sites corresponding to the rules, it ensures that the dimerization reaction is generated between them.

At the time the reaction is created, it is also reversed and the decomposition reaction is also created.

## 2.7 Species transformation reactions are made possible by the species possessing a "transformation-enabling" sub-complex

The second kind of reaction supported by libmoleculizer is very broad: the transformation-reactions, which are generated by *"transformation-reaction"* rules. Transformation reactions are those that involve (generally) enzymatic function. Complex species can have any number of modifications changed and any number of small-molecule swaps (e.g. an ATP turning to an ADP) as desired in a transformation reaction. Any reaction in which a species undergoes a post-translational modification or has one small-molecule exchanged for another will be a transformation reaction.

Libmoleculizer generates these reactions by looking for complex species that have a particular *transformation-enabling complex*, as specified by the user in different "transformation-reaction" rules in the model. (For instance, a transformation-enabling complex may be an enzyme bound to a target which itself is bound to an ADP molecule). When libmoleculizer finds a species that matches, it transforms it according to the rule provided by the user (possibly

creating a new species in the process) and adds the new reaction to the list.

Depending on how the user defined the rule, optional extra reactants or optional extra products may be generated at this time.

## 2.8 Allosteric reaction rates are differing rates of association and dissociation among molecules, conditional on the states of the complexes the reaction molecules are found in

One complicating but absolutely key feature observed in real biochemical networks is allosteric binding and unbinding. What this means is that, generally speaking, particular association reactions (a particular affinity between two binding sites on two molecules) occur at an 'intrinsic' rate which reflects the physical relationship between the two participating binding domains.

However, when the species in which the associating molecules live are in certain specific forms, the default rates of interaction change dramatically. This is typically for one of two reasons. Either another one or more molecules in the complex have blocked one of the participating binding sites; or the identities of other molecules in the complex have resulted in a conformational change in shape to one of the participating binding sites. This conformational shape change causes the physical relationship between the two binding sites to change, resulting in a change to the intrinsic reaction rate.

In libmoleculizer, allosteric interactions are modeled first by allowing users to define binding sites with multiple states, called shapes, which represent different conformational states, resulting in different kinetic profiles. In a section of the model where where allosteric conditions are listed, any number of conditions that cause the binding site to go from its default shape to one of its non-default allosteric state can be defined.

Finally, when entering the association interactions involving binding sites that have different conformational shape states, a default rate – for the default shape – is given, followed by any number of non-default allosteric rates.

In this way, any degree of "arbitrary" kinetics can be added to a moleculizer model.

## 2.9 Names in libmoleculizer

Libmoleculizer generates explicit species and reactions listings of biochemical reaction networks, which can be read out with the libmoleculizer API to use in other programs for other purposes.

Some words, however, should be said about the facilities for naming species that exist in libmoleculizer, because otherwise difficulties arise.

The problem with naming complex species is that they are just represented internally as sets of connected parts. Because the sets of parts (molecule types

and bindings) have no preferred ordering to them, there isn't any one name for the species – in fact there are at least as many names for a species as there are ways of describing its components ( simply, if a species consists of $x$ molecules, there are at least $x!$ ways of describing the species).

First, what IS NOT the problem. For any species, we can get a name that describes the species. This is never a problem. We can always take a name we've been given and get back the same species every single time.

What is the problem? The problem is that because species have so many names that can describe them, we might look at two identical species, and generate names for the two of them, and, because the names are different, conclude the two species must be different. Even though the two species are really the same, they could have different names. The problem we have is in uniquely identifying complexes – with the problem of making sure that a complex has a uniquely identifying name.

Because of this, libmoleculizer has two solutions. The first is a unique, temporary name, which is called a "Tag" or "Tagged Name" in other parts of the manual. This "Tag" is unique over a particular run of libmoleculizer. As long as tags are only compared during a single program run they are your best option. However, because they are only unique for a given run, they should never be used to compare species generated during different runs. For this, we must use something more permanent.

The second solution is a permanent name, which we call a "Unique ID". The unique id is a name that persists beyond program runs. An identical species will always generate the same unique id, no matter what. Accordingly, they are good for activities where biochemical networks are saved to disk, and need to be compared between simulation runs. The downside to using unique ids is that they are slower to generate, when compared with tags. Thus, tags are preferred in all situations in which they can be used.

# Chapter 3

# The MZR Language for expressing rule-based models

A libmoleculizer input file is called a Moleculizer Rule File (MZR). MZR files are text files that contain a model of a particular biochemical network, written in a language that describes a set of molecule types, a set of reaction rules describing interactions between those molecular types, along with other information needed to describe these types of systems. This rule-based representation is used by libmoleculizer to create the explicit species and reactions that make up the expanded network.

This chapter describes the MZR file syntax, and tells how to use it to write rule-based models representing virtually any biological pathway that can be expanded into explicit lists of species and reactions that comprise the pathway.

## 3.1 MZR File Overview

A MZR file consists of multiple sections, each of which describes a portion of the information needed to represent a biochemical reaction network in libmoleculizer. There are sections for describing different molecule types, types of relevant post-translational protein modifications, kinds of reaction families, allosteric conditions, families of species and more.

When a MZR file is supplied to libmoleculizer, it is compiled into an intermediate xml format, which is subsequently used internally by libmoleculizer. For more information on the internal format either consult other parts of this manual (the information should be in here somewhere), or consult the original moleculizer documentation, which has a full discussion of it.

## 3.2 MZR File Syntax

A MZR rules file consists of one or more of the following sections: "Modifications", "Molecules", "Explicit-Allostery, "Allosteric-Classes, "Association-Reactions", "Transformation-Reactions", "Species-Classes", and "Explicit-Species". Most of these sections are optional.

Each section must begin with a header declaring the name of that section. A header is any line that begins with the string "===" and contains the exact type-sensitive name of that section somewhere in the line. Although this format may sound unusual, the end result is that a typical MZR file will have a top-down structure that looks like the following example.

```
===== Modifications =====
  # Modification definitions...

===== Molecules ============
  # Molecule definition...

=====  Association-Reactions =====
  # Dimerization type reaction definitions....

# And so on....
```

Each section is comprised of one or more statements, each of which adds a single piece of information to the biochemical network model. (There is a technical exception: many statements are not truly standalone, because they implicitly require the existence of another standalone statement in the model in order to be complete. An example could be a rule defining a protein interaction resulting in a post-translational modification to a target protein. While the reaction rule itself is essentially standalone, it requires a statement defining the modification in question to be located in the Modifications section in order for the reaction rule to be completely understood by libmoleculizer.)

In a MZR file, statements are separated by semi-colons; all other white-space within them, including newlines is ignored. Each statement consists of a comma-separated list of sub-statements, which can either describe complexes or parts of complexes (what are referred to herein as complex forms), give assignments, or define reaction rules patterns. What this actually means and how it is used will become clear over the next few sections. For now, understand that every statement ends with a semi-colon, and consists of one or more comma-separated components.

Finally, comments are permitted, and use the comment-start character "#". Comments extend from their beginning to the end of the line on which they appear.

### 3.2.1 Modifications

The modifications section consists of a set of statements that each defines a unique modification that can be used and referred to within the model. Each such definition consists of either one or two components, each of which takes the form of an assignment. The first assignment is mandatory and gives the modification being defined a name; it takes the form "name=MODIFICATION_NAME".

If mass-based reaction rate extrapolation is used, an additional, otherwise optional, assignment of "mass=MASS_OF_MODIFICATION" is needed as a part of the modification definition statement as well.

The first example gives an example of a modification section in a model that does not use reaction rate extrapolation (the default).

```
=== Modifications ========================
     name = None;
     name = Phosphorylated;
     name = DoublyPhosphorylated;
     name = GDP;
     name = GTP;
```

The next example gives an example of the same modification section, but which now supplies mass assignments as well. Notice how within each statement, the two component sub-statements (the name assignments and the mass assignments) are only separated with commas.

```
=== Modifications ========================
     # If these modifications are to be used in reactions using mass-based
     # rate extrapolation, then must be given masses.
     name = None, mass = 0.0;
     name = Phosphorylated, mass = 42.0;
     name = DoublyPhosphorylated, mass = 84.0;
     name = GDP, mass = 100.0;
     name = GTP, mass = 110.0;
```

### 3.2.2 Molecules

The molecules section consists statements that each defines either a mod-mol or a small-mol in the model. Both types of molecules correspond to types of indivisible physical units that participate in reactions; the difference between the mod- and small- mol types lies in the level of resolution of structural detail they are given, and consequently in the kinds of reactions they can participate in. Mod-molecules are structural molecules that have one or more binding sites and zero or more modification sites; they also may have masses, which can be used for mass-based reaction rate extrapolation if desired. Small-molecules are molecules that participate in binding reactions with other molecules, but which do not themselves have differentiated binding sites or any modification sites at all. However, like mod-molecules, they may have a mass if mass based reaction rate extrapolation is used.

Typically mod-molecules are used to represent biochemically active proteins, whereas small-molecules are used to represent small molecules.

#### Statements defining Mod-Molecules

Each statement that defines a mod-mol is made up of a complex-form specification that defines it, followed by an optional weight assignment (as always, weights are only mandatory if mass-based reaction rate extrapolation is used).

The meat of the definition occurs in the specification of the mol's complex-form statement: a complex form that defines a mod-mol consists of the mol's name followed by a parenthesis enclosed description of the structure of that mol – its binding sites and its modification sites.

Within a complex-form, each binding site can simply be defined by writing the name of that binding site in the parenthesis. If this is all the information provided, the binding site will be created with one and only one binding site shape, called "default". If the binding site exhibits biochemically allosteric behavior (if it displays multiple distinct profiles of kinetic behavior, depending on conditions, allosteric states must be specified here (for instance, a binding site may be more or less active depending on whether or not other binding sites are bound or not, or depending on the state of a particular modification site on that mod-mol, or depending on what kind of complex the mod-mol might be in). To do this, all the legal binding site shapes are declared in a comma separated list inside a pair of braces immediately following the name of the site. The first value is assumed to be the default shape for the binding site.

Each modification site consists of the modification site's name, proceeded by a * character (see example 4 in Fig **??**) for an example). The name of the modification site must be immediately followed by a pair of braces containing a list of modifications the modification site can have. The first in the list is assumed to be the default modification value for that modification site. Moreover, each modification type referred to by the list must have a corresponding definition in the modifications section. These references, like everything else in these files, are case-sensitive.

```
==== Molecules ==========
    # 1. Defines a mol named Ste20 with a single binding site named
    # 'to-Ste4'.
    Ste20(to-Ste4),
         mass = 100.0;

    # 2. Defines a mol with four binding sites.
    Ste5( to-Ste4, to-Ste11, to-Ste7, to-Fus3),
         mass = 100.0;

    # 3. Defines a mol with three binding sites. The third binding site,
    # to-Ste20, is defined with two possible shapes: the default
    # shape, named 'default' and a second shape called 'obstructed'.
    # In the allosteric sections, conditions under which the binding
    # site can be in either state can be described, and reaction rules
    # involving the to-Ste20 binding site can be given different rates
    # depending on the state of to-Ste20
    Ste4(to-Gpa1, to-Ste5, to-Ste20 { default, obstructed} ),
         mass = 100.0;

    # 4. Defines a mol with a single binding site and a single
    # modification site (modification sites begin with *'s). The
    # modification site, called ``PhosSite'' can have three different
    # modifications, a default modification named ``None'', as well as
    # modifications called Phosphorylated and DoublyPhosphorylated.
    # These modifications must be defined in the modifications section.
    Ste7( to-Ste5, *PhosSite { None, Phosphorylated, DoublyPhosphorylated } ),
         mass = 100.0;

    # 5. Defines a mod-mol with two binding sites, the first of which has
    # shape information, and one modification site.
    ReceptorGpa1Complex(to-Ste4 { GDP-bound-shape, GTP-bound-shape},
                        to-alpha, *GXP-site { GDP, GTP} ),
         mass = 100.0;
```

### Defining Small-Molecules

Each small-molecule statement consists of a name assignment for the new small-mol, as well as a mass assignment (although only needed if mass-based reaction rate extrapolation is used).

Small-molecules differ from regular molecules in the simplicity of their structure. This has two advantages. First, due to the simplicity of their structure,

libmoleculizer small-molecules correspond to biological small molecules in terms of what they can do and represent in the model. Second, due to their simplicity, they can be used internally by libmoleculizer more efficiently than molecules. Thus, by using small-molecules when appropriate, users both gain modeling and speed advantages over using regular molecules.

Several examples of small-mol statements are provided below.

```
==== Molecules ==============
   # If no mass is specified
   massless_alpha;
   alpha, mass = 42.3;
```

Mod-mol defining statements and small-mol defining statements can be freely mixed within the Molecules section.

## 3.3 Explicit-Species Section

The explicit-species section is where user specified names are assigned to different non-trivial (more than one mol component) complex species. Each statement within this section assigns a specific complex a name. These names are recorded by libmoleculizer and can subsequently be looked up and referred to.

The first component of an explicit-species statement is a complex-form that specifies the complex-species in question. Because explicit-species must be specifies completely this means that every modification state for every modification site on every mol must be specified.

To write the complex-form that represents the species in question, join the molecules with periods. For any binding between two species, append a binding id, consisting of a ! followed by an index unique to that binding, to the name of that binding site. If one of the molecules is a small-mol, simply write the associated binding syntax inside its structural specification (the parentheses). Finally each modification site that exists in the complex must be written along with its modification. An example is given in Fig **??**.

```
=== Explicit-Species ==================

    # The following explicit-species statement represents a
    # ReceptorGpa1Complex bound to a Ste4, from the
    # ReceptorGpa1Complex's to-Ste4 site to the Ste4's to-Gpa1 site.
    # Additionally, the ReceptorGpa1Complex's GXP modification site
    # should be bound to GDP.
    ReceptorGpaComplex(to-Ste4!1, *GXP-site { GDP } ).Ste4(to-Gpa1!1),
        name = ReceptorBoundGpa;

    # The following shows how to represent a small-mol (alpha) binding as part
    # of a three-way complex.
    alpha(!1).ReceptorGpa1Complex(to-alpha!1, to-Ste4!2, *GXP-site {GDP} ).Ste4(to-Gpa1!2),
        name = alpha-receptor-Ste4-complex;
```

## 3.4 Explicit-Species-Class

Statements within species-stream section have a similar relationship to statements in the explicit-species section as statements in the allosteric-omnis section have to those within the allosteric-plexes section.

In the explicit-species section, statements give an exact specification for a complex and associate a name with that exact complex. In the species-streams section, statements describe classes of complex and associate a name with the class.

As before, the first component of the statement is a complex-form specification. However, the additional syntax and meaning that was allowed and present in the allosteric-omnis section tends to apply here. For instance, to require a binding site as being unbound as a condition of class membership, write that binding site out in the mol it is contained in. To specify that a binding site is bound as a part of an unknown binding, write out the name of that binding site followed by the "!*" syntax. Likewise, modification values at modification sites may be specified or not, depending on user preferences.

