Reaction Description Language, Version 0.2.0 (alpha)

Rex Kerr

16th July 2004

1 Introduction

There is no introduction for now.

2 Conceptual Basics

2.1 Reactants

The elementary particles that take part in a reaction are called *molecules*. Each molecule is a member of a *species*, which determines various physical properties of that molecule such as diffusion constant, mass, charge, and so on. Each molecule has zero or more *binding sites* for other molecules. The number of sites is a property of the species, but what is bound to each site can vary from molecule to molecule.

Elementary particles can bind each other (via their binding sites) to form a *complex* consisting of two or more molecules. Complexes do not have a species, but rather inherit their properties from the molecules that make them up. This inheritance has a default behavior, but unique rules can be specified if this is desired.

2.2 Reactions

A reaction consists of three parts: a *trigger*, a *condition*, and an *outcome*. Reactions can also be labeled with a name for re-use. (This is particularly helpful for specifying *subreactions*.)

2.2.1 Triggers

A trigger is a list of one or two molecules that must be present for any reaction to occur. If these molecules are surface-associated, the trigger can also define *orientation equivalence classes* that defines the relative orientation of the trigger molecules, and optionally, the identity of the surface. If the reaction trigger is activated (by a collision between two molecules or a timestep, for unimolecular reactions), the reaction condition is tested.

2.2.2 Conditions

Conditions are arbitrary functions of the triggering molecules (and any complex they are a part of) and, if desired, global or local variables. The most common condition is a reaction rate, which determines the probability that the reaction outcome will occur. Other conditions can include the occupancy of various binding sites, the identity of molecules in a binding site, and so on. In all cases, the condition is interpreted as a probability (from 0 to 1, typically) that the outcome will happen. If some condition is not met (e.g. a binding site that needs to be empty is actually full), the probability will be zero. It is up to the user to return sensible probabilities for custom conditions.

If there are multiple nonzero conditions for a single trigger, then the probability that nothing will occur is equal to the product of the probabilities that nothing will occur in each case (i.e., the standard method of determining probabilities). If something does happen, which outcome happens is determined at random, with a probability weighted according to the probability returned by the condition.

2.2.3 Outcomes

Outcomes can destroy the triggering molecules (and arbitrary subsets of what is connected to them), can change their species, location, release molecules that are bound to them, create new molecules at arbitrary locations (bound or unbound), and test *subreactions*, reactions that have only single-molecule triggers.

2.2.4 Subreactions

Subreactions, like reactions, are composed of a trigger, a condition, and an outcome. Subreaction triggers can only have a single triggering molecule; the trigger is activated if that molecule was in the reacting molecule or complex (and has not been destroyed) or if it is created as part of the outcome of the main reaction. Conditions and outcomes of subreactions are unrestricted. Subreactions can be nested.

2.3 Something probably needs to go here.

And here it will be.

3 Defining Molecules

3.1 Naming Molecules

Each species of molecule needs to be specified in advance. This is done with the `DEFINE_MOLECULE` keyword, followed by the name of the molecule:

```
DEFINE_MOLECULE foo { ... }
```

This defines a molecule named `foo` and assigns to it the properties listed in braces. If one wishes to define multiple molecules at once, one can list them in braces after the `DEFINE_MOLECULES` keyword:

```
DEFINE_MOLECULES {
  foo { ... }
  bar { ... }
  IP3 { ... }
  . . .
}
```

3.2 Specifying Binding Sites

Binding sites are listed in parentheses after the name of a molecule. Note: many chemical reactions can be simulated without any binding sites at all! Defining binding sites, however, enables one to define reactions where combinatorial explosion would otherwise make even listing all the different molecular species impossible.

Each binding site has a name and a maximum occupancy that specifies how many molecules can bind to that binding site. If multiple molecules bind to the same binding site, they have no relative order to each other. In order to distinguish between binding at different sites, one must declare and name each distinct site. For example,

```
DEFINE_MOLECULES {
  foo ( b , i[4] ) { ... }
  bar ( f , iA , iB ) { ... }
  IP3 ( s )    { ... }
}
```

specifies two binding sites on `foo`: a `b` site that can bind one molecule, and an `i` site which can bind four; three binding sites on `bar`, `f`, `iA`, and `iB`, each of which can bind one molecule; and one single-occupancy binding site on `IP3` called `s`. If one wishes to refer to these sites, one uses the name of the molecule followed by a dot followed by the site name. Thus,

```
bar.iA
```

refers to the `iA` binding site of a `bar` molecule. If one wishes to refer to a specific slot in a site with multiple occupancy, one can specify the number after a second dot (starting with the number 1). Thus

```
foo.i.2
```

is the second slot in the `i` binding site of a `foo` molecule.

It is important to note that molecules bind each other. Therefore, if molecules of species `A` bind molecules of species `B`, each must have at least one binding site in order to bind the other. Attempting to bind a molecule with no binding sites is a syntax error.

