Vesimulus-Documentation

Tihamér Geyer

tihamer.geyer@bioinformatik.uni-saarland.de
Zentrum für Bioinformatik, Universität des Saarlandes, D–66041 Saarbrücken

V2.1, July 2012

Overview

Vesimulus is a simulation package for molecular stochastic simulations of metabolic or regulatory biological systems. It was developed because there was no software available which could perform simulations of how the proteins of the bacterial photosynthetic apparatus work in detail. These origins date back to 2005, when Florian Lauck set up the very first version during his bachelor thesis. Since then vesimulus was used to set up a consistent dynamic model of the bacterial photosynthesis including a complete parameterization in another two bachelor theses by Sarah Blass and Xavier Mol and also as a tool and testbed to study how to bridge the gap in scales between computational approaches from the molecular and the systems biological regimes.

The underlying "pools-and-protein" approach builds on the molecular biological descriptions of the considered enzymes by directly implementing the elementary binding, charge transfer, or conformational reactions as stochastic one-molecule-at-a-time processes. From these, encapsulated protein objects are implemented. Then, at runtime, the required number of independently working copies is instantiated. These proteins are connected to the metabolite pools via standardized connectors to build up the biological system under consideration. On the software side we make use of the object orientation of C++ in that the pools and all types of proteins are independent objects communicating via well defined methods. Thus the simulation is modular and can easily be extended by adding new classes for not yet considered protein types. The actual biological system, i.e., the stoichiometries and types of the proteins, the connecting pools and their sizes, the observable system parameters, and any dynamic changes during the simulation, are then read from a configuration file, which is parsed at runtime.

The rationale behind this design choice to hard code the proteins and only specify their numbers and connectivities at runtime was that the knowledge about the detailed inner workings of a protein change relatively slowly while parameters like reaction rates, initial conditions, or illumination profiles may be set to different values at each run of the simulation. Consequently, this documentation is organized as follows. We start by explaining the general concepts and how to set up a simulation. This section introduces the propagation techniques and the basic properties of the protein, pool, and observable classes. Next we explain how to extend the simulation engine by the proteins that you are actually interested in. Finally, the "Examples and Tutorials" section presents some actual simulations and their analysis.

Setting up a Simulation

Installing Vesimulus

The current version of the vesimulus engine including this documentation and the tutorial files can be downloaded from **`http://service.bioinformatik.uni-saarland.de/vesimulus`**. On this page you will also find updates to the simulation code and — hopefully soon — new proteins provided by us and other users.

Unpack the archive, check the Makefile for system specific setting, and compile by typing **`make`**. For this you need an ANSI C++ compiler (g++ works fine) and the Gnu Scientific Library for the random numbers. If everything works as planned you'll find an executable **`vesimulus`** in the build directory. You're set. A lot of documentation is included in the sources and can be read conveniently by running **`doxygen`** on the sources: type **`make doxy`** and open the file **`doc/html/index.html`** in your favorite web browser.

Defining the Simulation Setup

The simulation engine vesimulus knows about all proteins that were supplied during compilation. To actually start a simulation, you specify the length of the simulation (in seconds), the output interval (also in seconds), the length of the integration time step (in milliseconds), and the name of a setup file which contains the information about the proteins, the pools, the observables, and any changes during the simulation. These arguments are given on the commandline, like, e.g.:

```
$> ./vesimulus 10 1 10 fullVesicleTemplate.ves
```

This example specifies a simulation of a complete chromatophore vesicle that runs for 10 seconds at a time step of 10 ms and lists the pool occupancies and observables specified in the setup file to the terminal every second. An additional string argument is taken as the basename of an output file. Then, the console will only show an ASCII art progress bar. If the second optional argument is present (its actual value is ignored) then also the internal states of the proteins will be listed into a separate file for each copy of the proteins. With the above example setup, the following command

```
$> ./vesimulus 10 1 10 fullVesicleTemplate.ves fish dummy
```

will produce an output file called **`fish_pools.dat`** with the same text as was printed above to the terminal and, as the last argument (here: "dummy") is also present, many files of the form **`fish_[ProteinType]_[index]_internals.dat`** listing the changes inside the proteins.

Most of the simulation setup is defined in the setup file. It has the following structure. Except for empty or comment lines, which are denoted by a hash sign (#) at the beginning of the line, each line starts with a keyword which is **`protein:`** (note the trailing colon!) when a protein is to be defined, **`pool:`** for a pool, and, obviously, **`observable:`** for an observable. Additionally, the keyword **`change:`** indicates that the concentration in a pool should be changed at a certain time point. Generally, each object (= protein, pool, or observable) is defined on a line on its own. Thus, when ten copies of a certain protein are required, ten definition lines occur in the setup file.