The species-stream complex-form specifier should be followed by an assignment "name=YOUR_NAME_HERE".

```
====== Explicit-Species-Class ============
# This will select for all complexes that contain a phosphorylated A.
A(*M{Phos}),
        name=withaphosphorylated\_A;
```

## 3.5 Association-Reactions

The dimerization gens section consists of a set of statements, each of which describes an association/decomposition reaction between specific binding sites on specified molecules. In addition to having default on and off rates, the association and decomposition rates may be made dependent on the states of the specified binding sites. This allosteric behavioral information is specified as further sub-components of the main rule statement.

A dimerization-gen statement consists of one or more components. The first section describes the specific form of the reaction followed by default on and off rate information. Subsequent components describe allosteric behavior and, for each allosteric condition, add three components. In these cases, the first portion describes allosteric states of binding sites participating in the first, master reaction and the second and third portions describe the on and off rates for that allosteric case, and so on.

As always, components referred to within dimerization-gens statements must have corresponding definitions in other parts of the model. For instance, Molecules referred to in the reaction specification must be defined in the Molecules section, along with the binding sites and binding shapes referred to in the rules.

```
# Basic Dimerization.
A(to-B) + B(to-A) -> A(to-B!1).B(to-A!1),
      kon = 100.0,
      koff = 10.0;

  #
  # alpha/Receptor Binding.  This reaction shows how a binding of a
  mod-mol to a small-mol works
  alpha() + Receptor(to-alpha) -> alpha(!1).Receptor(to-alpha!1),
          kon = 100.0,
          koff = 10.0;

  # The receptor@oligo-1 + receptor@oligo-1 reaction has on and off
```

```
# rates of 10.0 when both receptors are in their default states.
# When one of them is in state {enabled} and the other is in state
# {default}, the on rate is 75 and the off rate is 10.0; when both
# participating binding sites have shape {enabled}, the on rate
# becomes 50.0.
Receptor(oligo~1) + Receptor(oligo~1) -> Receptor(oligo~1!1).Receptor(oligo~1),
      kon = 10.0,
      koff = 10.0,
   Receptor(oligo~1 { default} ) + Receptor(oligo~1 {enabled } )
     ->
       Receptor(oligo~1!1).Receptor(oligo~1),
         kon = 75.0,
         koff = 10.0,
   Receptor(oligo~1 { enabled } ) + Receptor(oligo~1 { enabled} )
     ->
       Receptor(oligo~1!1).Receptor(oligo~1),
         kon = 50.0,
         koff = 10.0;
```

## 3.6 Defining Allosteric Conditions

Allosteric behavior is represented in libmoleculizer as differing rates of interaction within dimerization reactions, conditional on the specific shapes of interacting binding sites. Specification of the conditions under which allosteric binding sites transition between their different shapes occurs in the two allosteric sections: allosteric-plexes and allosteric-omnis. Both of these sections are used for the same purpose: to give structural conditions such that binding sites of interaction change state. (The different states on a binding site of interaction are used in the rules defining those interactions to specify different interaction strengths depending on what shapes the binding sites have). The difference between the two sections is in where libmoleculizer looks for the structural conditions. That is, in both sections statements are given that describe two things: a biochemical entity in a specific enabling state, and one or more resulting binding site shape changes the binding sites within that entity. The difference between the two is that in the allosteric-plexes section, the structural entity specified must match exactly; in the allosteric-omnis section, the structural entity specified matches all complexes which contain the specified entity. (They may either match it exactly or the specified entity may be a sub-complex of the biochemical entity being matched.)

## 3.7 Allosteric-Complexes

An allosteric-plex is a rule that tells how the binding sites within specific complexes (specific up to state of its unspecified modification) have their shapes changed. Binding site shape information may then be used for specifying allosteric binding rates within the dimerization-gens section.

Each statement in this section consists of a single complex-form specification of a whole complex, that includes special syntax to describe how the binding sites in that complex change their shape.

The first step in the construction of the appropriate allosteric-statement is the construction of the complex-form that describes the relevant complex. This

is done by specifying the complex as a set of molecules, with binding-site binding information joining binding sites between molecules.

These are just examples of complex-forms. Because they do not yet use the special allosteric syntax that describes updated binding sites, they are not yet valid allosteric-plex statements.

```
A;  # This is the complex form that describes a particular singleton
    # mol.

A(first!1).B(second!1); # This is a complex form that describes a
                        # particular dimer of an A bound to a B, from
                        # the A's ``first'' binding site to B's
                        # ``second'' binding site.

# This complex describes a B-A-C trimer.  Looking at the binding
# information, we see that A is bound at its ``first'' site to B at
# its ``b-site''.  Likewise A is bound at its ``third'' site to the C
# at the C's ``c-site.''
A(first!1, third!2).B(b-site!2).C(c-site!1);
```

Although these complex-forms are syntactically correct (although they do not yet contain enough information to be valid allosteric statements, or any other valid statement at this moment), they may not be valid complex-forms. To be valid, the objects referred need to be be consistent with other parts of the model. By referring to A, B, and C molecules, these each must be defined in the "Molecules" section of the model. Furthermore, all three must be mod-molecules and be defined with the appropriate binding sites: A must have "first" and "third" binding sites and so forth.

By getting this far, in the context of an allosteric-plex, we have completely specified a complex type. (In the examples, these are A singletons, AB dimers, or ABC trimers respectively). That is, they completely specify the **the binding state** of the complex, they are so far silent on the modification state of the complex. If, for instance, the A molecule has a single modification site, which can either be phosphorylated or not, then AB dimers (complex-type) can have two forms of corresponding species: the AB dimer with the A phosphorylated and the AB dimer with the A unphosphorylated. To specify further conditions on the modification state of the specified complex, further modification site information must be known.

```
=== Modifications ===
name = None;
name = P;

==== Molecules  =====

A(to-b,*M{None,P});
AP( to-b, *M1{None,P}, *M2{None,P});
B(b);

==== Allosteric-Complexes =====

# Because no mention is made of A's modification site ``M'' here,
# this allosteric-plex will match AB dimers in which A is both
# phosphorylated and non-phosphorylated.
1.  A(to-b!1).B(b!1);

# Because this complex specifies a state for the modification site
# in the A mol, this complex will only match AB dimers in which the
# A is unphosphorylated.
2.  A(to-b!1, *M{None}).B(b!1);

# Like 2, but will match only AB dimers in which the A is
# phosphorylated.
3.  A(to-b!1, *M{P}).B(b!1);

# Partial specification is allowed (as well as full and no
# specification), this will identify all AP.B dimers in which
```

```
  # the M2 phosphorylation site on the AP is phosphorylated.  The
  # M1 site may be in either phosphorylation state to match.
  4.  AP( to-b!1, *M2{P}).B(b!2);
```

At this point, any specific structural form can be specified, with any level of selection on the modification state of that complex. To completely specify an allosteric-plex, only one thing remains: to specify the binding-site shape changes that occur for complexes that match the specification.

This is accomplished by specifying the binding-sites in question by making sure to include them in the complex form specification and then following them with the allosteric sections' special syntax: "FinishingState ¡- *", where FinishingState is the specific name shape the associated binding site ends up in.

```
=== Allosteric-Complexes =======

  # 1. This rule says that when there is an AB dimer, the ``to-c''
  # site on the A will be occluded.
  A(to-b!1, to-c {occluded<-*}).B(to-B!1)

  # 2. This rule states that when an A is doubly phosphorylated, its
  # ``binding-site'' ends up in the active shape/state.
  A(*M1{P}, *M2{P}, binding-site {active<-*});

  # 3. This rule states that when there is a ReceptorGpa1Complex bound to
  # an alpha and also bound a Ste4 exactly, then the to-Ste20 site on
  # the Ste4 in that complex takes the shape obstructed.
  ReceptorGpa1Complex(to-alpha!1, to-Ste4!2).alpha(!1).Ste4(to-Gpa1!2,to-Ste20 { obstructed <- * });

  # 4. This rule says that when there is an B-A-B complex, the
  # primary-site binding sites on the B's become active.
  A(left!1, right!2).B(to-A!1, primary-site {active <-} ).B(to-A!2, primary-site { active <- *);
```

## 3.8 Allosteric-Omniplexes

This section is virtually identical to the allosteric-plexes section, with one key conceptual difference. In the allosteric-plexes section the complex forms specified correspond to structurally identical (consisting of the same molecules in the same binding configuration) complexes. In the allosteric-omnis section, the complex forms match classes of complexes who possess the sub-complexes specified. See the example below for more examples.

```
# As an allosteric-plex, the following phrase will match singleton
# A's.  As an allosteric-omni, the following phrase will match all
# complexes that contain an A mol.
A;

# As an allosteric-plex, the following phrase will match all A-B dimers
# with a phosphorylated A.  As an allosteric-omni, the following will
match all complexes that contain a phosphorlyated A bound with a B.
A(to-b!1, *M{Phos}).B(to-a!1);
```

Essentially that is the only difference between an allosteric-plex statements and allosteric-omni statements: that of how to interpret them (which is also why they need separate sections). However, because complex-forms in allosteric-omnis select for classes of structural complex, there are two additional minor forms of specifiers allosteric-omni complex-forms can use, that allow specifying whether a binding site is or is not bound.

To specify a site is explicitly unbound, simply list the binding site in the complex form specification.

```
==== Allosteric-Omniplexes =====
  # This specifies all complexes that contain an A bound to B.
  A(to-b!1).B(to-a!1);

  # By adding an otherwise empty binding site ''to-c'' in the
  # structural specification, we indicate that this specifies all
  # complexes that contain an A bound to B whether the A has an unbound
  # ''to-c'' site.
  A(to-b!1, to-c).B(to-a!1);


  # By adding a partial binding specification(an exclamation point
  # indicating a bound state, followed by a * which matches anything),
  # this specification will select all complexes that contain an A
  # bound to a B where the A's binding site ''to-c'' is also bound.
  A(to-b!1, to-c!*).B(to-a!1);

  # Note that by default the following complete rule does not require
  # that the ''to-c'' site which changes shape must be unbound.
  A(to-b!1, to-c {occluded <- *}).B(to-A!1);

  # If we wish to force this constraint, we must explicitly force the
  ''to-c'' site to be unbound, using the ''!-'' syntax..
  A(to-b!1, to-c!- {occluded <- *}).B(to-A!1);

  # We can also combine syntaxes.  This example shows how binding-site
  # shapes can be updated on bound-and-specified(the !1 on A),
  # bound-and-not-specified (the to-something) on C, unbound (unbound on
  # B), or unspecified (default) on B) within a rule.

  A(to-b!1 {state1<-*}, to-c!2, fizzy)
  .B(to-a!1,unbound!- {state2 <-* }, default {state3<-*} )
  .C(to-b!2, to-something!* {state4});
```

One final note is that all specified allosteric-omni complexes must be simply connected. That is, everything specified must form one contiguous sub-complex in which every mol specified is unique.

## 3.9 Transformation-Reactions

The dimerization-gens section deals with all binding site interactions that bind or unbind proteins together. A second important class of protein-protein reactions are the generic enzymatic reactions. These are reactions in which post-translational modifications can be made and small molecules are exchanged, possibly with the aid of a helper molecule, and possibly resulting in an additional final product as well.

The omni-gen rule represents a generic reaction that can generate, for each complex species containing a particular omniplex, a reaction with flexible characteristics. The generated reactions can change any small-mol component of the omniplex into another small-mol, if desired. The generated reactions can change the modification at modification site on any one mod-mol component of the omniplex, if desired. The generated reactions can be binary, if desired, requiring any explicit reactant species as a co-reactant with the species that is recognized as containing the omniplex. The generated reactions can produce an arbitrary explicit product species, in addition to the transformed primary reactant, if desired.

Each of these separate activities on the part of the generated reactions is engaged simply by including the appropriate elements in this reaction generator specification.

The first half of an omni-gens statement, the reaction specification should be written in the form “Omniplex + (optional_explicit_reactant) -¿ Transformed_Omniplex

+ (optional_explicit_product)". Examples follow:

```
==== Explicit-Species =====

A(to-b!1).B(to-a!1), name = AB_dimer;


==== Transformation-Reactions ==========

# A simple phosphorylation reaction
E(b1!1).S(b1!1, *Mod{ None }) -> E(b1!1).S(b1!1, *Mod{ Phosphorylated }),
      k = 100.0;

# By listing a site, such as the b2 site on E here, and leaving it
# empty, this specification will only apply to those complexes with
# that site free.
E(b1!1, b2).S(b1!1, *Mod{ None }) -> E(b1!1).S(b1!1, *Mod{ Phosphorylated }),
      k = 100.0;

# Likewise, by adding the !+ syntax, you can force that a binding site
# simply must be bound.
E(b1!1, b2!+).S(b1!1, *Mod{ None }) -> E(b1!1).S(b1!1, *Mod{ Phosphorylated }),
      k = 100.0;

# The same phosphorylation, but this time adding a GTP -> GDP
# hydrolization.
E(b1!1,to-GxP!2).S(Sb1!1, *Mod{ None }).GTP(!2) ->
  E(b1!1,to-GxP!2).S(Sb1!1, *Mod{ Phosphorylated }).GDP(!2),
    k = 100.0;

# Multiple small molecules can be exchanged at once.
E(b1!1, b2!2).GTP(!1).GTP(!2) -> E(b1!1, b2!2).GDP(!1).GDP(!2), k = 100;

# Multiple phosphorylations can be done at once.
A(b1!1, *M1{none}).B(b2!1, *M1{none}) -> A(b1!1, *M1{none}).B(b2!1, *M1{none}),
    k = 100.0;

# The same, but this time adding an additional optional
# explicit-species.
E(b1!1,to-GxP!2).S(Sb1!1, *Mod{ None }).GTP(!2) + AB\_dimer->
  E(b1!1,to-GxP!2).S(Sb1!1, *Mod{ Phosphorylated }).GDP(!2),
   k = 100.0;

# an optional product species....
E(b1!1,to-GxP!2).S(Sb1!1, *Mod{ None }).GTP(!2) ->
  E(b1!1,to-GxP!2).S(Sb1!1, *Mod{ Phosphorylated }).GDP(!2)  + AB\_dimer,
    k = 100.0;

# With both an optional reactant and an optional product.  Remember,
# the names of singleton species (the singleton A mol) can be used as
# explicit names.
E(b1!1,to-GxP!2).S(Sb1!1, *Mod{ None }).GTP(!2) + A ->
  E(b1!1,to-GxP!2).S(Sb1!1, *Mod{ Phosphorylated }).GDP(!2)  + AB\_dimer,
    k = 100;
```

## 3.10 A simple modeling example

### 3.10.1 Two enzyme conversion of A-¿A*

In this section, we will give as an example a two-enzyme, single substrate reaction among hypothetical proteins. Let us suppose we have a system with a target protein Y, a protein with two enzyme binding domains (E1 and E2) and a phosphorylation site and two enzymes A and B. A has a binding domain that binds to E2 and a small-mol binding site "to-Gxp" that is bound to either GDP or GTP.. B has a binding domain that bindings to E1 as well as a phosphorylatable modification site.