3.3 Specifying Molecular Properties

3.3.1 Basic Properties

There are a number of pre-defined molecular properties that can be set when defining a molecule. These are

```
D
D_2d
R
... (list incomplete)
```

These determine the diffusion in 3D and 2D and the molecular radius (and other stuff I haven't listed). To set these properties, simply set the property name equal to its value:

```
DEFINE_MOLECULES {
  foo ( b , i 4 ) {
    D = 2.5e-6
    D_2d = 0
    R = 3.9
  }
  . . .
}
```

(Talk about units. Talk about how the RHS can be a variable name or function.)

As with binding sites, properties can be referred to using dot notation. For instance, `foo.D` is the diffusion constant of a molecule of species `foo`.

3.3.2 Inherited properties

If one molecule is very similar to another, you can have the second molecule *inherit* the properties of the first. This is done using the same syntax as for class inheritance in C++:

```
DEFINE_MOLECULES {
  foo ( b , i 4 ) {
    D = 2.5e-6
    D_2d = 0
    R = 3.9
  }
  bar ( f , iA, iB) : foo {
    D_2d = 1.5e-9
  }
  . . .
}
```

In this example, `bar` inherits the values `D = 2.5e-6`, `D_2d = 0`, and `R = 3.9` from `foo`. The value of `D_2d` is then overridden to be `1.5e-9` for `bar`.

3.3.3 Custom properties

In some cases, it is useful to have user-defined properties. Property names must first be declared using the keyword `DECLARE_PROPERTY` followed by the name of the property. Multiple names can be declared using `DECLARE_PROPERTIES { ... }`. These properties can then be assigned (or not assigned) in the species declaration just like the predefined properties. User-written reaction condition / reaction outcome code can then read these properties. For example, if we wanted to define a property called `CHARGE`, we would write:

```
DECLARE_PROPERTY CHARGE
DEFINE_MOLECULES {
  foo ( b , i 4 ) {
    D = 2.5e-6
    D_2d = 0
    R = 3.9
    CHARGE = -1
  }
  . . .
}
```

If a custom property is declared but not defined for a species, its value is set to zero. All custom properties are 64-bit floating-point numbers (doubles).

3.3.4 Custom variables

Some molecules may need additional user-defined variables in order to keep track of their state. These variables can be defined on a species-by-species basis using the VARIABLE keyword followed by the name of the variable. Variables are 64-bit floating-point numbers (doubles). Variables are initially set to zero unless you specify otherwise (by assigning a value). Be judicious in the use of variables; they can add significantly to the memory usage of molecules. Here is an example of adding a variable called `PHOS_STATE` to `foo` and setting it to `1` initially:

```
DEFINE_MOLECULES {
  foo ( b , i 4 ) {
    D = 2.5e-6
    . . .
    VARIABLE PHOS_STATE = 1
  }
  . . .
}
```

Custom variables can be referred to using dot notation, e.g. `foo.PHOS_STATE`.

3.4 Derived Molecules

So far, all the molecules we have been working with have been *primative molecules*, that is, molecules that cannot be decomposed into parts. If one doesn't need the full power of molecular complexes, one can create *derived molecules*, which are molecules composed of other primative or derived molecules. The components of a derived molecule cannot interact with other molecules; the derived molecule acts as a single entity, except for remembering the identity of its components.

3.4.1 Defining derived molecules

To define a derived molecule, one lists the component primative or derived molecules after a semi-colon in the list of binding sites. For example:

```
DEFINE_MOLECULES
{
  foo ( i[4] ) { . . . }
  bar ( iA , iB ) { . . . }
  foobar ( i[4] , iA , iB ; foo , bar ) { . . . }
}
```

In this case, it is not necessary for foo and bar to have binding sites for each other; that is implicit in the definition of the defined molecule.

3.4.2 Assigning binding sites

Since the derived molecule can have all new binding sites, it is necessary to specify which binding sites of the components map onto which binding sites in the derived molecule. This is done by assigning the binding sites of the derived molecule to the binding sites of the components:

```
DEFINE_MOLECULE foobar ( i[4] , iA , iB ; foo , bar )
{
  i = foo.i
  iA = bar.iA
  iB = bar.iB
  D = ...
}
```

All sites in both the derived molecule and the component molecules must be assigned. If a binding site is set to 0, anything bound there will be destroyed when the binding site disappears, and it will be empty when the binding site appears. Thus, if the contents of that site should be released, one must include the product when specifying the reaction that creates or destroys the derived molecule.

If there are multiple components with the same name, use .1, .2, etc. to refer to them:

```
DEFINE_MOLECULE bar_dimer ( iA1 , iA2 , iC ; bar , bar )
{
  iA1 = bar.1.iA
  iA2 = bar.2.iA
  iC = 0
  bar.1.iB = 0
  bar.2.iB = 0
}
```

3.5 Molecular Classes

Sometimes it is useful to specify reactions that occur identically for an entire class of molecules. To define the class, use the keyword DEFINE_CLASS with the following syntax:

```
DEFINE_MOLECULES
{
  foo  ( b , i[4] ) {...}
  phoo ( b , j ) {...}
  fue  ( w , q ) {...}
}
DEFINE_CLASS barbinder (b,i : foo,phoo,fue)
{
  b = { foo.b , phoo.b , fue.w }
  i = { foo.i[0] , phoo.j , fue.q }
}
```

to define which molecules fall into that class and to specify the mapping between different binding sites. The binding sites of a class must be present in every single member of the class. Some members may have more binding sites, but those are inaccessible to any reactions dealing with that class of molecule.