The Pool Class

The most simple objects are the pools. These are passive containers of particles, which know their volume and the number of particles therein. Additionally, a pool can be defined as infinitely large. Then, the particle number does not change when proteins interact with this pool. Examples are the pool modeling the incident light for photosynthesis simulations or the large reservoir of protons in the cytoplasm. Consequently, a pool is defined in the setup as shown in this example, where the last keyword specifies whether the pool content is given in the output as particles, as concentration, or not at all (indicated by **`part`**, **`conc`**, or **`none`**, respectively).

```
#pool:    name   infinite?    particles    volume    output
pool:     Q      false         15          5.28e3    part
pool:     QH2    false        185          5.28e3    part
```

In this example the first line is a comment. The other two lines specify one pool for quinones (Q) and one for quinols (QH2). Both pools have a finite volume ("false" in the third column) of 5.28e3 nm^2 and are listed with their particle numbers. Initially, the Q pool contains 15 particles while the QH2 pool is initialized with 185.

Note that there are no units implied. In our simulations the pools for the membrane bound Q and QH2 were assumed to be essentially two-dimensional. You have to make sure that the relevant rate constants have consistent units! In this case an association rate from the pools onto a protein must have units of $nm^{-2}\,s^{-1}$.

All pools in a simulation are identical from the program side. It is only the proteins that they are connected to that give each pool its biological identity.

The Protein Class

The proteins are the central active parts of a simulation as they process and produce metabolites. In a **`protein:`** definition line the next two tokens after the keyword specify the protein type (the currently known types are explained below) and a numerical index which is used to generate a unique label for this protein copy. The rest of the line is specific information for that protein type. In the simulation setup, this part is passed on to the protein which has to do the parsing itself. This specific part consists of pool names that the protein is connected to, the respective association and dissociation constants, and any other parameters required by the protein.

At each time step during the simulation the central propagation loop calls the **`timestep()`** method of the proteins one after the other. This method, in turn, loops over all reactions of the respective protein. Each reaction then first checks whether the conditions are fulfilled that it might take place. For an association reaction this for example includes a test whether the binding site is empty, i.e., available for a new binding event, while for an electron transfer the donor has to be reduced and the acceptor oxidized. When this test is passed the actual reaction is considered.

For an **association reaction** to a binding site the protein needs to know the (name of the) pool and the association constant. The association constant k_{on} is then handed to the pool (by calling the **`take_out(k_on)`** method of the respective pool), which in turn determines the association probability P_{on} from its concentration ρ (particles per volume) and the time step Δt as

$$P_{on} = k_{on}\,\rho\,\Delta t$$

This probability is compared to a random number *r* from [0, 1] and when $P_{\mathrm{on}} \leq r$, the pool indicates that the `take_out()` was successful. The protein now has to adjust its registers indicating the states of the binding site and any potentially affected centers. An example of an association is this following (slightly reformatted) code snippet from the bacterial reaction center class that handles the binding of an oxidized quinone to the Q_b binding site. First, the condition that Q_b be empty is checked (`== false`), then the pool is queried with the respective k_{on}, and when the pool returns `true`, the status of the binding site is updated to occupied.

```
// reaction that binds a Q to the RC, if Qb is empty
void ProteinRC::reaction1(Protein *p) {
    if (!bs_Q) {
        if (QPoolp->take_out(Q_kon)) {
            bs_Q = true;
            writeInternals();
        }
    }
}
```

By calling `writeInternals()` the changes (that a Q has bound) are written to the respective log file if desired (see above). Otherwise this function just returns.

For reactions that are independent of any metabolite concentrations like dissociation, internal charge transfer, or conformational changes, a different strategy is used. When at such a reaction the respective conditions are met, a waiting time is calculated from the rate and used to initialize a timer which then triggers the actual reaction. This is shown in the following code snippet that handles the electron transfer from a bound cytochrome c_2 to the special pair and the subsequent (slower) dissociation from the reaction center:

```
// Transfers an electron from c2 to special pair and unbinds c2,
// takes back the virtual proton from the HInternal pool.
// The e- transfer occurs before the c2 unbinding starts
void ProteinRC::reaction4(Protein *p)
{
    if(bs_C)
    {
        switch(timer_r4)
        {
            case 0:
                // initialize timer, make sure its value is at least 1
                double t0 = 1.0/(c2ox_koff*delta_t);
                timer_r4 = static_cast<unsigned long int>(exp_ran(t0));
                if(timer_r4 == 0) { timer_r4 = 1; }
                // charge transfer c2 -> P
                HInternalPoolp->take(1L);
                reg_SP = 1;
                writeInternals();
                break;
            case 1:
                c2oxPoolp->put_back();
                bs_C = false;
                writeInternals();
                timer_r4 = 0;
                break;
            default:
                timer_r4--;
        }
    }
}
```

For such timer controlled reactions, a timer value of 0 indicates that the reaction was in its "rest state", i.e., not running. It can then be initialized and decremented in subsequent calls. When implementing such functions take care that the conditions that triggered the reaction can not be altered by some other reaction, or the timer would be stuck until the conditions become true again. This might lead to an unexpected behavior.

In this example a pool called **`HInternalPool`** is used to account for the displacement of the charge of the electron during the photo-induced oxidation of the special pair, as this process contributes to the electric part of the transmembrane potential.