Suppose that the pathway under consideration is one in which Y becomes bound by a GTP-possessing A and phosphorylated B, whence A hydrolyzes its GTP and B transfers its P03 group to Y's unphosphorylated site. To model this system we must define the 3 proteins, along with GTP and GDP (these are the 5 molecules involved in the system), modification states of none and

phosphorylated, along with 2 dimerization rules (the rules that govern binding and unbinding of A+Y and of A+B) and a single omni-gen rule (the rule that describes the transformation of the A.B.Y trimer).

```
===== Modifications ==========
# Note that this model, for the sake of using little space, does not
# use mass-based extrapolation, so no masses are included in the
# modifications or in the mol definitions.

name = none;
name = phosphorylated;


===== Molecules ==============
name = GTP;
name = GDP;
name = A(to-Y, to-Gxp);

# Note that by default, US start out phosphorylated.  We also could
# have had the default be none, although we would have either had to
# then explicitly create phosphorylated Bs, or provide another rule
# telling how Bs get phosphorylated.

name = B(to-Y, *mod-site {phosphorylated, none} );
name = Y( to-A, to-B, *mod-site {none, phosphorylated});

===== Dimer-Gens ===========

B(to-Y) + Y(to-A) -> B(to-Y!1).Y(to-A!1),
    kon = 50.0;
    koff = 500;

A(to-Y) + Y(to-A) -> A(to-Y!1).Y(to-A!1),
    kon = 50.0;
    koff = 500;

===== Transformation-Reactions ============

A(to-Y!1, to-GxP!2).B(to-Y!3, *mod-site {phosphorylated} ).Y(to-A!1, to-B!3, *mod-site {none}).GTP(!2) ->
  A(to-Y!1, to-GxP!2).B(to-Y!3, *mod-site {none} ).Y(to-A!1, to-B!3, *mod-site {phosphorylated}).GDP(!2),
  k = 100.0;
```

Suppose we wish to extend the model slightly to include dynamics that say that upon phosphorylation of Y, A and B will dissociate extra-quick. To do this, we would add one new allostery rule (which could either be an allosteric-omni or an allosteric-plex, depending on what our intentions are) as well as new allosteric sections to both dimerization-gens that describe speedy unbinding in the presence of phosphorylated Y. That example is shown below.

```
===== Modifications ==========
# Note that this model, for the sake of using little space, does not
# use mass-based extrapolation, so no masses are included in the
# modifications or in the mol definitions.

name = none;
name = phosphorylated;


===== Molecules ==============
name = GTP;
name = GDP;
name = A(to-Y, to-Gxp);

# Note that by default, Bs start out phosphorylated.  We also could
# have had the default be none, although we would have either had to
# then explicitly create phosphorylated Bs, or provide another rule
# telling how Bs get phosphorylated.

name = B(to-Y, *mod-site {phosphorylated, none} );
name = Y( to-A, to-B, *mod-site {none, phosphorylated});

===== Allosteric-Omni =========
# This must be an omni, as opposed to a plex, because these kinetics
# should apply to all complexes Y occurs in: AY, BY, and ABY - any of
# them should quickly dissociate.

Y(to-A {inactive <-*}, to-B {inactive <-*}, *mod-site {phosphorylated});


===== Dimer-Gens ===========

# Both these rules say that by default B and Y (or A and Y) bind
```

```
# together, but when Y is phosphorylated, they practically fly apart.

B(to-Y) + Y(to-A) -> B(to-Y!1).Y(to-A!1),
    kon = 50.0;
    koff = 5.0,
  B(to-Y {default} ) + Y(to-A {inactive} ) -> B(to-Y!1).Y(to-A!1),
      kon = 1.0;
      koff = 500.0;

A(to-Y) + Y(to-A) -> A(to-Y!1).Y(to-A!1),
    kon = 50.0;
    koff = 5.0,
  A(to-Y {default} ) + Y(to-A {inactive} ) -> A(to-Y!1).Y(to-A!1),
      kon = 1.0;
      koff = 500.0;

===== Transformation-Reactions ==========

A(to-Y!1, to-GxP!2).B(to-Y!3, *mod-site {phosphorylated} ).Y(to-A!1, to-B!3, *mod-site {none}).GTP(!2)
 ->
  A(to-Y!1, to-GxP!2).B(to-Y!3, *mod-site {none} ).Y(to-A!1, to-B!3, *mod-site {phosphorylated}).GDP(!2),
  k = 100.0;
```

# Chapter 4

# Libmoleculizer interfaces

Libmoleculizer is a software library: a software component ready to be used in building other software programs. In order for these other programs (clients) to use libmoleculizer, it is important that they understand the interfaces that libmoleculizer uses. That is the topic of this chapter.

Currently, libmoleculizer has two interfaces that can be used by other programs. The first, primary interface is written in C++, the primary language libmoleculizer is written in. The second interface is written in C, and which largely wraps and exposes functions in the C++ interface.

This chapter will discuss both of these.

## 4.1 The C++ Interface

If your program is written in C++, you should use the C++ interface (although it is possible to use the C interface as well). To do this, make sure to include the header file "libmoleculizer/mzr/moleculizer.hh" in your program.

Also please notice that, unless specified otherwise, all classes and function described are all contained in namespace mzr.

### 4.1.1 The mzrSpecies class

The mzrSpecies class represents a unique type of complex species. This class has several main functions.

**std::string mzrSpecies::getTag() const**

Calling this function causes the tagged name of the mzrSpecies to be returned. The tagged name is a unique name for this species, but which is only valid during a single libmoleculizer run.

**std::string mzrSpecies::getName() const**

Calling this function returns the unique id of the mzrSpecies. This unique id is unique to the complex completely. The same complex will always generate the same unique id, even between sessions. However, it takes much more time to generate than getting a tagged name.

**std::string mzrSpecies::getWeight() const**

This function returns the mass of the species. Note this is calculated by summing over all the masses of the molecules that make up the species, as well as summing over all the modifications on the molecules in the species. Therefore the masses are only valid if correct masses were added to the model in the first place.

**std::string mzrSpecies::expandReactionNetwork() const**

This function causes the mzrSpecies to be expanded. If the species has been expanded already, this will have no effect. However, the first time this is called on a species, that species is compared against all other already-expanded mzrSpecies. If new reactions can be formed while doing this, they are created, along with any new species products and everything is added to the running libmoleculizer.

**The mzrReaction class**

The mzrReaction class represents a reaction between mzrSpecies. This includes information about the numbers and identities of reactants, the numbers and identities of products, as well as kinetic information.

**bool mzrReaction::hasReactant( const mzrSpecies* ) const**

Pass this function in a pointer to a mzrSpecies and it will return whether or not that species is a substrate of the reaction.

**bool mzrReaction::hasProduct( const mzrSpecies* ) const**

Pass this function in a pointer to a mzrSpecies and it will return whether or not that species is a substrate of the reaction.

**const multMap& mzrReaction::getReactants() const**

This function returns a constant reference to a multMap, which is a type defined as std::map<mzrSpecies*, int>, representing the reactants of the reaction. Each reactant type is in the map as a key; its value is the stochiometry of that reactant (1 means one of that species is used up in single reaction; 2 means two of that species are used up; etc).

**const multMap& mzrReaction::getProducts() const**

This function returns a constant reference to a multMap, which is a type defined as std::map<mzrSpecies*, int>, representing the products of the reaction.. Each product type is in the map as a key; its value is the stochiometry of that reactant (1 means one of that species is created in single reaction; 2 means two of that species are created; etc).

**double mzrReaction::getRate() const**

This function returns the rate of the reaction. Typically, this value will be the same kinetic parameter that went into the reaction rule that created this reaction. However, if mass-based reaction rate extrapolation is used, the base kinetic rate is multiplied by the reduced mass of the reactant species to give a derived rate for the equation. Regardless of which system libmoleculizer is configured to use, this function will return the kinetic rate of the reaction.

**std::string mzrReaction::getTaggedName() const**

This function will return a unique name for the reaction, based on combining the tagged names of its products and reactants. As such, much the same can be said about this function and the names it produces as can be said about the same function for mzrSpecies. Its advantage is that it is quite fast (a constant time operation); the downside is that tagged names are only unique over a given moleculizer run.

**std::string mzrReaction::getUniqueName() const**

This function will return a unique name for the reaction, based on combining the unique ids of its products and reactants. As such, much the same can be said about this function and the names it produces as can be said about the same function for mzrSpecies. Its advantage is that the names it produces uniquely identify reactions between runs of libmoleculizer; the disadvantage is that it is much slower than getTaggedName (which operates in a fast constant time) – getUniqueName runs at a slow O(n log n) at best.

### 4.1.2 moleculizer

The moleculizer class represents a reaction network that can be expanded. In a very high level view, it is responsible for reading and interpreting rule-based models and then managing the expansion of that network using those rules.

**void moleculizer::moleculizer()**

This is the one and only constructor of a moleculizer object.

**void moleculizer::setGenerateDepth(int)**

This function sets the generation depth that is to be used for species and reaction generation. It can be called at any time. By setting it to 0, no reaction generation is ever performed. 1 is the default and means that when a species or reaction is expanded, a single round of species and reaction generation is performed. For numbers greater than 1, that many rounds of species/reaction generation will be performed. For instance, when the generation depth is two, when a species is expanded, each of the new species produced by that expansion are themselves expanded (although only a single time).

**unsigned int moleculizer::getGenerationDepth()**

This function returns the generation depth. This usually has the value of 1, but can be set to any non-zero value. This function returns the generation depth's current value.

**void moleculizer::setRateExtrapolation(bool)**

This function controls whether or not libmoleculizer should use mass-based reaction-rate extrapolation when it expands a reaction. If this is set to false (the default value), when a reaction is created based on a reaction rule, the reaction rule's kinetic value is passed directly into the reaction. If set to true, then the kinetic value is multiplied by the reduced-mass of the reacting species before being passed into the reaction. This function MUST be called prior to loading a model in order for it to have an affect.

**bool moleculizer::getRateExtrapolation() const**

This function returns a boolean that says whether or not reaction rate extrapolation is being used in the current, loaded model.

**void moleculizer::loadCommonRulesFileName(const std::string& aFileName )**

This function takes a filename of a file containing a rule-based model using the standard mzr syntax and loads it.

**void moleculizer::loadCommonRulesString(const std::string& commonRulesAsString)**

This function takes a string containing the contents of a mzr rules file and loads it.

**void moleculizer::loadXmlFileName( const std::string& aFileName )**

The typical rules files used by libmoleculizer are mzr files, written in a compact, human readable form. However, they are compiled internally to an xml form.

If, for some reason, you want to work with models written in this internal form (for instance, for legacy support – the xml syntax used internally is the old moleculizer rule format), you can load a model by passing in the name of the file into this function.

**void moleculizer::loadXmlString( const std::string& documentAsString )**

This function is basically the same as the previous function, however, in this case a string containing the contents of an libmoleculizer xml string are passed to the function.

**void moleculizer::addParameterStatement(const std::string& statement )**

This function allows the addition of a single parameter statement by passing in a string.

**void moleculizer::addModificationStatement( std::string& statement)**

This function allows adding a single modification statement to the model.

**void moleculizer::addMolsStatement( std::string& statement)**

This function allows adding a mzr-format molecule definition statement to the model.

**void moleculizer::addAllostericPlexStatement( std::string& statement)**

This function adds a single allosteric-plex mzr-format statement to be added to the model.

**void moleculizer::addAllostericOmniStatement( std::string& statement)**

This function adds a single allosteric-omni statement to the model.

**void moleculizer::addDimerizationGenStatement( std::string& statement)**

This function adds a single dimerization-gen statement to the model

**void moleculizer::addOmniGenStatement( std::string& statement)**

This function adds a single omni-gen reaction rule to the model.

**void moleculizer::addUniMolGenStatement( std::string& statement)**

This function adds a single uni-mol-gen reaction rule statement to the model.

**void moleculizer::addSpeciesStreamStatement( std::string& statement)**

This function adds a new species stream definition to the model.

### 4.1.3 Functions for looking at the state of libmoleculizer

**int moleculizer::getNumberOfDefinedModifications() const**

This function returns the number of modifications that have been defined in the current model.

**int moleculizer::getNumberOfDefinedMols() const**

This function returns the number of small-molecule and modifiable-molecules that have been defined in the current model.

**int moleculizer::getNumberOfDefinedRules() const**

This function returns the number of reaction rules that have been loaded into libmoleculizer

**int moleculizer::getNumberOfDimerReactionRules() const**

This function returns the number of association-reaction rules that have been defined in the current model.

**int moleculizer::getNumberOfOmniGenReactionRules() const**

This function returns the number of omni-gen reaction rules that have been defined in the current model.

**int moleculizer::getNumberOfUniMolReactionRules() const**

This function returns the number of uni-mol-gen reaction rules that have been defined in the current model.

**int moleculizer::getNumberOfPlexFamilies() const**

This function returns the number of distinct plex families (a plex family is a collection of species that are composed of the same molecules bound in the same way, but ignoring the modification states of the same).

### 4.1.4 Functions for expanding species and reactions

**void moleculizer::generateCompleteNetwork()**

Calling this function will generate the complete biochemical network that can be generated given the species identities already present with the loaded model.

Essentially this function keeps scanning the list of species, expanding unexpanding species, until nothing is left unexpanded. At this point, nothing new can be generated and the function returns. Note that many models generate infinitely large networks; others will simply generate gargantuan sized networks. In either of this cases this function will take a long time to run, if it ever, in fact, returns. Beware of calling this function on these sorts of models.

**CachePosition moleculizer::generateCompleteNetwork(long maxNumSpecies, long maxNumRxns = -1)**

This function generates a bounded network. That is, it will return enough information to describe a self-contained network with no more than maxNumSpecies and, if defined, no more than maxNumRxns. It does this by returning a CachePosition, which is defined as a std::pair, where the first value is a SpeciesList iterator and the second value is a ReactionList iterator.

There are two basic way to use this function. The first, easiest way is to use it immediately after loading a model. Then, the network consisting of all species from begin() to the first component of the CachePosition (the species list iterator) in the getDeltaSpeciesList() and all reactions from begin() to the second component of the CachePosition (the reaction list iterator) will be a bounded reaction network, that can be simulated in a self-contained manner.

The second way to use this function is to record all the current species and reactions present, clear the delta cache using resetCurrentState(), call the generateCompleteNetwork function, add the newly generated species and reactions (the new stuff in the delta species and delta reaction caches).

