Libmoleculizer 1.1.2 User's Manual

Nathan Addy

The Molecular Sciences Institute

July 9, 2009

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 **??** and **??**. The second objective is to describe the MZR file format in which reaction rules are specified; this is given an overview in chapter **??**, and more explicitly in chapter **??**. 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 **??**.

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 **??**.

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 1.1.2of 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 inter-

action 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;
```