More details on the methods that make up a protein class are described in the section on how to implement a protein yourself further down.

The Observable Class

An observable is a system wide accessible query function. With an observable for example an experimentally accessible quantity like the transmembrane potential or the average oxidation state of all cytochrome *c* can be determined. Observables can be queried both from the proteins and from the output routine of the simulation engine. They are not called from the central propagation loop itself.

Due to their nature as system wide "variables" there can only be one instance of any given observable in the simulation. Also, the observables are passive in the sense that they do not alter any pool occupancies or protein states.

Constants

Constants canbe defined in the setup file with the **`constant`**: tag. The first token after this keyword is used as the constant's name and the rest of the line as its value. Constants can then be referenced further down in the setup file by prefixing it with a $ sign. They work similar to shell variables (dumb text replacement). Just make sure that one constant name is not a substring of another constant!

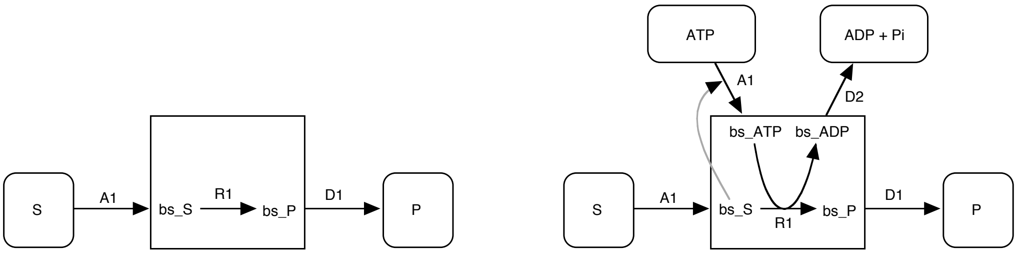
Some basic calculations can be done with the **`evaluate:`** keyword. The first token is taken as a variable name and the rest of the line is piped through Perl for evaluation after potential constants expansion.

```
#constant:   label        value
constant:    TEN          10
constant:    FLOAT_TEN    $TEN.0
constant:    OUTFILE      fish_parm$FLOAT_TEN.txt
evaluate:    THOUSAND     $TEN * $TEN * $TEN
evaluate:    SQRT2        sqrt(2.0)
```

How to Write Your Own Protein/Observable

As an example for how to implement a new protein we will use a simple example of an enzyme that converts a substrate S into a product P. The first iteration will reproduce the effective Michaelis-Menten kinetics. In a second iteration we add ATP consumption. More specifically, ATP can only bind after the substrate has bound and is released from the enzyme independently of the product after the conversion has occurred.

The first version of the new enzyme consequently needs two binding sites, bs_S for the substrate and and bs_P for the product, with their respective association and dissociation reactions A1 and D1 and the internal conversion reaction R1 (see the following figure). The more elaborate second version additionally needs binding sites for ATP and ADP, an association reaction A2 which depends on the state of bs_S, and a second dissociation reaction D2. Of course, the names for the binding sites and reactions itself do not have any meaning and they can be chosen according to any other convetion, too.



In the above figure the protein is indicated by the rectangle and the pools by the rounded rectangles. As can be seen, the first version needs three rate constants and the names of the substrate and the product pool, i.e., a total of five parameters. The more complex second version requires five rates and four pool names. Note that here each reaction follows the simple mass action kinetics.

To implement our new protein, which will be called MMEnzyme (for Michaelis-Menten-Enzyme), we start from the supplied connector protein which takes particles out of a pool and releases them into another pool. So, go to the directory with the protein sources (proteinsource), copy proteinConnector.cpp and proteinConnector.h into proteinMMEnzyme.cpp and proteinMMEnzyme.h, and open them in your favorite text editor.

To avoid name clashes, search and replace any occurrences of ProteinConnector by ProteinMMEnzyme. Don't forget the #ifdef in the header file and the #include line in the source file. Also, chop out (or replace) the description in the header file.

Note that comments starting with //! or /*! are used by doxygen to generate a documentation from the sources.

Register the Protein/Observable

To register a new protein with the parser, a shell script parses all header files for tags LABEL: and CLASSNAME: and constructs a part of the source code for the parser from the information found there. Replace the information of the connector such that the first lines of proteinMMEnzyme.h look like the following:

```
// Definitions for the automated inclusion into the parser
// LABEL:        MMEnzyme
// CLASSNAME:    ProteinMMEnzyme
```

Now issue a **`make clean; make`**. The new protein is now known to the simulation.