### 4.1.5 Utility Functions

**xmlpp::Document* moleculizer::makeDomOutput(bool verbose)**

This function causes moleculizer to write out the current state of moleculizer to an xml, whose structure is discusses elsewhere in the documentation. This function can be used with the loadXml* functions in libmoleculizer to resume state. By default, species are only listed with their tagged names in this output unless verbose is set to true, in which case all species are listed with both their tagged names.

**xmlpp::Document* moleculizer::makeDomOutput(bool verboseXML, CachePosition networkSizeRange )**

This function causes moleculizer to write out a bounded network to an xml file (after running generateCompleteNetwork() ). However for this to work properly, resetCurrentState() must not have been called during the program run.

**void moleculizer::writeOutputFile( const std::string& fileName, bool verbose = false)**

Pass this function the name of a file and it will write the current model to disk in a way that allows reloading later by the loadXmlFile function. This data includes not only the rule information present in the original model, but also the list of all species and reactions that have been generated. When verbose is true, the output will include the names of all the unique ids of the species, as opposed to merely the tagged names.

**void moleculizer::writeOutputFile( const std::string& fileName, bool verbose, CachePosition pos)**

This function is intended to be used with the generateCompleteNetwork(int,int) function (the one that generates a network whose size is bounded from above by the integer parameters into the file. By passing in a cache position piece of data, this function writes out the original model, as well as the state of the delta lists, up to the cache position.

**void moleculizer::getUserNames(std::vector<std::string>& refVector) const**

This function takes a reference to a (usually empty) vector of strings. It then inserts into that vector, copies of all the user names that have been defined.

**void moleculizer::recordUserNameToSpeciesIDPair( const std::string& userName, const std::string& speciesID )**

This function takes a user defined string along with a species ID and records this association. After calling this function, the user-name will be associated with that species/speciesID and can be used accordingly in all the associated functions.

**bool moleculizer::nameIsUserName( const std::string& possibleUserName) const**

Pass this function in a name, and it will return true if the name is a user-name and false otherwise. This is equivalent functionality to returning whether the string possibleUserName is contained in the vector of strings generated by the getUserNames function.

**std::string moleculizer::convertSomeNameToTaggedName(const std::string& name) const**

This function will take any kind of species name that libmoleculizer is aware of (either a user provided name, a tagged name, or a species ID, and will convert it to a tagged name. This function has been useful to some of the client programs of libmoleculizer, which, when adding libmoleculizer species into their simulation

model, use the species ID, if it is less than 256 characters, and use the tagged name otherwise. In these cases, rather than keeping track of where the name came from automatically, it was actually easier and faster to use this function, which simply returns the tagged name of the species, regardless of where the name came from.

**const mzrSpecies* moleculizer::getSpeciesWithSomeName( const std::string& name) const**

**mzrSpecies moleculizer::getSpeciesWithSomeName( const std::string& name)**

By passing in either a user-name, a species ID, or a a tagged name, these functions will return the associated species pointer, in const-correct form.

**mzrSpecies* moleculizer::getSpeciesWithTaggedName( const std::string& taggedName)**

**const mzrSpecies* moleculizer::getSpeciesWithTaggedName( const std::string& taggedName) const**

These functions take a tagged named as a parameter, and return a pointer to the accompanying species, in const-correct form. If no such tagged name has been recorded by the system, these functions will throw a fnd::NoSuchSpeciesXcpt.

### 4.1.6 mzrSpecies* moleculizer::getSpeciesWithUniqueID( const std::string& uniqueID )

### 4.1.7 const mzrSpecies* moleculizer::getSpeciesWithUniqueID( const std::string& uniqueID ) const

Both these functions take a unique id as their parameter and return a poiinter to the accompanying species, in const-correct form. If no such species exists, this function will attempt to create it and add it to simulation, returning the result. If the species does not exist, and then subsequently cannot be created, these functions throw a mzr::IllegalNameXcpt exception.

**void moleculizer::getSpeciesStreams( std::vector<std::string>& speciesStreamNames) const**

This function takes a reference to a (typically empty) vector of strings. It then inserts into that vector the names of each of the species-streams that have been recorded by libmoleculizer.

**int moleculizer::getNumberOfSpeciesStreams() const**

This function returns the number of species streams (called species classes in the rules) that have been recorded by libmoleculizer.

**int moleculizer::getNumberOfSpeciesInSpeciesStream(const std::string& streamName) const**

This function takes the name of a species class and returns the number of species that have been generated thus far which are a part of that species class.

**void moleculizer::getSpeciesInSpeciesStream(const std::string& streamName, std::vector<const mzr::mzrSpecies*>& speciesVector) const**

This function takes the name of a species class as its first parameter and a reference to a (typically empty) vector of const mzrSpecies pointers. This function then inserts pointers to each of the species within that species stream into the vector.

**bool moleculizer::speciesWithTagIsInSpeciesStream(const std::string speciesTag, const std::string& speciesStream ) const**

This function takes the tagged name of a species and the name of a species class and returns true if the species is a member of the species class and false otherwise. This function is the fastest way possible to answer this question.

**bool moleculizer::speciesWithUniqueIDIsInSpeciesStream(const std::string speciesTag, const std::string& speciesStream ) const**

This function takes the unique id of a species and the name of a species class and returns true if the species is a member of the species class and false otherwise. Although the "speciesWithTageIsInSpeciesStream" function is faster, this function is the fastest way possible to answer the question if only the species ID is possessed.

**int moleculizer::getMolCountInSpecies(const std::string& molName, const mzr::mzrSpecies* pSpec) const**

This function returns the number of molecules present in a particular species. Essentially this function uses dynamic_cast to convert the mzr::mzrSpecies* to a plx::mzrPlexSpecies*, and then inspects the structure of the mzrPlexSpecies, and is used largely as a convenience to the libmoleculizer library.

**int moleculizer::getMolCountInTaggedSpecies(const std::string& molName, const std::string& taggedName) const**

This function returns the number of molecules present in a particular tagged species, by finding the corresponding mzrSpecies* and calling getMolCountInSpecies on the pointer. This function is largely used as a convenience to the c-interface for this library.

### 4.1.8 Other

**bool moleculizer::recordSpecies( mzrSpecies*)**

This function is typically only used internally by libmoleculizer to record new species that have been created by expanding species and reactions, however is exposed to users. Pass a new species pointer into this function and it will be recorded by libmoleculizer. This function will return true if the species is new, and false if the species has already been recorded by libmoleculizer.

**bool moleculizer::recordSpecies( mzrSpecies*, std::string&)**

This function is typically only used internally by libmoleculizer to record new species that have been created by expanding species and reactions, however is exposed to users. Pass a new species pointer and a reference to a string into this function. If the species has not been recorded by libmoleculizer this function will record it. The species ID of the species will be placed into the reference to the string regardless, and the function will return true if the species has not been seen by libmoleculizer before, and false otherwise.

**void moleculizer::mustRecordSpecies( mzrSpecies* pSpecies ) throw( utl::xcpt )**

**void moleculizer::mustRecordSpecies( mzrSpecies* pSpecies, SpeciesTag& refName ) throw( utl::xcpt )**

These two functions are essentially the same as the previous two functions (void moleculizer::recordSpecies(mzrSpecies*) and void moleculizer::recordSpecies(mzrSpecies*, const std::string&) functions. However, they expect to be successful in their recording. Accordingly they throw exceptions if the species has already been recorded by libmoleculizer.

**bool moleculizer::recordReaction( mzrReaction* pRxn )**

The function is typically used internally by libmoleculizer to record reactions generated by the expansion of species, however is exposed to users just in case. This function records the reaction pRxn in its lists and always returns true.

**void moleculizer::mustRecordReaction( mzrReaction* pRxn ) throw( utl::xcpt )**

The function is typically used internally by libmoleculizer to record reactions generated by the expansion of species, however is exposed to users just in case. This function records the reaction pRxn in its lists and always returns. (Technically it will throw a fnd::DuplicatedCatalogEntryXcpt if the reaction has been recorded before, but for the time being, this always succeeds).

**mzrSpecies* moleculizer::findSpecies( const std::string& specTag ) throw( fnd::NoSuchSpeciesXcpt )**

**const mzrSpecies* moleculizer::findSpecies( const const std::string& specTag ) const throw( fnd::NoSuchSpeciesXcpt )**

These two functions are the fundamental functions for locating species. Pass them a string containing the tagged name of a species and these functions will return a pointer to that species in const-correct form. If no species with that tag is recorded these functions will throw a fnd::NoSuchSpeciesXcpt.

**bool moleculizer::checkSpeciesIsKnown( const std::string& speciesTag ) const**

This function will return true if a species with the given species tag has been recorded by libmoleculizer and false otherwise.

**bool moleculizer::findReactionWithSubstrates( const mzrSpecies* A, std::vector<ReactionTypeCptr>& reactionVector)**

This function is used to locate all unimolecular reactions involving a species A. To use it, pass in a species pointer to the species in question, and a reference to a vector of reaction pointers (mzrReaction*). If the species has not been expanded, it will be. Then, all unimolecular reactions involving A as a substrate will be inserted into the reaction vector. The function will return true if the vector is non-empty and false otherwise.

**bool moleculizer::findReactionWithSubstrates( const mzrSpecies* A, const mzrSpecies* B, std::vector<ReactionTypeCptr>& reactionVector)**

This function is used to locate all binary reactions in terms of two substrates, A and B. To use it, pass in species pointers involving the two substrates into this function along with a reference to a vector of reaction pointers (mzrReaction*). If either species is unexpanded, they will be expanded. Then, all binary reactions involving these two species will be inserted into the passed in reactionVector. The function will return true if the vector is non-empty and false otherwise.

**const std::list<mzrReaction*>& moleculizer::getReactionList() const**

This function will return a const reference to a list (std::list<mzr::mzrReaction*>) of all the reactions that have been recorded thus far by libmoleculizer.

**SpeciesCatalog& moleculizer::getSpeciesCatalog()**

**const SpeciesCatalog& moleculizer::getSpeciesCatalog() const**

These functions will return a reference to a SpeciesCatalog (a map<const std::string*, mzrSpecies*>, although the keys in the map are compared by the dereferenced

value of their strings). By using these functions, the entire list of generated species can be read out at any time.

**unsigned int moleculizer::getTotalNumberSpecies() const**

This function returns the total number of species that are in the network at any given time.

**unsigned int moleculizer::getTotalNumberReactions() const**

This function returns the total number of reactions that are in the network at any given time.

**unsigned int moleculizer::getNumberDeltaSpecies() const**

This function returns the number of species in the delta species list at any given time.

**unsigned int moleculizer::getNumberDeltaReactions() const**

This function returns the number of reactions in the delta reactions list at any given time.

**const SpeciesList& moleculizer::getDeltaSpeciesList() const**

This function returns a constant reference to the delta species list, of type std::list<mzr::mzrSpecies*>. This is a list of all species that have been generated since the last time moleculizer::resetCurrentState() has been called, and can be used to incrementally generate and read out portions of the network.

**const ReactionList& moleculizer::getDeltaReactionList() const**

This function returns a constant reference to the delta reactions list (of type std::list<mzr::mzrReaction*>). This is a list of all the reactions that have been generated since the last time moleculizer::resetCurrentState() has been called, and can be used to incrementally generate and read out portions of the network.

**void moleculizer::resetCurrentState()**

This function clears the lists of delta species and delta reactions.

**void moleculizer::incrementNetworkBySpeciesTag( const SpeciesTag& rName ) throw( utl::xcpt )**

This function expands the species with the provided species tag. If the species has already been expanded, this will do nothing. However, if it has not been expanded, it will likely cause species and reactions to be generated, each of

which will be placed in the complete lists of species/reactions as well as the delta lists of species/reactions.

**std::string moleculizer::convertSpeciesTagToSpeciesID( const SpeciesTag& rTag ) const throw( utl::xcpt )**

This function will take a species tag and covert it to a species ID. It will throw a NoSuchSpeciesXcpt if a species with the species tag cannot be found.

**std::string moleculizer::convertSpeciesIDToSpeciesTag( const std::string& rID) const throw( utl::xcpt )**

This function will take a species ID and convert it to a species tag. It will throw a NoSuchSpeciesXcpt if a species with the supplied species ID cannot be found.

## 4.2 The C Interface

If your program is written in C, you should use the C interface. To make use of the c-interface, include the file "libmoleculizer/mzr/libmzr_c_interface.h".

### 4.2.1 Data Types

There are three main data types used by the libmoleculizer c-interface, and each of them are discussed in this section.

**moleculizer***

The moleculizer*, referred in the interface as a moleculizer handle, is the main data type used, although it acts almost completely as a black box. A moleculizer* must be created using the createNewMoleculizerObject function and freed using the freeMoleculizerObject function. It is required as the first parameter to every function in the interface, but otherwise cannot be interacted with in any meaningful way.

**species**

This structure represents a moleculizer species. A species structure contains four sub-elements: a char* name (containing the species' tag), a double* mass (this represents the species mass, although is meaningful only if the appropriate masses for each of the molecules have been set in the rules), a double* radius (this was added for Smoldyn – it estimates the radius based on the species' mass, an assumption it is globular, and the average density for protein), and a double* diffusion coefficient (this is also Smoldyn based, and simply returns 3.0 for molecules with masses greater than 1500 and 100.0 for species with masses less than 1500).

Mostly, the species structure is used to read out its tagged name.

**reaction**

This structure represents a moleculizer reaction. The reaction structure consists of a char* name, consisting of an automatically generated and unique name for the reaction, an int numberReactants and an int numberProducts which describe the number of products and reactants that make up that reaction, a species** reactantVector (an array of species pointers making up the reactants), a species** productVector (an array of species pointers making up the products), and a double* rate, which describes the rate of the reaction.

If mass-based reaction rate extrapolation is used, the basic reaction rate multiplied by the reduced mass of the reactant species is used here. If it is not used, the basic kinetic rate supplied to the reaction rule that generated this reaction is what is used.

### 4.2.2 Functions

**moleculizer* createNewMoleculizerObject()**

This function is the first function that must be called to use libmoleculizer using c. Simply call the function and it returns a pointer to a newly allocated moleculizer*. This pointer is then used as the first parameter to all other c-interface functions. The pointer is owned by the caller. Hang onto it and free it using the freeMoleculizerObject function, lest memory leaks.

**void freeMoleculizerObject( moleculizer* handle)**

Simply pass in a pointer to a moleculizer structure, as allocated by the createNewMoleculizerObject function, and all its memory will be released back to the operating system. Remember, don't you dare use the pointer after passing it to this function, or you'll (best case) segfault the program.

**int setRateExtrapolation( moleculizer* handle, int extrapolation)**

This function should be used prior to loading a set of rules. As with all other functions in this interface, it takes a pointer to a moleculizer object allocated using "createNewMoleculizerObject". It also takes an integer parameter. If the parameter is non-zero, rate extrapolation is set on the moleculizer* object. If set to zero, rate-extrapolation is turned off.