Derived molecules can be derived from classes, but anything in a binding site not explicitly mentioned in the class definition will be destroyed upon creation of the derived molecule.

4 Working with Complexes

4.1 Naming complexes

A complex consists of two or more molecules bound together via their binding sites. Although a complex may not have any true hierarchy, when one is naming it, one considers one molecule to be the root of the hierarchy, and its name is listed first. Any molecules bound to its binding sites are the listed in parentheses using the syntax `site_name : molecule_bound.site_bound`. If a molecule which is bound has further things bound to it, they may be listed in parentheses following the name of the molecule. If a site has multiple slots, the things bound in each slot can be listed in brackets. If a site must be empty, it should be listed as containing `0` (zero). If the occupancy of a site does not matter, do not list it. For example:

```
foo( b : bar.f(iA:IP3.s,iB:0) , i : [IP3.s , IP3.s , 0 , 0] )
```

This example takes a little work to unpack. We have a `foo` molecule. The `foo.b` binding site has a `bar` in it (specifically, the `foo.b` site and `bar.f` sites have bound each other). The `bar` has an `IP3` in its `iA` site and nothing in its `iB` site. The `foo.i` binding site has two `IP3`s and two empty slots in it. In each case, `IP3` is binding to `foo` or `bar` with `IP3`'s only binding site. There are other ways to name this exact same complex. For instance:

```
bar( f : foo.b(i:[IP3.s,IP3.s,0,0]) , iA:IP3.s , iB:0 )
IP3( s : foo.i ( b : bar.f(iA:IP3.s,iB:0) , i : [IP3.s , 0 , 0] ) )
IP3( s : bar.iA ( f : foo.b(i:[IP3.s,IP3.s,0,0]) , iB:0 )
```

Note that in the second example, we have listed only three entries for `foo.i` since one of them is already specified on the outside. (Need to mention the difference between `foo.i.1` and `foo.i.2` and `foo.i` in this context.)

4.2 Building complexes and defining aliases

A specific, commonly-used complex may be constructed using the `DEFINE_COMPLEX` command. This lets you name a complex and then build it out of molecules by joining bonds together. One first lists all of the molecules present in the complex in parentheses after the name of the complex and then assigns bonds using `=`. If there are multiple molecules of the same type in the `COMPONENTS`, one can refer to them by `name.number`, as shown here:

```
DEFINE_COMPLEX foobar ( foo , bar, ip3[3] )
{
  foo.b = bar.f
  bar.iA = ip3.1.site
  foo.i.1 = ip3.2.site
  foo.i.2 = ip3.3.site
}
```

This builds the complex that we named in the previous section; unassigned bonds are assumed to be empty. One can then use `foobar` as an alias for the lengthy names given above. It is a syntax error to have free components at the end of the construction. It is legal to construct cyclic complexes, but keep in mind that most reactions involving cyclic complexes require custom user-written reaction outcome code. Cyclic complexes can only be referred to using aliases; the naming scheme in the previous section is unable to adequately describe them.

If one wishes to create an alias for a fragment of a complex that uses wildcards, one can assign bonds to the wildcard character `*` to indicate that it doesn't matter what is at that site. For instance, if we only care that `foo` and `bar` are bound to each other and don't care about how many `IP3`s they have bound, we would write

```
DEFINE_COMPLEX foobar2 ( foo , bar )
{
  foo.b = bar.f
  foo.i = *
```

```
    bar.iA = *
    bar.iB = *
  }
```

For a binding site with multiple occupancy it is possible to assign some sites values and some sites wildcards. One can use array notation if one wishes (e.g. `foo.i = [*,*,*,0]` refers to something with no more than three `foo.i` slots filled).

4.3 Assigning Properties to Complexes

4.3.1 Direct Assignment

Properties can be assigned directly to a complex when the complex itself is defined, using the same syntax as when defining properties for molecules. For instance:

```
DEFINE_COMPLEX foobar2
{
  COMPONENTS { foo,bar }
  foo.b = bar.f
  foo.i = *
  bar.iA = *
  bar.iB = *
  D = 1.3e-9
  . . .
}
```

Alternatively, properties for un-aliased complexes can be specifed using the `COMPLEX_PROPERTIES` keyword, followed by the name of the complex, and then the assignment of properties in braces:

```
COMPLEX_PROPERTIES foo(b:bar.f)
{
  D = 1.3e-9
  . . .
}
```

4.3.2 Default inheritance

If properties are not defined explicitly for a complex, they are inherited from the component molecules. In most cases, the complex is searched for a *primary component* whose properties become the properties of the complex. Primacy is determined by the following factors, in order: highest value of the `PRIMACY` property, highest radius, lowest alphabetical order. If `PRIMACY` is not set, it defaults to zero. For example, if we have:

```
DEFINE_MOLECULES {
  foo(b,i 4) { D=2.9e-6 R=7.1 }
  bar(f,iA,iB) { D=2.4e-6 R=3.9 }
  IP3(site) { D=8.4e-6 R=0.92 PRIMACY=1 }
}
DEFINE_COMPLEX foobar3
{
  COMPONENTS { foo,bar }
```

```
    foo.b = bar.f
    foo.i = *
    bar.iA = *
    bar.iB = *
  }
```

and we have two complexes:

```
  foo(b:bar.f(0,0),0)
  foo(b:bar.f(IP3.site,0),0)
```

then the first will have properties `D=2.9e-6 R=7.1` since the radius of `foo` is larger than the radius of `bar`. (If the radius were not set, it would have `bar`'s properties, as `bar` comes before `foo` in dictionary order.) The second will have the properties `D=8.4e-6 R=0.92 PRIMACY=1`, since it has `IP3` in it, and `IP3` has a higher primacy than foo and bar.