Parameter Parsing

During setup, the main parser reads the configuration file and instantiates a new copy of the **`MMEnzyme`** for every line that starts with **`protein: MMEnzyme`**. The next number after the protein type is used to create a unique label by appending it to the protein type. The rest of the line is passed on to the protein for parsing. We therefore need a method that extracts the protein specific parameters from the setup line. Assume the following line to define one copy of our new MMEnzyme:

```
protein:  MMEnzyme 1   Substrate  2e3   8   Product  120
```

This shall define that the substrate is taken up from a pool called "Substrate" with a rate of 2 x 10^3 nm^3s^{-1}. The internal reaction has a speed of 8 s^{-1}, and the dissociation of the product into the pool "Product" takes place with a k_{off} = 120 s^{-1}. Consequently, the protein needs two pointers to the two pools and three variables to hold the rates. These are defined in the **`private`** section of the header file (note the doxygen related additions to the comments):

```
//! @name Pointers to the two pools and variables for the uptake constants
//@{
Pool *SubstratePoolp, *ProductPoolp;
double kon_S, koff_P, k_conv;
//@}
```

During setup, additional variables are used to temporarily store the names of the pools until their definitions will be read later in the setup file. These are also defined in the **`private`** section of the header file:

```
//! @name Char[]s for the pool names (see length above)
//@{
char SubstratePool[MAX_POOLNAME_LENGTH],
     ProductPool[MAX_POOLNAME_LENGTH];
//@}
```

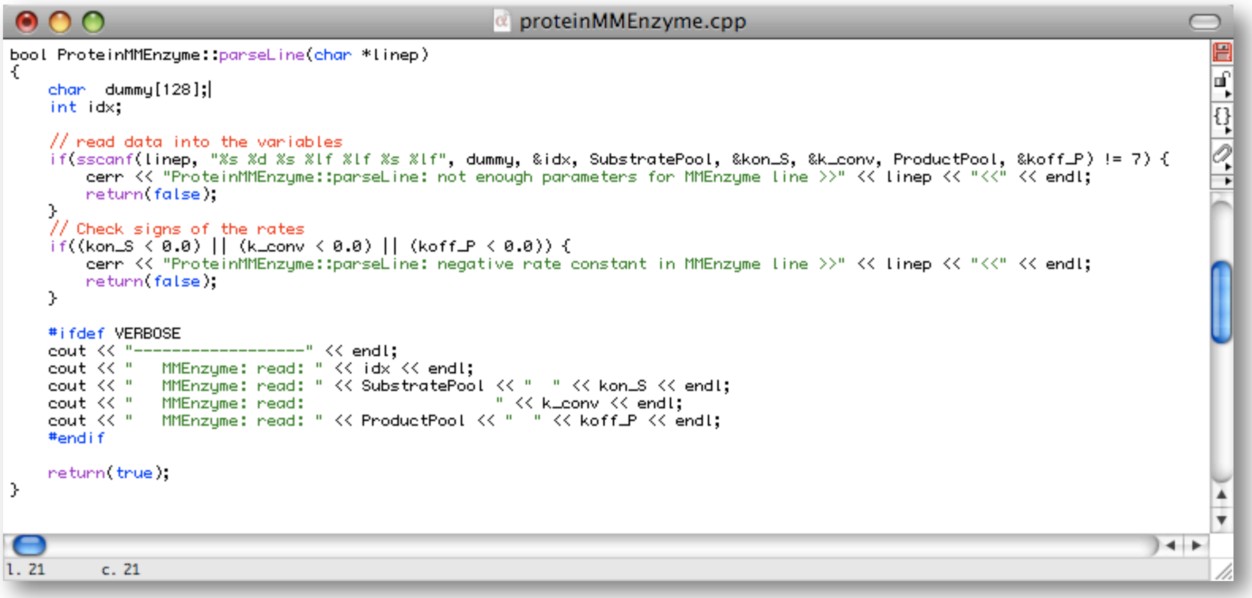
Then, the binding sites have to be defined. As binding sites can either be empty or occupied, we use boolean variables for them:

```
// Defs of the internal registers and binding sites.
bool bs_S, bs_P;
```

Finally, we need two timers for the internal conversion from substrate to product and for the density independent dissociation of the product:

```
//! Timer variables for some of the reactions
unsigned long int timer_R1, timer_D1;
```

The actual parsing is performed by the method **`parseLine()`**. Switch to the source file **`proteinMMEnzyme.cpp`**, locate this function, and identify the line where the setup line is parsed with **`sscanf()`**. Adapt the format string and the target variable, the sign test for the rates, and the debug output. Then, **`parseLine()`** should look similar to the following screenshot:



```
bool ProteinMMEnzyme::parseLine(char *linep)
{
    char  dummy[128];
    int idx;

    // read data into the variables
    if(sscanf(linep, "%s %d %s %lf %lf %s %lf", dummy, &idx, SubstratePool, &kon_S, &k_conv, ProductPool, &koff_P) != 7) {
        cerr << "ProteinMMEnzyme::parseLine: not enough parameters for MMEnzyme line >>" << linep << "<<" << endl;
        return(false);
    }
    // Check signs of the rates
    if((kon_S < 0.0) || (k_conv < 0.0) || (koff_P < 0.0)) {
        cerr << "ProteinMMEnzyme::parseLine: negative rate constant in MMEnzyme line >>" << linep << "<<" << endl;
        return(false);
    }

    #ifdef VERBOSE
    cout << "------------------" << endl;
    cout << "    MMEnzyme: read: " << idx << endl;
    cout << "    MMEnzyme: read: " << SubstratePool << "    " << kon_S << endl;
    cout << "    MMEnzyme: read:                        " << k_conv << endl;
    cout << "    MMEnzyme: read: " << ProductPool << "    " << koff_P << endl;
    #endif

    return(true);
}
```