**int loadCommonRulesFile(moleculizer* handle, char* fileName)**

This function takes a moleculizer handle as well as the filename of a common-rules ".mzr" file. This function loads and parses that file and adds its information into the handle object. The function returns 0 on success; non-zero otherwise (1 = Unknown error, 2 means the document was unparsable, 3 means the moleculizer handle has already had a set of rules loaded into it; 4 means the file could not be found ).

**int loadCommonRulesString( moleculizer* handle, char* file)**

This function is quite similar to the loadCommonRulesFile function. However, instead of passing in a c-string containing the name of a file, this function takes in a c-string containing the file contents directly and loads it into the handle. The function returns 0 on success; non-zero otherwise (1 = unknown error; 2 = document unparsable; 3 = rules have already been loaded into the handle object).

**int loadXMLRulesFile(moleculizer* handle, char* fileName)**

Typically not used, this function takes a filename containing moleculizer rules in xml format and loads them. (Note that the xml format here used to be the old moleculizer format; those old files can be loaded in using this function. Additionally, the new common-rules format .mzr files are compiled internally into xml, which can be loaded using this fule). This function returns 0 on success and non-zero otherwise( 1 on unknown error, 2 if the document is unparsable, 3 if rules have already been loaded into the handle, and 4 if the file was not found).

**int loadXMLRulesString( moleculizer* handle, char* file)**

Typically not used (XML input is depreciated in favor of the new mzr rules format (mzr files)), this function takes a moleculizer handle and a char* containing the contents of an xml rules file and loads that information into the handle. This function returns 0 on success, 1 on unknown error, 2 if the document passed in was not parsable, and a 3 if rules have already been loaded into the handle.

**int expandNetwork( moleculizer* handle)**

This function takes a moleculizer handle and fully expands the network, by continually scanning the list of species and reactions, and expanding them until nothing is left to expand. BEWARE: for either infinitely large networks, or even for many large but finite networks, this function make take a very long time to return if ever, so use with some caution (or use the getBoundedNetwork function instead). This function returns 0 on success, 1 for an unknown error, and 2 if no model has been loaded.

**int getBoundedNetwork( moleculizer* handle, long maxNumSpecies, long maxNumReactions, species*** pSpeciesArray, int* pNumSpec, reaction*** pReactionArrry, int* pNumRxns)**

This function takes a moleculizer handle, a number indicating a maximum number of species and a maximum number of reactions (the maximum number of reactions may be negative, in which case only the maximum number of species is used), as well as addresses of a species array, an integer holding the number

of species returned, an address of a array of reaction pointers, and an address of an integer that will hold the number of reactions.

**int incrementSpecies( moleculizer* handle, char* speciesTag)**

This function takes a moleculizer handle as well as a species tag and expands that species, possibly generating new species and new reactions. The function returns 0 on success, 1 for an unknown error, and 2 if the speciesTag is unknown to the moleculizer object.

**int expandSpeciesByTag( moleculizer* handle, char* theTag)**

This function takes a moleculizer handle as well as a species tag and expands that species, possibly generating new species and new reactions. The function returns 0 on success, 1 on failure.

**int expandSpeciesByID( moleculizer* handle, char* theID)**

This function takes a moleculizer handle as well as a species ID and will expand that species. The function returns 0 on success and 1 on failure.

**int getNumberOfSpecies(moleculizer* handle)**

This function returns the number of species that have been generated in the reaction network contained in the moleculizer handle.

**int getNumberOfReactions(moleculizer* handle)**

This function returns the number of reactions that have been generated in the reaction network contained in the moleculizer handle.

**int getDeltaSpecies( moleculizer* handle, species*** pSpeciesArray, int* pNum)**

Pass this function a moleculizer handle, as well as the address of a species pointer array and the address of an int. The function copies all the delta species contained in the handle and copies them into the array of species pointers and copies the size of that array into the pNum. The species array is then owned by the caller and can be iterated over and its information read out. The species array must be freed using the freeSpeciesArray function. The function returns 0 on success and 1 on unknown failure.

**int getDeltaReactions( moleculizer* handle, reaction*** pReactionArray, int* pNum)**

Pass this function a moleculizer handle, as well as the address of a reaction pointer array and the address of an int. The function copies all the delta reactions currently contained in the handle into the reaction pointer array and

puts the number of reactions copies into pNum. The reaction array is owned by the caller and can be iterated over so that its information can be read out. The reaction array must be freed using the freeReactionArray function. The function returns 0 on success and 1 on unknown failure.

**int clearDeltaState( moleculizer* handle)**

This function takes a moleculizer handle and clears the delta caches (the delta species array and the delta reaction array). This function returns 0 on success and 1 on unknown failure.

**int getAllSpecies(moleculizer* handle, species*** pSpeciesArray, int* numberSpecies)**

This function takes a moleculizer handle, the address of a species pointer array and the address of an int. It then copies every species that has been generated into the species array and copies the number of species into numberSpecies. The resulting array is owned by the caller and must be freed using the freeSpeciesArray function. The function returns 0 on success and 1 on unknown error.

**int getAllReactions(moleculizer* handle, reaction*** pReactionArray, int* numberReactions)**

This function takes a moleculizer handle, the address of a reaction pointer array and the address of an int. It then copies every reaction that has been generated into the reaction pointer array and the number of reactions into numberReactions. The resulting array is owned by the caller and must be freed using the freeReactionArray function. The function returns 0 on success and 1 on unknown error.

**int getNumModificationDefs( moleculizer* handle)**

This function takes a moleculizer handle and returns the number of modifications that have been defined in the rules.

**int getNumMolDefs( moleculizer* handle)**

This function takes a moleculizer handle and returns the number of molecule types that have been defined in the rules.

**int getNumReactionRules( moleculizer* handle)**

This function takes a moleculizer handle and returns the number of reaction rules (association-reactions and enzymatic-reactions, consisting of all dimer-gens, omni-gens, and uni-mol-gens) in the program.

**int getNumDimerDecompReactionRules( moleculizer* handle)**

This function takes a moleculizer handle and returns the number of association reactions (dimer-gens) defined in the rules.

**int getNumOmniGenReactionRules( moleculizer* handle)**

This function takes a moleculizer handle and returns the number of enzymatic reactions (omni-gens) defined in the rules.

**int getNumSpeciesStreams( moleculizer* handle)**

This function takes a moleculizer handle and returns the number of species-streams that have been defined in the model.

**int getReactionsBetween(moleculizer* handle, char* speciesID2, char* speciesID2, reaction*** ptrReactionPtrArray, int* numReactions)**

This function takes a moleculizer handle as well as the unique ids of two species (they can be equal, which will find all reactions of the form A + A -> ? ). By also passing in a pointer to an array of reaction pointers as well as an address of an int, this function will search for all reactions that involve the two species as substrates and place all of them into the reaction pointer array, and place their number into the numReactions pointer. The reaction pointer array is owned by the caller and must be freed using the freeReaction array function. This function returns 0 on success, 1 on unknown error, and 2 if one of the species has an illegal name.

**int getUnaryReactions(moleculizer* handle, char* speciesID, reaction*** ptrReactionPtrArray, int* numReactions)**

Use this function to find all unary reactions involving a particular substrate. To use it, pass in a moleculizer handle, the species ID of the species concerned, and the address of a pointer of reaction pointers and the address of an int. This function finds all unary reactions that involve the substrate and copy them into the array of reactions, and copy the number of reactions found into the numReactions parameter. The returned array is owned by the caller, and must be freed using the freeReactionArray function. This function returns 0 on success, 1 on unknown error, and 2 if the species name passed in is illegal.

**int getAllExteriorSpecies(moleculizer* handle, species*** pSpeciesArray, int* numberSpecies)**

This function takes the standard moleculizer handle, an address of an array of species pointers, and an address of an int, and copies in all unexpanded species into the species array and their number into the numberSpecies parameter. As always, the memory returned is owned by the user and should be freed using the

freeSpeciesArray function. The function returns 0 on success and 1 on unknown failure.

**int getMolCountInTaggedSpecies(moleculizer* handle, const char* molName, const char* speciesTag)**

This function takes the name of a molecule type and a species tag, and returns the number of that type of molecule that are found in the speciesTag. The function returns -1 if there is an error of some kind.

**void getAllSpeciesStreams( moleculizer* handle, char*** speciesStreamArray, int* numberSpeciesStreams)**

Use this function to get the names of all the different species streams that have been registered in a moleculizer handle. To use it, pass in a moleculizer handle and an address of an array of character pointers as well as an address to an int. This function copies out all the names of the species streams into the array and copies their number into the numberSpeciesStreams parameter. As always, this memory is owned by the user and should be freed using the freeCharPtrArray function.

**int getAllStreamSpecies(moleculizer* handle, char* streamName, species** pSpeciesArray, int* numberSpecies)**

This function takes a moleculizer handle, the name of a species stream, and addresses to a species pointer array and an int. The function takes the tagged names of all the species in that species stream and places them into the array, and their number into numberSpecies. The function returns 0 on success, 1 on an unexpected error, and 2 if the species stream does not exist.

**int checkSpeciesTagIsInSpeciesStream( moleculizer* handle, const char* speciesTag, char* speciesStream)**

This function takes a moleculizer handle, the name of a species tag, and the name of a species stream. It returns 0 if not, 1 if so. A -1 means an unknown error, a -2 if the species does not exist, and a -3 if the species stream does not exist.

**int checkSpeciesIDIsInSpeciesStream( moleculizer* handle, const char* speciesID, char* speciesStream)**

This function takes a moleculizer handle, the name of a species ID, and the name of a species stream. It returns 0 if not, and 1 if true. A -1 is returned if there is an error: either the species ID or the species stream does not exist.

**int convertTaggedNameToUniqueID( moleculizer* handle, char* speciesTag, char* speciesID, unsigned int idSize)**

Use this function to convert a species tag into a species ID. Pass in a moleculizer handle, a species tag, as well as a character buffer that will hold the species ID and an integer parameter that holds the size of the buffer. The function returns 0 on success, a 1 on unknown error, a 2 if the species tag cannot be found in the moleculizer handle, and a 3 if the character buffer is not large enough to hold the resulting species ID.

**int convertUniqueIDToTaggedName( moleculizer* handle, char* speciesID, char* speciesTag, unsigned int tagSize)**

Use this function to convert a species ID into a species tag. Pass in a moleculizer handle, a species tag, as well as a character buffer that will hold the species tag and an integer parameter that holds the size of the buffer (the size of species tags is fixed on any given system, but should never be larger than 15 characters). The function returns 0 on success, a 1 on an unknown error, a 2 if the species ID is unknown to libmoleculizer, and a 3 if the character buffer is too small to hold the resulting species tag.

**int convertUserNameToTaggedName(moleculizer* handle, char* theUserName, char* correspondingTag, unsigned int bufferSize)**

Use this function to convert an explicit, user name to a tagged name. As other similar functions, it takes the moleculizer handle and the user name as parameters, as well as a character buffer and an integer parameter holding the size of the buffer. The function returns 0 on success, 1 on unknown error, 2 if there is no such explicit species name registered, and 3 if the buffer isn't large enough to hold the resulting tag.

**int convertUserNameToUniqueID(moleculizer* handle, char* theUserName, char* correspondingSpeciesID, unsigned int bufferSize)**

This function works identically to the convertUserNameToTaggedName function, except that it returns a species ID. Otherwise everything else is the same. The function returns 0 on success, 1 on unknown error, 2 if there is no such user name registered, and 3 if the buffer isn't large enough to hold the resulting ID.

**int convertSomeNameToTaggedName( moleculizer* handle, char* theName, char* taggedNameBuffer, unsigned int bufferSize)**

This function works similarly to other functions, except in this case it can take (with associated performance penalty) any name as its second parameter. If the name is found, either as a tagged name, an explicit name, or a species ID, it is converted to a tag and that tag is copied into the taggedNameBuffer. If successful, the function returns 0. It also returns 1 on unknown error, 2 if the

name cannot be found, and 3 if the tagged name was found but is too large to copy into the taggedNameBuffer.

**int getExplicitSpeciesList(moleculizer* handle, char*** theExplicitSpeciesNames, unsigned int* numSpecies)**

Use this function to get the list of all registered explicit species. Pass it in a moleculizer handle, the address of an array of character pointers and the address of an int. This function will copy over all the explicit names into the array and copy their number into numSpecies. The resulting array of strings is owned by the user and should be freed using the freeCharPtrArray function. The function returns 0 on success and 1 on an unknown error.

**void freeReactionArray( reaction** pRxnArray, unsigned int numArrayElements)**

Pass this function a reaction array and the number of elements in that array and everything is freed.

**void freeSpeciesArray( species** pSpeciesArray, unsigned int numArrayElements)**

Pass this function a species array and a number of elements in that array and everything is freed.

**void freeCharPtrArray( char** pCharArray, unsigned int numCharPtrElements)**

Pass this function an array of strings (the memory should all be owned by the user, as is always the case for libmoleculizer functions) and the number of elements in that array and all the associated memory is freed.

**void freeReaction( reaction* pRxn )**

This function frees a single reaction pointer. It is mostly used internally by the freeReactionArray function.

**void freeSpecies( species* pSpecies )**

This function frees a single species pointer. It is mostly used internally by the freeSpeciesArray function.

# Chapter 5

# Installing Libmoleculizer

## 5.1 Compiling and installing from source

The libmoleculizer package uses the Gnu Autotools to setup the build (this is the familiar configure, make, and make install style compilation) and should be fairly easy to install on a wide variety of platforms.

### 5.1.1 Prerequisites

Libmoleculizer has a single prerequisites which are necessary to compile and run libmoleculizer, a version of the libxml++ library. Any version should be able to be detected and used properly with libmoleculizer.

The easiest way to install libxml++ is to use your systems package manager, if you have one, which will ensure that all the necessary files are installed and configured properly, in an automatic fashion. For instance, on a Ubuntu linux system, entering the command "sudo apt-get install libxml++2.6-dev" or "sudo apt-get install libxml++1.0-dev", depending on which version you prefer (version 1.0 is simpler and smaller, version 2.6 is larger and has more dependencies, but has unicode support). On a Fedora linux system, the command "sudo yum install libxml++" should work.

On Macintosh systems, there is no standard package manager that comes with the system. MacPorts (http://www.macports.org/) and Fink (http://www.finkproject.org/) are installable package managers for Mac systems, both of which provide libxml++. Using Fink, either of the commands "sudo fink install libxml++1" or "sudo fink install libxml++2" will work, depending on which version you wish to install (1 is much smaller; 2 has unicode support). Using MacPorts, the same corresponding commands would be either "sudo port install libxmlxx" or "sudo port install libxmlxx2", which will install either version 1 or 2 respectively.