Two directives alter this default inheritance behavior. The first is `COMPUTE_DIFFUSION` and the second is `COMPUTE_RADIUS`. These are instructions to assume (for the entire model) that by default, the diffusion constant is computed according to the formula $D_{\mathrm{A,B}} = \left(D_{\mathrm{A}}^{-3} + D_{\mathrm{B}}^{-3}\right)^{-1/3}$ and the radius is computed according to $R_{\mathrm{A,B}} = \left(R_{\mathrm{A}}^{3} + R_{\mathrm{B}}^{3}\right)^{1/3}$. These formulae are derived for diffusing spheres with conservation of mass, but may not be accurate for any realistic situation. Simply placing `COMPUTE_DIFFUSION` or `COMPUTE_RADIUS` anywhere in the file will turn on this behavior.

4.3.3 Algorithmic assignment

Any direct assignment may be made via an array indexed by free parameters or by evaluation of a function of the complex. Later sections contain more information on writing arrays and functions.

4.4 Internal Representation of Complexes

Internally, a complex consists of a data structure containing property information for the complex (or a pointer to such information, for complexes with many instances or simple property inheritance), a pointer to the primary molecule in a complex, and an array of all free binding sites for all molecules in the complex (and pointers to the molecules that own them). The primary molecule is the one with the highest primacy (including radius and low-alphabetic-order); if there is a tie, the primary molecule is the one with highest summed primacy of things bound to it. In the case of symmetric completely symmetric molecules, the first highest-primacy molecule encountered becomes the primary molecule for that complex.

(Need to talk about implementation of primacy some more–how to compute it.)

5 Specifying Reactions

5.1 Parts of a Reaction

To specify a reaction, one uses the `DEFINE_REACTION` keyword (or `DEFINE_REACTIONS` for multiple reactions). This is followed by the name of the reaction (which is optional), the triggering molecules in parentheses, and then the `CONDITION` and `OUTCOME` blocks. An example is below, and each part will be discussed in turn.

```
  DEFINE_REACTION foobindsbar ( foo , bar )
```

```
{
  CONDITION
  {
    foo.b = 0
    bar.f = 0
    RATE = 1.7e-9
  }
  OUTCOME
  {
    foo.b = bar.f
  }
}
```

5.2 Triggers

5.2.1 Specifying molecular identity

Triggers can specify one or two species names and zero or one surface name. Having zero, or three or more species names is a syntax error, as is having more than one surface name.

If there is only one species name, the trigger is activated at every time step for each molecule with that species name, whether or not it is in a complex. If there is also a surface name, the trigger is only activated if that molecule is in a surface of that type, or is in a complex that is in a surface of that type.

If there are two species names, the trigger is activated whenever one molecule collides with the other during diffusion (either alone or as part of colliding complexes). If there is also a surface name, at least one of the molecules has to be in a surface of that type.

5.2.2 Specifying molecular orientation

Molecules which are in a surface also have an orientation, which one can reference with the `.ORIENT` field. Orientations are either `+1` (or `HEAD`) or `-1` (or `TAIL`), measured relative to the surface normal of the surface they are embedded in. A molecule that is not in a surface has an orientation of zero. In a trigger, orientations are grouped into equivalence classes; if two molecules care about each other's relative orientation, they are in the same equivalence class. If not, they are in different equivalence classes. By default, molecules in the trigger are all in the same equivalence class (the `:1` class, see below), but any surface listed is in a different equivalence class. Thus, molecules care about each other's orientation, but not about the orientation of the surface.

To alter this default behavior, one specifies the equivalence class after the name of the molecule or surface by using an integer. Positive and negative integers of the same absolute value are opposite directions in the same equivalence class. Different integers refer to different orientation equivalence classes. For example:

```
DEFINE_REACTION x ( foo:1 , bar:-1 )
```

assigns `foo` and `bar` to the same equivalence class, but says that `foo` and `bar` have to be in opposite orientations. This would be met if `foo.ORIENT=1` and `bar.ORIENT=-1`, or if `foo.ORIENT=-1` and `bar.ORIENT=1`. Because of this, the trigger

```
DEFINE_REACTION x ( foo:-1 , bar:1 )
```

means exactly the same thing. In contrast,

```
DEFINE_REACTION x ( foo:1 , bar:0 , outer_mito_memb:1 )
```

says that `foo` has to be oriented in the positive direction and in the `outer_mito_memb` surface, but `bar`'s orientation doesn't matter. (`bar:2` also would say that `bar`'s orientation doesn't matter, as nothing else would be in that equivalence class.)