Check for Pools and Observables

When all the parsing and initialization is done for all proteins, pools, observables, and changes, then the proteins and pools can be connected. For this, the parser calls the method **`verify_pools()`** of the proteins (and of the observables, too, but this is a different section of this document). During the pool setup, the parser collects a map with the pool names and the pointers to the respective pools. This map is now queried to convert the stored pool names from the protein's setup line into the correct pointers. For each pool that the protein needs to know, there is a section like the following (but without the line break in the error message):

```
if(!(Pools->count(string(SubstratePool)))) {
    cerr << "ProteinMMEnzyme::verify_pools: pool "<<SubstratePool<<" not defined" <<
endl;
    exit(-3);
}
SubstratePoolp = getPool(SubstratePool);
```

When a pool is not found the simulation is terminated the hard way as there is no chance to reliably guess what would have been the correct pool label in the setup file. This is left to the user to correct and try again.

If the protein also needs to access observables, a similar code fragment is used. Check, e.g., the BC1Dimer_2 for how to test for observables. Now the code used in the model setup phase is run and the actual simulation will start.

The Elementary Reactions

During the simulation, the main loop of the simulation class calls the protein's **`timestep()`** method, which in turn calls all the functions of the protein. Before these can be called, they have to be defined and registered. Registration of the functions is done in the constructor.

For our simple MMEnzyme we need three reactions which will be named reaction_A1, reaction_R1, and reaction_D1. For each of these the following code is used to add them to the protein's map of functions.

```
void (Protein::*pr1)(Protein *) = (void(Protein::*)(Protein *))
                                  (&ProteinMMEnzyme::reaction_A1);
fu1.pfunction = pr1;
setFunction("A1", fu1);
```

The timers for the conversion and the dissociation reaction are also initialized to 0 in the constructor.

The corresponding reaction is defined further down in the source file. They also have to be declared in the header file. The biological condition is that the active center where the conversion takes place is empty. Here, the state of the molecule under conversion is encoded in the currently occupied binding site. In the protein function, we therefore have to check that both binding sites are empty. Then the substrate pool is queried and, when successful, bs_S is switched to true.

```
// A1: association of the substrate
void ProteinMMEnzyme::reaction_A1(Protein *p) {
    if(!bs_S && !bs_P) {
          if(SubstratePoolp->take_out(kon_S)) {
                 bs_S = true;
                 writeInternals();
          }
    }
}
```

The logging of the internal states via **writeInternals()** will be explained later. We don't have to change the particle number in the pool, this is handled by the pool's **take_out()**. The next step is the conversion of the substrate into the product. Here, we use a timer. To denote that the conversion is running we flip the product binding site to occupied as soon as the timer is initialized. The substrate binding site is released only when the conversion is finished, i.e., when the timer is down to 1. To make sure that the reaction takes place even when the rate constant is set very high or the timestep is long (which lead to a very small t0), we should verify that the timer timer_R1 is set to value of at least 1.

```
// R1: conversion of substrate in bs_S into product in bs_P
void ProteinMMEnzyme::reaction_R1(Protein *p)
{
    if(bs_S)
    {
        switch(timer_R1)
        {
            case 0:  // initialize timer, start conversion
                double t0 = 1.0 / (k_conv * delta_t);
                timer_R1 = static_cast<unsigned long int>(exp_ran(t0));
                if(timer_R1 == 0) { timer_R1 = 1; }

                bs_P = true;
                writeInternals()
                break;

            case 1:   // conversion done
                bs_S = false;
                timer_R1 = 0;
                writeInternals();
                break;

            default:    // count down
                timer_R1--;
        }
        }
    }
```

Now we only need the last reaction, the unbinding of the product. This can start as soon as the conversion is over, i.e, as soon as bs_S == false. The same timer structure is used with timer_D1:

```
// D1: dissociation of the final product
void ProteinMMEnzyme::reaction_D1(Protein *p)
{
    if(!bs_S && bs_P)
    {
        switch(timer_D1)
        {
            case 0:  // initialize timer
                double t0 = 1.0 / (koff_P * delta_t);
                timer_D1 = static_cast<unsigned long int>(exp_ran(t0));
                if(timer_D1 == 0) { timer_D1 = 1; }
                break;

            case 1:   // release!
                bs_P = false;
                ProductPoolp->put_back();
                timer_D1 = 0;
                writeInternals();
                break;

            default:    // count down
                timer_D1--;
        }
    }
}
```

Now the product is released and both binding sites are empty again and the enzyme is ready to process the next product molecule in its active site. An alternative implementation could have been with a single binding site plus a variable denoting the current state of the substrate (substrate or product). R1 then would toggle that state-variable.