**Manually compiling and installing libxml++**

If you do not have a package manager, or just wish to go it alone, the next option is to compile libxml++ from source. Source packages for libxml++ can be obtained from the downloads page of the libxml++ webisite, which can be found at http://libxmlplusplus.sourceforge.net// If this is the case, using version 1.0, as opposed to any of the 2.x versions, is HIGHLY recommended, as libxml++-1.0 has a single prerequisite, whereas libxml++-2.x has legion. Installing 2.x from source is possible, and not terribly tricky, but is very time consuming, and will not be discussed here.

To install libxml++-1.0 from scratch, another library, libxml2, must first be compiled and installed. As it may already be installed on your system, attempt to run ./configure from within the libxml++-1.0 directory. If it works, great: followup with running make and then make install and you're done. If not, follow the steps in the next paragraph to ensure libxml2 is installed.

Libxml2 (http://xmlsoft.org/) can be downloaded from ftp://xmlsoft.org/libxml2/, At the time of this writing, the latest version is 2.7.3. It has no prerequisites and so simply running ./configure, make, and make install should be enough to install libxml2.

Once this is done, libxml++-1.0 can be compiled, using a similar ./configure, make, and make install procedeure.

After this is done, all the necessary preconditions for installing libmoleculizer have been satisfied.

### 5.1.2 Simplest compile/installation Procedure

Depending on your system configuration, the following procedure will probably be all that is necessary to get a working version of libmoleculizer onto your system.

First, obtain the source. The best option for this is to go to *http://sourceforge.net/projects/moleculizer/* and download the libmoleculizer-1.1.3.tar.gz source file, and make sure it is unpacked.

Assuming everything is setup, and the prerequisites are met, then installation will likely be as easy as going into the source directory and entering the following commands, which will compile the source code into an executable and install it in a place where it is globally accessable.

```
./bootstrap.sh
./configure
make
sudo make install
```

### 5.1.3 Can't or Dont want to install libmoleculizer globally

By default **./configure** installs libmoleculizer to a global location, usually /usr/lib or /usr/local/lib depending on your system. This may either be unacceptable or impossible, say if you do not have administrator access on your

computer. This can be fixed py passing a –**prefix=*location*** flag to the **./configure** command.

### 5.1.4 ./configure cannot find necessary libraries

Configure may fail and say that libxml++ cannot be found. The first thing to do, of course, is make sure it is installed on your machine. If it isn't, that's the problem. Follow the directions in the Prerequisites section and repeat.

If it is installed, the likely problem is that pkg-config cannot find libxml++.

If they are, then configure must be explicitly told where to find the libraries and/or the corresponding header files. The easiest way to do this is to pass the header file locations to the CXXFLAGS environmental variable, and the library locations to the LDFLAGS environmental variable.

### 5.1.5 Enabling the demos

Libmoleculizer has several demos that come with it, that demonstrate in principle how to design a particle simulator using libmoleculizer (particlesim_demo), how to design a stochastic simulator a la Gillespie using libmoleculizer (stochasticsim_demo), how to use libmoleculizer to expand reaction networks (network_expander_demo), and how to use the c interface (c_interface_demo). These examples are located in the doc/demos/simulators/ directory, however, by default, they are neither built nor installed.

If you desire to use these programs (recommended when learning to use libmoleculizer), add the flag –enable-demos when calling configure, i.e. run "./configure –enable-demos".

# Chapter 6

# Cookbook

This chapter contains explicit examples of many standard model constructs that users may find useful.

Note that in the examples in this chapter, certain information needed in the model may not be explicitly listed. For instance, in an example of a dimerization reaction, the mod-mols involved may not be explicitly defined, although they would need to be in an explicit model.

## 6.1 Defining Explict Modifications

Most models of signal transduction pathways will include modifications; at the least, modifications representing phosphorylation groups are often added. A 'None' or 'Null' modification almost needed as well to represent a modification that doesn't exist: an 'un-phosphorylated' or 'un-ubiquitinated' or generally 'un-modified' state.

6.1 gives an example of a simple modification section that defines one phosphorylation section.

Listing 6.1: 'Example of a simple modification section'

```
<modifications>
  <modification name='None'>
    <weight-delta daltons='0.0' />
  </modification>
  <modification name='phosphorylated'>
    <weight-delta daltons='42.0'/>
  </modification>
</modifications>
```

## 6.2 Defining Abstract Modifications

It is sometimes adventageous to define modifications that do not explicitly correspond to physical modifications. For instance, when modeling a doubly-phosphorylatable kinase target, it may be desirable to model the target as having only one modification site that can be in a 'singly-phosphorylated' or 'doubly-phosphorylated' state. This can simplify reaction generation at the expense of some physical explicitness.

Using this simplified and abstractified system, only two reaction generators must be written, which take the target from 'None' to 'phosphorylated', and another from 'phosphorylated' to 'doubly-phosphorylated'.

```
<modifications>
  <modification name='None'>
    <weight-delta daltons='0.0' />
  </modification>
  <modification name='phosphorylated'>
    <weight-delta daltons='42.0'/>
  </modification>
  <modification name='doubly-phosphorylated'>
    <weight-delta daltons='84.0'/>
  </modification>
</modifications>
\end{lstlisting}

\section{Defining a binding-site on a mod-mol}
\section{Defining an allosteric form of a binding-site}
\section{Defining a site that has different behavior based on the
  state of the containing protein}
\section{Defining a binding site that has different behavior based on
  more complicated factors}
\section{Defining a simple dimerization reaction}

The most general kind of reaction generator in libmoleculizer is the
dimerization reaction, which defines a binding reaction between two
proteins.

In its simplest form, it can be written as follows.

\section{Defining a dimerization reaction with allosteric kinetics}
\section{Defining a modification state decomposition}
A user may wish to express that a particular modified protein will
decompose into an unmodified protein at some particular rate.

This can be done using a uni-mol-gen.
```

```
\begin{lstlisting}
<uni-mol-gen>
  <enabling-mol name='Protein' />
  <enabling-modifications>
    <mod-site-ref name='PhosphorylationSite'>
      <mod-ref name='phosphorylated' />
    </mod-site-ref>
  </enabling-modifications>
  <modification-exchanges>
    <modification-exchange>
      <mod-site-ref name='PhosphorylationSite' />
      <installed-mod-ref name='None' />
    </modification-exchange>
  </modification-exchanges>
  <rate value='1e5' />
</uni-mol-gen>
```

## 6.3 Defining a Complicated Reaction

# Chapter 7

# API Reference

This chapter lists the functions that occurs in each of the APIs and documents them in a flat style.

## 7.1 C++ API Reference

Note that all these functions are contained in the mzr namespace. Therefore to use libmoleculizer, either a "using namespace mzr" directive must be used, or the "mzr::" namespace must be appended to certain commands.

### 7.1.1 A short word on including, compiling, and linking

To use this software, the file libmoleculizer/mzr/moleculizer.hh must be included. However, all header files are installed to $Prefix/libmoleculizer-1.1.3, where $Prefix is the base installation directory, e.g /usr or /usr/local. Therefore you will have to add the compiler flags "-I$Prefix/libmoleculizer-1.1.3" to your compiler flags. While this can be done manually, a much, much easier method is to use pkg-config. While you should consult the pkg-config man page for more information, if you simply add the results of "pkg-config libmoleculizer-1.1.3–cflags" to your compilation step, and "pkg-config libmoleculizer-1.1.3–libs" to your linking step, everything will be taken care of automatically.

Conclusion: add #include "libmoleculizer/mzr/moleculizer.hh" to any files that use libmoleculizer, add the flags listed in "pkg-config libmoleculizer-1.1.3–cflags" to your compilation steps, and add the flags listed in "pkg-config libmoleculizer-1.1.3–libs" to your linking steps.

### 7.1.2 The moleculizer class

The major object of libmoleculizer is the moleculizer class, which represents a reaction network defined by a set of rules.

**moleculizer::moleculizer(void)**

This is the one and only constructor of moleculizer.

**void moleculizer::setGenerateDepth( unsigned int generateDepth )**

This function sets the generation depth. The value of the generation depth determines the behavior of expanding species and reactions. When the function expandReactionNetwork is called on either a reaction or a species, the network will be expanded to that many levels of generationDepth. If the generationDepth is set to 0, no expansion and no species/reaction generation will ever be done. If the generationDepth is 1 species and reactions will be generated, but they themselves will not be expanded. If the generation depth is 2, species and reactions will be generated, and each of those will themselves be expanded, however the species and reactions they generate will not be expanded. A generation depth of 3 will expand those, but not the next level, and so on. **The default generation depth is 1.** Setting a generation depth that is too high may impede performance.

**int moleculizer::getGenerationDepth( void ) const**

Calling this function will return the generation depth as currently set in a moleculizer object.

**void moleculizer::setRateExtrapolation( bool rateExtrapolation )**

This function sets rate extrapolation for the moleculizer object. Accordingly, it must be set prior to loading a model; if it is not, this function throws a mzr::modelAlreadyLoadedXcpt exception. If it is set to false, the reaction rate for a created reaction is the same as the reaction rate supplied in the reaction rule. If it is set to true and the generated reaction is binary, the reaction kinetics supplied in the rules will be multiplied by the reduced masses of the two species: $\sqrt{\frac{m_1 * m_2}{m_1 * m_2}}$. Please see 2 for more details. The default property of moleculizer is to not use reaction rate extrapolation.

**bool moleculizer::getRateExtrapolation() const**

This function returns the current value of the rate extrapolation.

**void moleculizer::attachFileName( const std::string& fileName)**

This function takes a string consisting of the filename of a mzr rules file. Calling it reads the file and loads it into the moleculizer object. If a file has already been loaded into moleculizer, this will throw a utl::modelAlreadyLoadedXcpt exception. If the file does not exist, or otherwise cannot be read or parsed, this function will throw a utl::dom::noDocumentParsedXcpt.

**void moleculizer::attachString( const std::string& fileString)**

Calling this function with a string containing the contents of mzr file will cause it to be loaded into moleculizer. If a file has already been loaded into moleculizer, this will throw a utl::modelAlreadyLoadedXcpt exception. If the string cannot be parsed, this function will throw a utl::dom::noDocumentParsedXcpt exception.

**void moleculizer::attachDocument( xmlpp::Document* pDoc)**

Calling this function with a document that has already been parsed using libxml++ will load it into the model. *It is not expected that this function will be generally used by users.* If a model has already been loaded into moleculizer, this function will throw a utl::dom::noDocumentParsedXcpt.

**virtual void moleculizer::generateCompleteNetwork()**

Calling this function will cause the moleculizer object to generate the complete reaction network (i.e. continue expanding species and reactions until no new species and reactions can be generated according to the rules). If a model has not yet been loaded, this function will throw a mzr::ModelNotLoadedXcpt exception. **Beware that this function may take a very long time to run, if it returns at all.**

**CachePosition generateCompleteNetwork(long maxNumSpecies, long maxNumRxns = -1)**

Use this function to generate a network that is bounded by maxNumSpecies and maxNumRxns (if a maxNumRxns parameter is specified). This function returns a CachePosition, which is a std::pair, where the first entry is an iterator to a position in the deltaSpeciesList of type std::list¡mzr::mzrSpecies*¿::iterator and the second entry is an iterator to a position in the deltaReactionList of type std::list¡mzr::mzrReaction*¿::iterator. By iterating from theDeltaSpeciesList.begin() to the CachePosition.first, and by iterating from the DeltaReactionList.end() to the CachePosition.second, a complete network which possesses fewer than maxNumSpecies and (if specified) fewer than maxNumRxns.

**bool moleculizer::getModelHasBeenLoaded() const**

This function returns true if a model has been loaded into the moleculizer object and false otherwise.

**int moleculizer::getNumberOfPlexFamilies() const**

A plex family in moleculizer is a structurally identical collection of species. That is, for a given complex species, its corresponding plex family is the set of all complex species that only differ in terms of their modification values. This function returns the number of plex species in moleculizer.

**void moleculizer::getSpeciesStreams( std::vector¡std::string¿& speciesStreamNames) const**

This function takes a reference to a vector of strings (semantically the vector probably should be empty, but does not have to be), and places into it the names of each of the species streams (see Chapter 2 for a discussion of species streams) that have been defined in the rules files.

**int moleculizer::getNumberOfSpeciesInSpeciesStream( const std::string& speciesStream) const**

This function returns the number of species that are present in a particular species stream. I.e. if a species stream "Species containing an A bound to B" has been defined in the rules file, calling getNumberOfSpeciesInSpeciesStream("Species containing an A bound to B") will return the number of species matching that criterion.

**void moleculizer::getSpeciesInSpeciesStream(const std::string& streamName, std::vector¡const mzr::mzrSpecies*¿& speciesVector) const**

This function takes the name of an existing species stream as well as a reference to a vector of const mzr::mzrSpecies*. It inserts into the vector pointers to each of the species generated thus far that match the conditions of the species stream.

**void moleculizer::getSpeciesInSpeciesStream(const std::string& streamName, std::vector¡mzr::mzrSpecies*¿& speciesVector)**

This function does the same thing as "void moleculizer::getSpeciesInSpeciesStream(const std::string& streamName, std::vector¡const mzr::mzrSpecies*¿& speciesVector) const", except that it does it with a vector of mzrSpecies*, as opposed to const mzrSpecies*. Use this if you wish to expand any of the species, as opposed to just recording information about them.

**const mzrSpecies* moleculizer::getSpeciesWithName( const std::string& speciesName )**

This function takes a unique id and returns a pointer to a const mzrSpecies that corresponds to that name. This function will first look up the name in the list of species and reactions. If it finds it, it will return it. If it does not, it will construct it from scratch (note that it will do so assuming coherency – that the mols that occur within the name also exist in the rules and have the same definitions). Therefore, this is the function to be used when dealing with serialized data.

**void moleculizer::recordUserNameToGeneratedNamePair( const std::string& userName, const std::string& genName )**

This function takes two string parameters. The first is a user provided name, the second is the unique id of a species that must already exist in the network. This function will allow the species to thereafter be looked up using the user name. This function throws a fnd::NoSuchSpeciesXcpt if the species does not exist.

**bool moleculizer::nameIsUserName(const std::string& possibleUserName) const**

This function takes a string corresponding to a potential user name, and returns either true or false depending on whether or not the string parameter has been recorded as a user-name.

**std::string moleculizer::convertUserNameToGeneratedName(const std::string& possibleUserName) const**

This function takes a string parameter corresponding to a user name and returns the generated unique id of the corresponding species. This function throws a mzr::unknownUserNameXcpt exception if the user name has not been registered.

**const mzrSpecies* moleculizer::findSpecies( const std::string& name) const**

This function takes a string containing a unique id and returns a corresponding pointer to a const species. If the species does not exist, the function throws a fnd::NoSuchSpeciesXcpt exception.

**mzrSpecies* moleculizer::findSpecies( const std::string& name)**

This function takes a string containing a unique id and returns a corresponding pointer to the species. If the species does not exist, the function throws a fnd::NoSuchSpeciesXcpt exception.