Formally, the orientation condition is met when the following conditions apply. Let A and B be molecules or surfaces with orientations $p_A = A.\texttt{ORIENT}$ and $p_B = B.\texttt{ORIENT}$ and reaction equivalence classes of n_A and n_B. Then for every pair listed in the trigger condition, we must either have $|n_A| \neq |n_B|$or $p_A n_A = p_B n_B$.

If one of the molecules is not in the surface, its orientation is `+1` or `HEAD` if it is on the positive side of the surface and `-1` or `TAIL` if it's on the other side.

5.3 Conditions

The condition block is a list of statements each of which must be true, followed by a single statement that gives a probability for being true; if any statement in the list fails, a probability of 0 is assumed. There are a variety of default statements that one can use. The names of molecules used in the conditions block is assumed to refer to the molecules listed in the trigger block. If there are two molecules of the same type, one can use `.1` and `.2` to distinguish them.

5.3.1 Bond conditions

A bond condition is a statement that a given bond is equal to 0 (is empty) or contains a specific other molecule, or is in a specified set of molecules. Examples of each are given below.

```
foo.b == 0
foo.b == bar
foo.b IN [ bar , IP3 ]
```

Additionally, if one compares a bond with a number, that is interpreted as a condition for how many binding sites must be filled or empty:

```
foo.i < 3
```

One can use dot notation to refer to different molecules of the same type. For example, in a reaction with trigger molecules `foo` and `IP3`, we might say

```
foo.i[1] == IP3.2
IP3.1.site == 0
```

to require that the first site in the `i` binding site of `foo` has an `IP3` in it, but to explicitly say that it is different than the `IP3` that has just collided with the complex containing `foo`. (This would always be true, but in more complex conditions with multiple binding sites, one could need to refer to many molecules with the same name but different identities.)

In reality, conditions are evaluated on single molecules, and the names we have defined are for the species of the molecule: `foo` refers to the species foo, not to any particular molecule. When entering a reaction block, however, the triggering molecule is labeled `foo.1`, and when one tries to use a species where a molecule is expected, it converts to the first instance of a molecule of that species, i.e. `foo` is converted to `foo.1`. So everything just works. Hopefully.

If one wishes to verify something complex–for example, we can only bind to the iB site of bar if bar has foo bound to it, and foo has at least two IP3s in its i binding site–one can proceed a piece at a time:

```
TRIGGER { bar IP3 }
CONDITION
{
  bar.f == foo
  foo.i >= 2
  bar.iA == IP3.2
  bar.iB == 0
  IP3.site == 0
}
```

Two useful shorthands are defined. First, if the right-hand side is a binding site rather than a molecule, it means that the two molecules are bound to each other using those binding sites. For example,

```
bar.1.f == foo
foo.b == bar.1
```

can be written using this shorthand as

```
bar.f == foo.b
```

Second, one can use the IN keyword to state that a molecule is part of a given complex. For example,

```
bar.1 IN foo( b:bar.1.f(iA:IP3.site,0) , i:[IP3.site,0,0,0])
```

(using dot notation to avoid ambiguity) is equivalent to the bond conditions

```
bar.f == foo.b
foo.i[1] == IP3.site
foo.i == 1
bar.iA == IP3.site
bar.iB == 0
```

5.3.2 Variable-dependent conditions

Equalities and comparisons can also be made with global variables, species variables, and molecule variables. The syntax is the same as above. For example,

```
bar.ORIENT = TAIL
```

requires that `bar` be in the `TAIL` orientation.

5.3.3 Logical conditions

Conditions can be grouped together in braces, and these groupings can be operated on by the logical operators `AND`, `OR`, and `NOT` (anything else?). For example, if we wanted to make sure that `IP3` was not bound to `foo`, we would write

```
NOT { IP3.site == foo }
```

and if we wanted to say that exactly one IP3 had to be somewhere in a foo-bar complex, we would write

```
{
  foo.i == 0
  bar.iA == IP3
  bar.iB == 0
} OR {
  foo.i == 0
  bar.iA == 0
  bar.IB == IP3
} OR {
  foo.i == [IP3,0,0,0]
  bar.iA == 0
  bar.iB == 0
}
```

Groupings can be nested. NOT has higher precedence than AND and OR.

5.3.4 Rates and probabilities

The last statement in a condition block is a rate constant. This will be converted to a probability based on the current time step. For unimolecular reactions, if the rate is known to be constant, one can supply a value for the UNCHANGING variable; FOREVER is the default, while TIME_STEP requires the rate to be recomputed every time step; units are seconds. For example,

```
UNCHANGING = 1.5e-3
RATE = 3.1e5
```

specifies that the rate is known to be constant for at least the next 1.5 milliseconds. (With such a simple expression, the rate is obviously constant for all time, but more complex expressions are possible.) This constancy is used to accelerate the computation of unimolecular reactions.

5.4 Outcomes

Molecules referred to by name in either the trigger section or the condition section are available to reference in the outcome section. As usual, refering to a species when a molecule is expected gives the first instance of a molecule of that species in this context (e.g. foo would mean foo.1).