Output

To watch the **`MMEnzyme`** process its substrate we can observe how the numbers in the substrate and product pools change. Even more insight (and debugging information) is gained when we monitor the occupation of the binding sites, too. For this, we need to fill in the two logging methods **`listInternals()`** and **`writeInternals()`**.

During setup, **`listInternals()`** is called which creates a string containing a comment/header line with the names of the internal sites for the respective output file, while **`writeInternals()`** writes out the actual values separated by tabs into the log file (the name of which was set during setup).

```
std::string ProteinMMEnzyme::listInternals() {
    return(string("#time\tbs_S\tbs_P"));
}
```

writeInternals() also contains the test whether the changes should actually be logged or not. For this the global variable **`outputInternals`** is queried.

```
void ProteinMMEnzyme::writeInternals() {
    if(outputInternals) {
        double realTime = timer * delta_t;
        internalsOutFile.open(internalFileName.c_str(), fstream::app);
        internalsOutFile << realTime << "\t" << bs_S << "\t" << bs_P << "\n";
        internalsOutFile.close();
    }
}
```

MMEnzyme V2.0

The protein defined above can now be extended as explained previously by adding binding sites for ATP and ADP together with the respective on- and off-reactions (and the respective pool variables etc). To distinguish between the two version, the expanded form can be called **MMEnzyme2** (both as class and as label for the setup). Then one can compare the two forms or even use them in the same simulation for two different protein types. One question could for example be under which conditions the ATP supply can be neglected to simplify the setup.

When in the real enzyme the ATP binding may only occur after the substrate has bound, then the corresponding test in the ATP binding reaction would be

```
if(!bs_ATP && bs_S) { ...bind ATP... }
```

If the other reactions are encoded correctly, we don't have to test whether the ADP binding site is empty, because substrate binding would only occur after ADP has dissociated. Setting up these conditions for more complex biological scenarios can sometimes even reveal what all is not yet known about a specific protein. However, one may implement different formulations of the reaction conditions and check whether they behave differently under specific conditions, which can then be investigated in an experiment.

Additionally, the main conversion reaction R1 in the expanded version of the enzyme would only start when ATP has bound:

```
if(bs_S && !bs_P && bs_ATP) { ...perform R1... }
```

Examples and Tutorials

For the following tutorials it is assumed that you successfully compiled the simulation engine and that you walked through the previous section where the implementation of a new protein was explained on the "MMEnzyme". If you did not do that yet then you can copy the two files **`proteinMMEnzyme.cpp`** and **`proteinMMEnzyme.h`** from the Tutorials folder into the main folder and recompile vesimulus (type **`make clean; make`** to register the new protein).

Characteristics of the MMEnzyme Model Protein

As a minimal setup to test the newly implemented MMEnzyme protein model, we need one enzyme and two pools, one for the substrate and one for the product. To initialize the enzyme we use the setup line quoted above. The two pools are set up as follows.

```
pool:      Substrate    true   1000  1e5    part
pool:      Product      false  0     1e5   part
```

Both pools have a volume of 10^5 nm^3 and are listed in the output with particle numbers. Initially, the product pool is empty, whereas the substrate pool is kept fixed at 1000 particles, i.e., at a density of 10^{-2} nm^{-3} (have a look at the supplied file **`MMEnzyme.ves`** in the **`Tutorials`** folder). Now run the simulation with this one enzyme for ten seconds, an output interval of 1 second, and a time step of ten milliseconds:

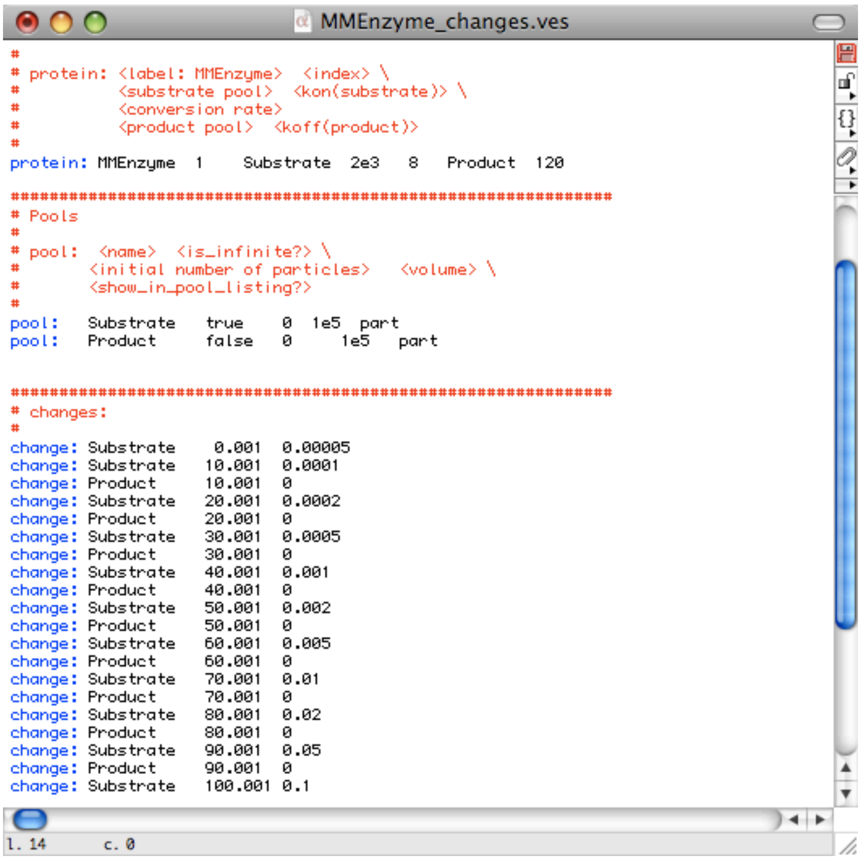
```
$> ../vesimulus  10  1  10  MMEnzyme.ves
```