**bool moleculizer::findReactionWithSubstrates(const mzr::mzrSpecies* pSpecies, std::vector¡const mzr::mzrReaction*¿& rxnVector)**

This function is the way to find all unary reactions that a particular species participates in. Pass in the species of interest by pointer, as well as a reference to a vector of const mzrReaction pointers. Moleculizer will insert into that vector the set of all unary reactions that have the species as their one and only substrate. The function returns true if at least one const mzrReaction* is inserted into the vector and false otherwise – i.e. the function returns whether any such reactions were found.

**bool moleculizer::findReactionWithSubstrates(const mzr::mzrSpecies* spec1, const mzr::mzrSpecies* spec2, std::vector¡const mzr::mzrReaction*¿& rxnVector)**

This function is the way to find all binary reactions that a particular set of reactions participates in. To use this function, pass in pointers to the two species of interest (if you are interested in finding all reactions of the form A + A -¿ ? for some species A, simply pass in the pointer to species A twice) as well as a reference to a vector of const mzr::mzrReaction pointers. This function will insert pointers to each binary reaction that have both spec1 and spec2 as substrates into the vector reference and return true if there are reactions between these two species and false otherwise.

**bool moleculizer::checkSpeciesIsKnown( const std::string& speciesName ) const**

This function takes a string parameter corresponding to a unique id and returns whether or not this a species with this unique id has been recorded with moleculizer.

**const SpeciesCatalog& moleculizer::getSpeciesCatalog() const**

Every species that has been generated during the current run of libmoleculizer is stored in a data structure of type SpeciesCatalog, which is defined as a std::map¡std::string*, mzr::mzrSpecies, aux::compareByPtrValue¡mzrSpecies*¿ ¿. [1]. This datastructure maps pointers to strings containing unique ids (as opposed to tags, although this may change in a future release).

Calling this function returns a const reference to this data structure.

**SpeciesCatalog& moleculizer::getSpeciesCatalog()**

Does the same thing as the non-const version of this function, except that in this version, the species in the species catalog can be manipulated – although the only manipulation you do should be to expand them. Anything else will likely result in ill-defined behavior at best, segfault at worst.

**const std::list¡const mzr::mzrReaction*¿& moleculizer::getReactionList() const**

This function returns a const reference to a list containing every reaction that has been generated during this session of moleculizer.

[1] For those unfamiliar, the last parameter of the SpeciesCatalog type list describes how things are stored and looked up in the map. By default, maps store and look up pointers by the value of pointers themselves. Because aux::compareByPtrValue¡mzrSpecies*¿ is passed into the map, the SpeciesCatalog instead looks things up by the dereferenced value of the pointer. That is, the map behaves as though it is a map¡std::string, mzr::mzrSpecies¿, except that you must pass in pointers to the strings instead.

**bool moleculizer::recordSpecies( mzr::mzrSpecies* pSpecies )**

Pass a pointer to a mzrSpecies into this function to attempt to register the species with moleculizer. If the species has already been added, nothing happens and the function returns false. If the species has not been recorded before, its name is generated and it is recorded in the map of names to mzrSpecies* (the same one that can be checked with the moleculizer::getSpeciesCatalog() functions). The species is also added to the delta species list.

This function is probably relatively useless to the user herself, as rule expansion is typically the procedure that generates species, rather than direct user intervention (the rule expanding components use the "recordXXX" functions extensively). But if you have a use for it, knock yourself out!

**void moleculizer::mustRecordSpecies( mzr::mzrSpecies* pSpecies )**

This function does the same thing as moleculizer::recordSpecies, except that it *must succeed. If it does not, this function throws a DuplicatedCatalogEntryXcpt exception.*

**bool moleculizer::recordSpecies( mzr::mzrSpecies* pSpecies, std::string& name )**

*This function simultaniously records a pSpecies, as in the moleculizer::recordSpecies(mzr::mzrSpecies* pSpec) function, and places the generated name for the species into the string reference parameter (the recorded name is placed into the name parameter no matter whether the function returns true or false).*

**bool moleculizer::recordReaction( mzr::mzrReaction* pRxn )**

*This function records pRxn by checking it for consistency (ensuring that it has 0, 1, or 2 substrates – the function throws a utl::FatalXcpt exception if the reaction has 3 or greater substrates), and then records it in theCompleteReactionList (the list of all reactions that can be accessed with the moleculizer::getReactionList() function) as well as theDeltaReactionList (accessed with moleculizer::getDeltaReactionList and cleared with moleculizer::clearDeltaReactionList). If the reaction is recorded, the function returns true, otherwise false.*

*Please note: beacuse of the computational difficulties in comparing reactions, this function ALWAYS succeeds. The reaction that is passed in is always recorded, and the function always returns true. Libmoleculizer is able to make the guarentee that it only creates and attempts to insert reactions once. Consequently, no duplicates will ever be inserted on behalf of libmoleculizer. However, it is entirely possible to manually create new reactions and insert them as new ones. This will cause problems including having an inaccurate number of reactions generated as well as having too many reactions potentially returned for some queries (the right queries will have duplicates).*

**void moleculizer::mustRecordReaction( ReactionTypePtr pRxn )**

*This function does the exact same thing as moleculizer::recordReaction( mzr::mzrReaction* pRxn) does, except that it* must succeed. If it does not, it throws a DuplicatedCatalogEntryXcpt exception. Note that this function, because of compromises made in the name of computational efficiency, will never fail. See moleculizer::recordReaction for more details.

**unsigned int moleculizer::getTotalNumberSpecies()**

This function returns the number of species that have been generated and registered with moleculizer so far - this is the same as moleculizer::getSpeciesCatalog().size().

**unsigned int moleculizer::getTotalNumberReactions() const**

This function returns the number of reactions that have been generated and registered with moleculizer so far - this is the same as moleculizer::getReactionList().size().

**const std::list¡const mzr::mzrSpecies*¿& moleculizer::getDeltaSpeciesList() const**

This function returns the delta species list, which is the list of all species that have been generated since the last time moleculizer::resetCurrentState() was called.

**const std::list¡const mzr::mzrReaction*¿& moleculizer::getDeltaReactionList() const**

This function returns the delta reaction list, which is the list of all reactions that have been generated since the last time moleculizer::resetCurrentState() was called.

**unsigned int moleculizer::getNumberDeltaReactions() const**

This function returns the number of reactions that are in the delta reaction list (this is the same as moleculizer::getDeltaReactionList().size()). Please see chapter **??** for more information on the delta reaction and species lists.

**unsigned int moleculizer::getNumberDeltaSpecies() const**

This function returns the number of species that are in the delta species list (this is the same as moleculizer::getDeltaSpeciesList().size()). Please see chapter **??** for more information on the delta reaction and species lists.

**void moleculizer::resetCurrentState()**

This function clears the delta species and delta reaction lists.

**void moleculizer::incrementNetworkBySpeciesName( const SpeciesID& rName )**

This function looks up the species with the supplied species id and calls its expandReactionNetwork function, presenting that species to the network, and generating any 1) unary reactions with that species as the substrate and all 2) binary reactions involving that species and any other species that have already been expanded.

### 7.1.3 The mzrSpecies class

All species in moleculizer are mzr::mzrSpecies, with the vast majority of those being mzrPlexSpecies. This section discusses the functions in use these classes. [2]

**std::string mzrSpecies::getTag() const**

This function returns a tag to the mzrSpecies, which is a kind of name that can and should only be used within a single instance of moleculizer (it has no persistance – see the section on naming in chapter 2 for more information).

**std::string mzrSpecies::getName() const**

This function returns a unique id to the mzrSpecies, which is a kind of name that uniquely specifies the mzrSpecies in question, even between different instances of the moleculizer object that created it. See the section on naming and unique ids in chapter 2 for more information.

**double mzrSpecies::getWeight() const**

This function returns the weight of the species. This is typically done by adding up each of the weights of the mols that make up the species, along with the masses of any modifications those mols might possess.

**void mzrSpecies::expandReactionNetwork()**

This function expands the reaction network around mzrSpecies. If the species has not been previously examined, each of its features are registered with moleculizer and compared with each feature that has already been registered, according to the rules, with possible generation of species and reactions. If this species has already been expanded, this function does nothing.

The result is that any unary reactions that this species can participate in are created according to the rules, as well as all binary reactions that have this

[2] This list probably isn't an absolutely complete reference, merely a reference to any function the authors think anyone might ever want to use. As always, the source code is the final arbiter for the complete capabilities of this code. Check src/mzr/mzrSpecies.hh and src/plex/mzrPlexSpecies.hh for the basics of this class

species along with another already-expanded species as substrates. All these reactions are registered with moleculizer. Furthermore, any species that occur as products to any of these reactions and which are have not yet been registered are, in an unexpanded state (assuming that the generation depth has its default value of 1). See chapter 2 for more details on expanding species to generate new species and reactions.

**void mzrSpecies::expandReactionNetwork(unsigned int i)**

This function expands the reaction network around mzrSpecies, to a depth of i. This means that the species is expanded, species and reactions are potentially generated, and for any reaction that is generated, each of the product species is called with expandReactionNetwork( i - 1 ). If a species either has already been expanded before, or if i has a value of 0, nothing happens and the function returns.

Typically this function is not used. Instead mzrSpecies::expandReactionNetwork() calls mzrSpecies::expandReactionNetwork( DEFAULT_GENERATION_DEPTH ) with the default generation depth. However, if desired, this function can be called to temporarily override this behavior (if a more shallow or more deep expansion around a particular species is preferred). If the user wishes to override this behavior permanently, do so by calling moleculizer::setGenerateDepth.

### 7.1.4 The mzrReaction class

**mzrReaction::expandReactionNetwork()**

This function calls mzrSpecies::expandReactionNetwork() on each of the product species of the reaction.

**bool mzrReaction::hasReactant(const mzrSpecies* species) const**

This function returns true if the passed in species is one of the substrates of this reaction, false otherwise.

**bool mzrReaction::hasProduct( const speciesType* species ) const**

This function returns true if species is one of the products of this reaction, false otherwise.

**int mzrReaction::getReactantStochiometry( const speciesType* species ) const**

**const std::map¡mzrSpecies*, int¿& mzrReaction::getReactants() const**

This function returns a constant reference to a map from mzrSpecies* to integers, that represents the reactants to the reaction. Each of the keys in the map is a pointer to one of the substrates, with its value being equal to the multiplicity of that substrate in the reaction.

**const std::map¡mzrSpecies*, int¿& mzrReaction::getProducts() const**

This function returns a constant reference to a map from mzrSpecies* to integers, that represents the products to the reaction. Each of the keys in the map is a pointer to one of the products, with its value being equal to the multiplicity of that product in the reaction.

**const std::map¡mzrSpecies*, int¿& mzrReaction::getDeltas() const**

This function returns a constant reference to a map from mzrSpecies* to integers, that represents both the substrates and the products to the reaction. Each of the keys in the map is a pointer to one of the species involved in the reaction. Its value is the overall effect to that species population the reaction has: a positive value means the species population increases by that value and thus that species is a product of the reaction; a negative value means the species population decreases by that value: the species is a substrate of the reaction.

**int mzrReaction::getArity() const**

This returns the number of reactants in the reaction – the sum of all the multiplicities of each of the reactions.

**void mzrReaction::addReactant( mzrSpecies* pSpecies, int multiplicity )**

This function will modify the reaction by adding this species with the given multiplicity to it. If the species is already a reactant or product, this function will deal with it appropriately (if it is already a substrate, this function will add the multiplicity parameter to the reactants multiplicity chart found within the reaction; if it is a product, it will be added as a substrate per usual, but the corresponding entry in the delta chart will have the multiplicity parameter subtracted from it, and so on).

If you have a good reason, and know what you are doing, go ahead and use this function. However, it is hard for us to think of a case in which this would be appropriate (these functions are typically used by the reaction generators, which correspond to rules that look for features and new species to generate new reactions and species) for you to use this. But I guess you know best....

**void mzrReaction::addProduct( speciesType* pSpecies, int multiplicity )**

This function is basically the same as the mzrReaction::addSpecies function, except that it works oppositely, in terms of products and not reactants. Accordingly, the same warnings appear: unless you have a really, really good reason to use this function (reasons that we cannot really imagine as we write this), you really shouldn't.

**double mzrReaction::getRate() const**

This function returns the rate of the reaction as a double. If reaction rate extrapolation has not been enabled, this will have the same numerical value as the appropriate on or off-rate provided in the rule that generated this reaction. If reaction rate extrapolation has been enabled, it will be equal to that value times the reduced mass of the substrates. Please consult chapter 2 for more information.

**void mzrReaction::setRate( double newRate )**

Updates the reaction rate to newRate. You probably shouldn't ever have to use this, except if you were potentially extending libmoleculizer. Pass it a value, and the getRate() function will return that value thereafter.

**std::string mzrReaction::getName() const**

This function generates a string that can be used to output this reaction. If the reaction has substrates A with multiplicity 1 and B with multiplicity 2 and a single product B, this string will look something like "A + 2 B -¿ C".

### 7.1.5 Other global functions for working with mzrSpecies and mzrReactions

The functions here all help with estimating various spatial parameters for the mzrSpecies and mzrReactions. This section expalains each of them.

All these functions are located in "mzr/spatialExtrapolationFunctions.hh".

**double extrapolateIntrinsicReactionRate(const mzr::mzrReaction* pRxn)**

This function calculates the intrinsic reaction rate of the reaction. This is done by solving the equation $\frac{1}{k} = \frac{1}{k_a} + \frac{1}{k_D}$, where $k$ is the overall reaction rate, $k_a$ is the intrinsic reaction rate, and $k_D$ is the diffusion rate, defined as $4 * \pi * (r_1 + r_2) * (D_1 + D_2)$, where $r_i$ is the radius of the $i^{th}$ species, estimated by assuming the species is roughly globular and has density equal to $1.22\ \frac{g}{cm^3}$, and $D_i$ is the diffusion coefficient of the species. See the entry for getDiffusionCoeffFromSpecies to see how this is estimated.

**double extrapolateMolecularRadius( const mzr::mzrSpecies* pSpecies)**

This function returns an estimate of the molecular radius of the species in meters. This is done by calculating volume by assuming average protein density, $1.22\frac{g}{cm^3}$ and then solving for radius by assuming the protein is globular ($V = \frac{4}{3}\pi r^3$).

**double extrapolateMolecularRadius(const double& mass)**

This function returns an estimate for the radius of a protein with the given mass. It works exactly the same as other version of the function: extrapolateMolecularRadius( const mzr::mzrSpecies* pSpecies), except that it takes masses instead of species.