5.4.1 Molecule creation and destruction

New molecules and defined complexes can be created with the CREATE command. It is a syntax error to create a complex with bonds that are not explicitly defined. For example, if we have

```
DEFINE_COMPLEX foobar1 (foo,bar)
{
  foo.b = bar.f
  foo.i = [IP3,0,0,0]
  bar.iA = 0
  bar.iB = 0
}
```

```
DEFINE_COMPLEX foobar2 (foo,bar)
{
  foo.b = bar.f
  foo.i = [IP3,*,*,*]
  bar.iA = *
  bar.iB = *
}
```

then `CREATE foobar1` would be valid, but `CREATE foobar2` would be invalid. Use a dot after the complex or molecule name followed by a number if you need to keep track of individual molecules or complexes of the same type. Here is an example of molecule creation:

```
DEFINE_REACTION (foo,bar)
{
  CONDITION
  {
    foo.b == 0
    bar.f == 0
    bar.iA == IP3
    bar.iB == 0
    RATE == 3e2
  }
  OUTCOME
  {
    CREATE IP3.2
    bar.iB = IP3.2.site
    . . .
  }
}
```

In order to distinguish between the `IP3` bound to `bar.iA` and the new one, we label the new one `IP3.2`. (The previous one would be `IP3.1`.)

If one creates a new complex, one can refer to parts of it using dot notation of the names of the molecules in it. These are numbered in order of their priority; it is often convenient to step through bond by bond to make sure one hasn't mistaken the priority. For example, we can pick out the `IP3` of `foobar1` that is bound to bar as follows:

```
CREATE foobar1
IP3.3 := foobar1.bar(iA:IP3.3.site)
```

I don't like this at all.

You destroy things with `DESTROY`. If you use `DESTROY_JUST`, it will only destroy the molecule. If you use `DESTROY_FROM`, it will destroy everything connected to the molecule. If you use `DESTROY`, it will destroy everything connected to the molecule that isn't already named. If you use `DESTROY_TO [ ]` it will destroy everything connected to the molecule that isn't listed in the array.

5.4.2 Molecule orientation

The orientation of molecules can be swapped with the `FLIP` keyword, followed by the name of the molecule.

5.4.3 Bond reassignment

Bonds can be reassigned in exactly the way you'd expect. Stuff that ends up disconnected at the end of the outcome block are turned into free molecules or complexes.

5.4.4 Manipulation of variables

Use `typename varname` for user-defined variables. They're created implicitly when first used, and disappear at the end of the outcome block. Scope is local.

5.4.5 Delayed availability of products

I'm not sure we want to implement this. If so, use `INERT time` to specify that the continuation of the outcome block is delayed for that period of time. Maybe this has to be the first thing in the outcome block?

5.5 Subreactions

These are exactly like reactions, except they can only have one trigger molecule (and, optionally, one surface type). They can be referred to by name, or in a nameless `DEFINE_REACTION` block inside an outcome block.

6 Reaction Utility Building Environment

6.1 Types

RUBE is a strongly-typed language with optional automatic typecasting for convenience.

6.1.1 Simple types

There are three simple types in RUBE.

BOOL	Logical value, either TRUE (1) or FALSE (0)
NUMBER	32 bit signed integer
REAL	Double precision floating-point value (64 bit)
STRING	A string; string constants are specified between double quotes.

6.1.2 Complex types

There are six complex types in RUBE.

VEC	Three dimensional vector with real components
MOL	A single specific molecule present in the simulation (derived or primative)
SPECIES	A set of properties shared by molecules of the same type
CPLX	A group of molecules bound together
TILE	A single triangular surface element in a specific location in the simulation
SURFACE	A set of properties shared by tiles of the same type

6.1.3 Derived types

There are three derived types in RUBE.

array	A linear or multidimensional ordered array of the the same type
group	A collection of named primatives of potentially different types
reference	A pointer to some other object–maybe we don't need this?

6.1.4 Static declaration

Any type may be either const or dynamic (the default is dynamic). Constant variables cannot be modified after they have been defined. If a derived type is const, all elements of it are const. If one element of a derived type is const, the other elements may still be dynamic.

6.2 Variables

One can declare new simple and complex variables using C syntax: `type_name variable_name` or optionally, `type_name variable_name = value`

6.2.1 Array notation

Array variables are declared using the notation `type_name variable_name[dim1][dim2]...[dimN]`

Array constants are specified as lists between brackets. These can be nested to form (rectangular) multidimensional arrays.

If `R` is an array, the size of the array is referenced by `R.size`. This will be a number if R is a linear array, or a number array with length equal to the dimension of `R` if `R` has higher dimension.

To index into an array, use a dot followed by the index; if a multidimensional array is dereferenced with nothing between dots, the subarray will be returned. For example,

```
REAL two_by_two[2][2] = [ [1.7 2.9e3] [3.5 9.4e2] ]
REAL R[4][9][3]
REAL r = R.2.3.1
REAL r2[9] = R.3..2
REAL r2b[3] = R.4.7
REAL rsize[3] = R.size
REAL rdim = R.size.size
```

Indexing starts at 1. One can allocate array sizes dynamically based on the values of variables of type number.