The console output shows that the one enzyme produces about 60 product molecules during the ten second long simulation. Rerun the simulation a few times to get a feeling for the variation of the total turnover.

Next we want to verify that the enzyme actually follows a Michaelis-Menten kinetics for the steady state throughput with the substrate concentration. For this we can run the simulation for various initial values of the substrate pool and record the total turnover. Another option is to change the substrate level during the simulation. To do so, set the initial particle number in the substrate pool to zero and add lines like the following at the end of the setup file which repeatedly increase the substrate concentration and reset the output accumulated in the product pool:

```
change:    Substrate    10   0.0001
change:    Product      10   0
change:    Substrate    20   0.0002
change:    Product      20   0
     :
change:    Substrate   130   1.0
change:    Product     130   0
```

Continue to increase the concentration every ten seconds using concentrations of 0.0005, 0.001, 0.002, 0.005, 0.01 … 10.0. Then run the simulation until 170 seconds with an output interval of 10 seconds. The turnover is zero except for the last ten seconds interval. This is because in the simulation code changes to the pool concentrations are performed before the output. To overcome this, set the pools at 0.001, 10.001, 20.001, etc seconds and watch the Michaelis-Menten characteristic unfold. Your setup file should now look like the following screenshot.



```
#
# protein: <label: MMEnzyme>  <index> \
#          <substrate pool>  <kon(substrate)> \
#          <conversion rate>
#          <product pool>  <koff(product)>
#
protein: MMEnzyme  1    Substrate  2e3   8   Product  120

##############################################################
# Pools
#
# pool:  <name>  <is_infinite?> \
#       <initial number of particles>   <volume> \
#       <show_in_pool_listing?>
#
pool:   Substrate   true    0  1e5  part
pool:   Product     false   0     1e5   part


##############################################################
# changes:
#
change: Substrate    0.001  0.00005
change: Substrate   10.001  0.0001
change: Product     10.001  0
change: Substrate   20.001  0.0002
change: Product     20.001  0
change: Substrate   30.001  0.0005
change: Product     30.001  0
change: Substrate   40.001  0.001
change: Product     40.001  0
change: Substrate   50.001  0.002
change: Product     50.001  0
change: Substrate   60.001  0.005
change: Product     60.001  0
change: Substrate   70.001  0.01
change: Product     70.001  0
change: Substrate   80.001  0.02
change: Product     80.001  0
change: Substrate   90.001  0.05
change: Product     90.001  0
change: Substrate   100.001 0.1
```

Save the output into a file and plot the third vs. the second column with logarithmic axes. Can you fit a Michaelis-Menten characteristic through the data points? Of course, with only a single enzyme and the short intervals the stochastic variations between subsequent runs are enormous. You can improve the reproducibility by running longer steady state intervals or by increasing the number of proteins. Add another nine lines defining MMEnzyme proteins (each with a different index number) and rerun, re-plot, etc. The maximal total turnover should be about 750 product molecules per ten seconds and should decrease to half its value at about 350 substrate molecules in the pool.

Now comment out all but one MMEnzyme proteins and rerun the simulation with logging of the protein internal states:

```
$> ../vesimulus  10  1  10  MMEnzyme.ves   fish  dummyArg
```

The former terminal output is found in fish_pools.dat and the occupation states of the two binding sites in **fish_MMEnzyme_1_internals.dat**. Examine this file to see that actually first bs_S is occupied, then both bs_S and bs_P are on during the conversion. Finally, only bs_P = 1 indicates that the product is ready to be released. The relative timing of the events can be visualized nicely with the following command in **Gnuplot** (the "\" indicates that there should not be a line break — enter everything on one line).

```
> plot [0:2][0:1.4] "fish_MMEnzyme_1_internals.dat" u 1:2 w ste, \
                 "fish_MMEnzyme_1_internals.dat" u 1:($3+0.1) w ste
```

You see that the conversion starts immediately when the substrate is bound (which is the implemented behavior) and that the enzyme spends most of its time converting substrate into product. From these files one can now extract, e.g., the total time that the enzyme has a converted product attached or the average state that one or multiple enzymes are in (waiting, processing, releasing product, etc)