**double getDiffusionCoeffForSpecies( const mzr::mzrSpecies* pSpecies)**

This function checks the value of the cutoff that determines the estimation as to whether a species is a small mol or a protein species. By default, objects which weigh more than 1500 daltons are treated as protein and those which weight less are treated as small mols – this cuttoff value can be inspected and set via the getSmallMolProteinCutoff and setSmallMolProteinCutoff functions. If it is determined to be a small mol, a diffusion coeff of value getSmallMolDiffusionCoeff() is returned; if it is a protein, a diffusion coeff of value getProteinDiffusionCoeff() is returned. See any of this functions along with the setSmallMolDiffusionCoeff and setProteinSmallMolDiffusionCoeff functions for more information.

**double getSmallMolProteinCutoff()**

This function returns the current value of the cuttoff that is used to determine whether a species is a small mol or a protein in the getDiffusionCoeffForSpecies function.

**void setSmallMolProteinCutoff(const double& cut)**

This function sets a new value of the cutoff that is used to determine whether a species is a small mol or a protein in the getDiffusionCoeffForSpecies function.

**double getProteinDiffusionCoeff()**

This function returns the diffusion coeff for proteins. By default it is set to $3.0\frac{\mu m}{cm^2}$, but can be set by using the setProteinDiffusionCoeff function.

**double getSmallMolDiffusionCoeff()**

This function returns the diffusion coeff for small molecules. By default it is set to $100.0\frac{\mu m}{cm^2}$, but can be set by using the setSmallMolDiffusionCoeff function.

**void setProteinDiffusionCoeff(double rate)**

This function sets the value - by default set to $3.0\frac{\mu m}{cm^2}$ that is returned by the getProteinDiffusionCoeff function.

**void setSmallMolDiffusionCoeff(double rate)**

This function sets the value - by default set to $100.0 \frac{\mu m}{cm^2}$ - that is returned by the getSmallMolDiffusionCoeff function.

## 7.2 The C-Interface API

This portion of the chapter is devoted to discussing all the functions that exist in the C-Interface API. For an introduction to how to use this, please consult the relevant section in chapter **??**. As always, the best possible "reference" to the code is the code itself. This can be found in "mzr/libmzr_c_interface.h" and "mzr/libmzr_c_interface.cc".

**moleculizer* createNewMoleculizerObject()**

This function creates a new moleculizer object and returns a pointer to it. This pointer is owned by the caller and must be freed using the freeMoleculizerObject function, or else the memory is lost and leaked. This pointer should be included as the paramter to all the other functions in this interface.

**void freeMoleculizerObject( moleculizer* handle)**

Call this function with a moleculizer* that has been created by the createNewMoleculizerObject to free it. This is the only way to properly release the memory held by the pointer.

**int setRateExtrapolation( moleculizer* handle, int extrapolation)**

Call this function prior to calling one of the functions for loading a model to set reaction rate extrapolation on the moleculizer object (if extrapolation is != 0) or to turn it off (if extrapolation == 0) – extrapolation is disabled by default.

If reaction rate extrapolation is enabled, when a binary reaction is created, its rate is set as the rate in the rule which created it multiplied by the reduced mass of the products. If reaction rate extrapolation is disabled, the rate provided in the rule is simply passed along as the rate of the reaction as is. Please consult chapter 2 for more details as to what this is and why it might be used.

This function returns error codes: 0 for success, 1 to indicate an unknown error, and 2 to indicate that this function was incorrectly called on a moleculizer object that already has had rules loaded into it.

**int loadXMLRulesFile(moleculizer* handle, char* fileName)**

**int loadXMLRulesString(moleculizer* handle, char* fileName)**

**int loadCommonRulesFile(moleculizer* handle, char* fileName)**

This function takes a moleculizer object as well as a null-terminated string that contains a file name (which in unix is the entire relative path – values

like “models/simple/simple-rules.mzr” or “ /models/my-model.mzr” or “the-model.mzr” are all perfectly valid).

This function returns the following error codes: 0 for success, 1 to indicate an unknown error, 2 to indicate the file was unparsable, 3 to indicate that rules have already been loaded here, 4 to indicate that the file was not found.

**int loadCommonRulesString( moleculizer* handle, char* file)**

This function takes a moleculizer object as well as a null-terminated string that contains the contents of a rules file. The result of the file is to load the information into file into the moleculizer object.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error, 2 to indicate the document was unparsable, 3 to indicate that rules have already been loaded.

**int writeDotFile(moleculizer* handle, char* fileName)**

**int expandNetwork( moleculizer* handle)**

Call this function on a moleculizer object that has had some rules loaded into it to have that network completely expanded out. **Beware!!!** This function may take a very long time to run, if it returns at all.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error, and 2 to indicate that the moleculizer object has not had rules loaded into it.

**int getBoundedNetwork( moleculizer* handle, long maxNumSpecies, long maxNumReactions, species*** pSpeciesArray, int* pNumSpec, reaction*** pReactionArrry, int* pNumRxns)**

**int incrementSpecies( moleculizer* handle, char* speciesTag)**

This function takes a moleculizer object, and a null-terminated char* containing the name of a species. It finds the corresponding species and expands it - any unary reactions that species can participate in will be created as well as any binary reactions that involve the parameter species along with as any other already expanded species, along with all product species of those reactions that have not yet been registered.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error, 2 indicates the species name passed in is not listed in the species within the reaction network.

**int expandSpeciesByTag( moleculizer* handle, char* theTag)**

This function will expand the reaction network around a particular species. That is, calling this function with a species pointer will cause several things to occur. 1) if this species has been expanded before, either using this function directly or by having had a reaction that includes this species as a product previously

expanded. 2) if this species has not been expanded, all unary reactions involving this species as a substrate will be created and all binary reactions that can occur in the rules between this species and all other already registered and already expanded species will be created; furthermore, any unknown products of any of those reactions will be registered as new, unexpanded species with moleculizer.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error.

**int expandSpeciesByID( moleculizer* handle, char* theID)**

This function will expand the reaction network around a particular species. That is, calling this function with a species pointer will cause several things to occur. 1) if this species has been expanded before, either using this function directly or by having had a reaction that includes this species as a product previously expanded. 2) if this species has not been expanded, all unary reactions involving this species as a substrate will be created and all binary reactions that can occur in the rules between this species and all other already registered and already expanded species will be created; furthermore, any unknown products of any of those reactions will be registered as new, unexpanded species with moleculizer.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error.

**int getNumberOfSpecies(moleculizer* handle)**

This function returns the number of species that the moleculizer handle that has been passed in has recorded thus far in the simulation.

**int getNumberOfReactions(moleculizer* handle)**

This function returns the number of reactions that the moleculizer handle that has been passed in has recorded so far in network expansion.

**int getDeltaSpecies( moleculizer* handle, species*** pSpeciesArray, int* pNum)**

This function returns a list of the delta species – the list of all species that have been created and registered with libmoleculizer since the last time the clearDeltaState function was called.

Pass it a moleculizer variable and the addresses of a species** and integer variables. This function will setup *pSpeciesArray such that it contains an array of *pNum species pointers; this array is the list of species that have been produced since the last time clearDeltaState was called.

As always, this array of species pointers is owned by the caller, and should be freed using the freeSpeciesArray function lest the memory is lost.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error.

**int getDeltaReactions( moleculizer* handle, reaction*** pReactionArray, int* pNum)**

This function returns a list of the delta reaction – the list of all reactions that have been created and registered with libmoleculizer since the last time the clearDeltaState function was called.

Pass it a moleculizer variable and the addresses of reaction** and integer variables. This function will setup *pReactionArray such that it contains an array of *pNum reaction pointers; this array is the list of reactions that have been produced since the last time clearDeltaState was called.

As always, this array of reaction pointers is owned by the caller, and should be freed using the freeReactionArray function lest the memory is lost.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error.

**int clearDeltaState( moleculizer* handle)**

This function clears the delta state of the moleculizer object.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error.

**int getAllSpecies(moleculizer* handle, species*** pSpeciesArray, int* numberSpecies)**

This function returns an array containing species*'s to each of the species that have been recorded thus far during the course of moleuclizer's network expansion.

It works the same way as most other functions in this interface: define a species** object as well as an int, and pass in their addresses into the function. This function will make the species** point to a array containing freshly allocated pointers to each species in libmoleculizer, and will put the size into the integer parameter. This array is owned by the user, and should be freed using the freeSpeciesArrays functions.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error.

**int getAllReactions(moleculizer* handle, reaction*** pReactionArray, int* numberReactions)**

This function returns an array of pointers to all the reactions that have been generated thus far in moleculizer's network expansion.

**int getNumModificationDefs( moleculizer* handle)**

**int getNumMolDefs( moleculizer* handle)**

**int getNumReactionRules( moleculizer* handle)**

**int getNumDimerDecompReactionRules( moleculizer* handle)**

**int getNumOmniGenReactionRules( moleculizer* handle)**

**int getNumUniMolGenReactionRules( moleculizer* handle)**

**int getNumSpeciesStreams( moleculizer* handle)**

**int getReactionsBetween(moleculizer* handle, char* speciesName1, char* speciesName2, reaction*** ptrReactionPtrArray, int* numReactions)**

This function finds all binary reactions between two speciesand returns them to the user.

To use this function, create a reaction pointer array and an integer, and pass their addresses into the function, along with a null-terminated string containing the name of the species in interest.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error, 2 to indicate the species name that was passed in could not be found.

**int getUnaryReactions(moleculizer* handle, char* speciesName, reaction*** ptrReactionPtrArray, int* numReactions)**

This function finds all unary reactions involving a particular species and returns them to the user.

To use this function, create a reaction pointer array and an integer, and pass their addresses into the function, along with a null-terminated string containing the name of the species in interest. The result is that moleculizer will place into *ptrReactionPtrArray an array of reaction*s that represent each of these reactions, and into *pNumReactions the number of elements in the array.

The reaction array that is placed into the reaction pointer array is owned by the caller of the function and must be freed using the freeReactionArray function in this interface.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error, 2 to indicate the species name that was passed in could not be found.

**int getAllExteriorSpecies(moleculizer* handle, species*** pSpeciesArray, int* numberSpecies)**

This function returns an array of all species that have not yet been expanded. Create a species** (array of species pointers) and an int that will hold the number of elements in that array. Pass in a moleculizer object, the addresses of

the species pointer array and the integer value. Moleculizer will insert into the array each of the species that is have been recorded, but not yet expanded, and place into the integer the number of elements that have been inserted.

Each of these species pointed to by elements of the array are owned by the caller and should be freed using the freeSpeciesArray function.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error.

**int getMolCountInTaggedSpecies(moleculizer* handle, const char* molName, const char* speciesTag)**

**int getAllStreamSpecies(moleculizer* handle, char* streamName, species** pSpeciesArray, int* numberSpecies)**

**void getAllSpeciesStreams( moleculizer* handle, char*** speciesStreamArray, int* numberSpeciesStreams)**

**int checkSpeciesTagIsInSpeciesStream( moleculizer* handle, const char* speciesID, char* speciesStream)**

**int checkSpeciesIDIsInSpeciesStream( moleculizer* handle, const char* speciesID, char* speciesStream)**

**int convertTaggedNameToUniqueID( moleculizer* handle, char* speciesTag, char* speciesID, int idSize)**

The purpose of this function is to take a tagged name of a complex species and convert it to a unique id.

This function takes as its first two parameters a moleculizer pointer as well as a null-terminated string that contains a tagged name of a species in the moleculizer object; the result of this function is to find the corresponding unique id of the species possessing that tagged name, and insert it into a charecter buffer: passed in as the third parameter, with the 4th being its size.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error, 2 to indicate that the tag was not found in moleculizer, 3 to indicate that the supplied buffer does not have enough memory to contain the resulting name.

**int convertUniqueIDToTaggedName( moleculizer* handle, char* speciesID, char* speciesTag, int tagSize)**

This function converts a unique id to a tagged name. To do this, pass in a moleculizer object, the unique id in question, along with a charector buffer and a parameter describing the size of that buffer. The interface will perform the conversion and insert the result into the speciesTag charector buffer.

**Note:** Because of the nature of tagged names, virtually all systems will need no more than 9 characters (length of 8 plus a null terminator) and no system should need more than 17 (16 plus a terminator) to hold the tag. Furthermore, for any system, all tagged names will have constant size on that system.

This function returns the following error codes: 0 for success, 1 to indicate an unknown error, 2 to indicate the unique id was not found in moleculizer, and 4 to indicate (if you get this you're probably doing it wrong) that the provided buffer is not large enough to hold the tagged name.

**int convertUserNameToTaggedName(moleculizer* handle, char* theUserName, char* correspondingTag, unsigned int bufferSize)**

**int convertUserNameToUniqueID(moleculizer* handle, char* theUserName, char* correspondingSpeciesID, unsigned int bufferSize)**

**int convertSomeNameToTaggedName( moleculizer* handle, char* theName, char* taggedNameBuffer, unsigned int bufferSize)**

**int getExplicitSpeciesList(moleculizer* handle, char*** theExplicitSpeciesNames, unsigned int* numSpecies)**

**void freeReactionArray( reaction** pRxnArray, unsigned int numArrayElements)**

Use this function to free an array of reactions, such as the arrays returned via the getAllReactions, the getReactionsBetween, and the getReactionsInvolving functions. Pass in the array as well as the number of elements in the array and they will all be freed.

**void freeSpeciesArray( species** pSpeciesArray, unsigned int numArrayElements)**

Use this function to free an array of species, such as the array returned by the getAllSpecies or getAllStreamSpecies functions. Pass in the array of species pointers as well as the number of elements in the array and they will be freed.

**void freeCharPtrArray( char** pCharArray, unsigned int numCharPtrElements)**

**void freeReaction( reaction* pRxn )**

This function will free a pointer to a reaction. Simply pass the pointer in to this function and it will be freed.

**void freeSpecies( species* pSpecies )**

This function will free a pointer to a species. Simply pass the pointer into this function and it will be freed.

# Chapter 8

# MZR Format Reference

This chapter serves as a complete reference to the MZR file format specification.

To understand it, at least a basic knowledge of the Relax NG format for specifying xml schemas will be assumed. (Never fear! Even if the format is unknown to you, dear reader, it is fairly straightforward and can be largely understood without difficulties!)

## 8.1 /moleculizer-input

This mandatory root element of a moleculizer model contains a single model subelement.

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar
    xmlns="http://relaxng.org/ns/structure/1.0"
    datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <element name="moleculizer-input">
      <element name="model">
        <ref name="model-content" />
      </element>
    </element>
  </start>
</grammar>
```

## 8.2 /moleculizer/model

The model element contains all the biology in the model. It consists of

```
<element name="model">
  <element name="modifications">
```

```
    <ref name="modifications-content" />
  </element>
  <element name="mols">
    <ref name="mols-content" />
  </element>
  <element name="allosteric-plexes">
    <ref name="allosteric-plexes-content" />
  </element>
  <element name="">
    <ref name="" />
  </element>
  <element name="">
    <ref name="" />
  </element>
  <element name="">
    <ref name="" />
  </element>
  <element name="">
    <ref name="" />
  </element>
</element>
```

## 8.3 A Complete Specification of MZR Format in RelaxNG Format