If an array is of size 1 it can collapse to the type that composes it (automatic typecasting).

Arrays of the same size with the same component type are considered to be of the same type.

6.2.2 Group notation

Group variables are declared using the notation `type_name variable_name(type_name field_name,...,type_na` `field_name)`.

Group constants are specified as lists of `field_name : value` pairs separated by commas, with the list surrounded by parentheses. A single member of a group can be referred to using `variable_name.field_name`.

Groups with the same fields with the same names are considered to be of the same type.

6.2.3 Alias notation

Aliases are similar to typedefs–need more work to explain how to use them exactly (and to figure out what they are and if we need them).

6.2.4 VEC

Vectors are groups of the form `(real x, real y, real z)`.

6.2.5 MOL

Molecules are groups whose binding sites are fields of type molecule, plus anything declared as VARIABLE for that molecule type. They also have the following fields:

```
const vec location
const species type
const cplx parent
(others?)
```

6.2.6 SPECIES

Ugh.

6.3 Operators

6.4 Execution control

6.4.1 If-else

6.4.2 While

6.4.3 Foreach

6.4.4 Goto

6.5 Functions

6.6 Standard Function Library

6.7 Examples

7 Abbreviated Notation

7.1 Reaction Groups

Abbreviated notation may be used to define reactions within a reaction group. Reaction groups consist of a the keyword `REACTION_GROUP` followed by the name of the group followed by a list of abbreviated reactions in braces. `DEFINE_REACTION` blocks may occur inside `REACTION_GROUP` blocks. `REACTION_GROUP` blocks may be nested.

7.2 Abbreviated Notation for Primative Molecules

7.2.1 Basic syntax

Reactants (trigger molecules) are listed on the left separated by `+` signs, followed by the reaction arrow `->`, followed by what is left at the end of the outcome on the right. Rates are indicated in brackets at the end of the line. An optional name can be given at the end following a colon. For example:

```
DEFINE_MOLECULES {
  A {...}
  B {...}
  C {...}
}
REACTION_GROUP simple
{
  A + B -> C [3.1e2] : rx1
}
```

This takes the primative molecules `A` and `B`; when they collide they have some probability of being annihilated and forming the primative molecule `C`.

7.2.2 Orientation specification

By default, all molecules are in the same orientation class. The prefix operator ~ refers to the opposite orientation of this default class. (The default is `:1` in full notation; ~ is `:-1`.) One can use postfix operators `'` and `,` to refer to the second orientation class (`:2` and `:-2`), and likewise with ■ and ■ (`:3` and `:-3`), and likewise with ■' and ■, and so on. For example:

```
A + ~B -> A' + B' [2.9e1]
```

requires `A` and `B` to be in opposite orientations, and after the reaction they have the same orientation (but that orientation is chosen randomly with respect to their starting orientations).

7.2.3 Implicit destruction and creation of molecules

If a molecule is listed on both the left and right side of an arrow, it persists (though possibly flipped if the orientation has changed). If a molecule is listed on the right and not on the left, it is created. If it's listed on the left and not on the right, it's destroyed. If one molecule is listed on the left, but two or more of that molecule are listed on the right, the first one persists and further ones are created. For example:

```
A + B -> ~A + A' + C [1.7e2]
```

This reaction takes the original `A` and `B` and destroys `B`. The original `A` is flipped, a new `A` is created in random orientation, and a new `C` is created with the same orientation as the original `A` and `B`.

7.2.4 Catalytic reactions

Catalytic reactions combine the products and declare them INERT for a time determined by the second rate listed in brackets. These are specified by placing the catalyst in the middle of the arrow, or at the left followed by a colon:

```
A -- B -> C [2.9e5,1.1e2]
B: A -> C   [2.9e5,1.1e2]
```

These reactions are identical: both have `B` as a catalyst, converting `A` to `C` with km of `2.9e5` and kcat of `1.1e2`.

7.2.5 Reversible reactions

Reversible reactions must have no more than two molecules on either side. If this is the case, a reversible reaction can be specified by using a double-headed arrow and specifying two rate constants (with > and < inside the bracket indicating forward and reverse rate, respectively). Catalytic reactions can also be reversible.

```
A + B --> C [3.1e2] : fwd1
C --> A + B [1.1e2] : bkw1
A + B <--> C [>3.1e2][<5.9e1] : fwbw1
B: A <--> C  [>2.9e5,1.1e2][<8.5e3,1.1e2]
A <- B -> C  [>2.9e5,1.1e2][<8.5e3,1.1e2]
```

The top examples show forward and reverse reactions `fwd1` and `bkw1`, followed by a shorthand notation for both reactions `fwbw1`. The bottom two examples show two ways of writing reversible catalytic reactions.

7.3 Abbreviated Notation for Derived Molecules

Derived molecules are created when their components are listed on the left and they are listed on the right. For example:

```
DEFINE_MOLECULES {
  A {...}
  B {...}
  C (;A,B) {...}
}
REACTION_GROUP example {
  A + B -> C [3.1e2]
  C -> A + B [1.1e2]
  C -> A     [5.2e1]
}
```

Unlike the previous example, where `C` was a primative molecule, here `A` and `B` are assembled into `C`. In the second reaction, the reverse occurs (`C` is disassembled). In the third, `C` is disassembled and the `B` that was in `C` is destroyed. If the molecules had any binding sites, whatever was bound to them would be reassigned according to the rules specified in the definition of the derived molecule.

7.4 Abbreviated Notation for Complexes

7.4.1 There are no complexes, only molecules

Since abbreviated notation has no extended conditions block (just a rate and possibly an inert time), abbreviated notation will act on any molecule whether inside or outside a complex. Make sure that this is what you want! It may be advisable to have molecules undergo a state change (i.e. be converted to a different molecule) when binding to complexes if their properties radically change. Alternatively, you may wish to make the product a derived molecule.

7.4.2 Binding reactions

If you have two molecules that bind each other, you can specify this by listing them on the left, and the (subset of the) complex that they make on the right. Alternatively, if you list a bound pair of molecules on the left, they can be dissociated on the right. For example:

```
DEFINE_MOLECULES {
  A(s) {...}
  B(s) {...}
}
REACTION_GROUP example {
  A + B -> A.s(B.s) [1e5]
  A.s(B.s) -> A + B [1e2]
}
```

If the bond is unavailable, the reaction won't proceed.

7.4.3 Sophisticated reactions

You can list complexes on the left hand side of a reaction; the reaction will only proceed if the complex is found. Anything listed can be broken apart into pieces or destroyed; if you want something more complex, use the longhand notation. That's what it's there for!

8 Examples

8.1 Sodium-Potassium Pump

```
DEFINE_MOLECULES {
  ADP {...}
  Pi  {...}
  ATP (;ADP,Pi) {...}
  K(s) {...}
  Na(s) {...}
  pump( k[2] , na[3] ) {...}
}
REACTION_GROUP exchange {
  ADP + Pi -> ATP              [...]        : regenerate
  pump + K <-> pump.k(K.s)     [>...][<...] : bind_K
  pump + ~Na <-> pump.na(Na.s) [>...][<...] : bind_Na
  pump(k:[K.1,K.2],na:[Na.1,Na.2,Na.3]) + ~ATP ->
     pump(0,0) + ~K.1 + ~K.2 + Na.1 + Na.2 + Na.3 + ~ADP + ~Pi [...] : do_it
}
```

8.2 CaM Kinase II

We'll assume that CaMKII has two phosphorylation sites that must be phosphorylated in order; this can only happen if two neighboring subunits (of a 7-member ring) are bound with CaM that has both N calcium binding sites bound.

```
DEFINE_MOLECULES {
  ADP {...}
  Pi(s)  {...}
  ATP (;ADP,Pi) {...}
  Ca(s) {...}
  CaM(k,n:2,c:2) {...}
  CaMKII(L,R,c,p1,p2) {...}
```

```
}
REACTION_GROUP simple {
  ADP + Pi -> ATP [...]
  CaM + Ca <-> CaM.n(Ca.s) [> [on1 on2].CaM.n] [< [off1 off2].CaM.n]
           <-> CaM.c(Ca.s) [...]
  CaMKII + CaM <-> CaMKII.c(CaM.k) [...]
}
DEFINE_REACTION assembly (CaMKII,CaMKII)
{
  CONDITION {
    { CaMKII.1.L==0
      CaMKII.2.R==0
    } OR {
      CaMKII.1.R==0
      CaMKII.2.L==0
    }
    COUNT(CaMKII.1,CaMKII) + COUNT(CaMKII.2,CaMKII) <= 7
    RATE = ...
  }
  OUTCOME {
    IF ( CaMKII.1.L==0 ) {
      CaMKII.1.L = CaMKII.2.R
    } ELSE {
      CaMKII.1.R = CaMKII.2.L
    }
    IF ( COUNT(CaMKII.1,CaMKII)==7 ) {
      MOL left = CaMKII.1
      MOL right = CaMKII.1
      WHILE ( left.L != 0 ) { left = left.L }
      WHILE ( right.R != 0 ) { right = right.R }
      left.L = right.R
    }
  }
  FUNCTION is_pair (MOL m) -> (BOOL b)
  {
    b = (m IS CaMKII(L:CaMKII(c:CaM))) OR (m IS CaMKII(R:CaMKII(c:CaM)))
  }
  DEFINE_REACTION phos1 (CaMKII,ATP)
  {
    CONDITION {
      CaMKII.1.p1 == 0
      CaMKII.1.c == CaM
      is_pair(CaMKII.1)
      RATE = ...
    }
    OUTCOME {
      REACT { ATP -> ADP + Pi }
      CaMKII.1.p1 = Pi.s
    }
  }
  DEFINE_REACTION phos2 (CaMKII,ATP)
  {
    CONDITION {
      CaMKII.1.p2 == 0
      CaMKII.1.p1 == Pi
      CaMKII.1.c = CaM
      is_pair(CaMKII.1)
      RATE = ...
    }
    OUTCOME {
      REACT { ATP -> ADP + Pi }
      CaMKII.1.p2 = Pi.s
    }
  }
}
```