A Minimal Pathway

With the MMEnzyme we can start to build the first simple "pathways" where the product of one processing stage is the substrate for the next stage as shown in the figure below. In the example the substrate S is converted into the final product P via two intermediates I1 and I2. In the sketch the enzymes are hidden behind the arrows and the pools "abbreviated" by the metabolites.

$$\mathrm{S} \longrightarrow \mathrm{I1} \longrightarrow \mathrm{I2} \longrightarrow \mathrm{P}$$

To discern the enzymes of the three reactions we extend the protein type in the setup file by an additional tag separated from the protein type by an underscore, which is ignored by vesimulus during setup but respected when creating the protein labels. When using extended protein types the indices may repeat for the different groups. We thus need a setup file with three (groups of) enzymes and four pools. Again, the first substrate pool is assumed to be infinite. For simplicity all three enzymes have the same kinetic parameters initially. The substrate pool is empty initially, then set to an intermediate concentration after two seconds and reset to empty five seconds later. Have a look at the supplied setup file miniPath.ves.

Now run this simulation with a time step of 10 milliseconds and an output interval of one second. You see that the first intermediate I1 is produced rather fast, but that even after 50 seconds there only a very small amount of I2 or P because the uptake of I1 and I2 is so slow. Increase the association constants for the second and third stage such that within 50 seconds there is no I1 or I2 left (to within stochastic uncertainty). Also save the output into a file and plot the pool levels.

Bacterial Photosynthesis: A Single Flash

The following scenario reproduces an experiment in which the transmembrane voltage and the cytochrome c oxidation state in the photosynthetic apparatus of the purple bacterium *Rb. sphaeroides* were monitored in response to a single strong flash (Barz *et al.*, 1995). For this we implement a typical chromatophore vesicle with all its proteins in the dark adapted state. The flash is modeled by a fast change of the "concentration" in the light pool. Have a look at the supplied setup file **`Vesicle_a7.ves`**. It defines ten LHCs, which all get photons from the same pool "Light" but put the absorbed photons as excitons into a different pool "E1" to "E10", each. Each LHC models the combined absorption of a dimeric LH1 with six associated (=closely coupled) auxiliary LH2.

The dimeric core complexes of *Rb. sphaeroides* are then implemented by connecting two RCs to each of the exciton pools, i.e., we need 20 RCs. The last column of the RC section defines a pool for "virtual" charges to account for the charge transfer that occurs when the electron is translocated from the special pair bacteriochlorophylls on the periplasmic side of the RC (= at the inside of the vesicle) to the bound Q_b quinon eon the cytoplasmic side (=at the vesicle outside). The negative charge which is thus transferred across the membrane to the outside of the vesicle is here counted as the remaining positive charge placed into the vesicle (hence the name **`H+Protein`**). When the special pair is later reduced the corresponding charge is removed again from this virtual counter pool. Correspondingly, you find further down in the pool section that the **`H+Protein`** pool has a volume of 1 (without any units).

Next in the setup file come the dimeric cytochrome bc_1 complexes. These are already implemented as dimers with two binding sites for quinones, quinols, and cytochrome c_2, etc. The bc_1 pump the protons against the transmembrane potential ΔΦ, therefore some of the internal reactions are slowed down exponentially with increasing ΔΦ. To get the actual value of ΔΦ, the *bc*1 query the observable **DeltaPhi_ChemCap**, which returns the combined contributions from the proton gradient difference between inside and outside of the vesicle (= "chemical" part) and the "electrical" part from the charges in the vesicle. By referencing a different observable it is then easy to investigate how the *bc*1 would behave when only the chemical or only the electrical part are considered. These two forms for ΔΦ are called **DeltaPhi_Chem** and **DeltaPhi_Cap,** respectively.

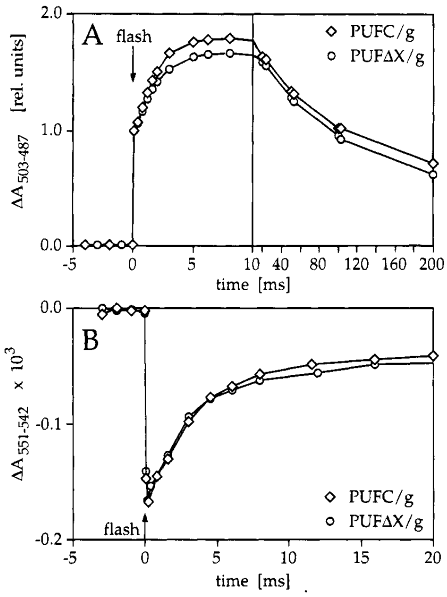
To complete the protein setup a single ATPase and a (pseudo)protein describing a number of protonateable residues (protonateable) are added before the relevant observables are defined. The most important one are the two forms of ΔΦ. While both chemical and electrical contribution affect the reactions in the proteins, only the electrical part could be measured via a spectral detuning of the the bacteriochlorophylls of the LHCs in the experiment. The other measured quantity was the oxidized fraction of the total cytochrome c content. The number of oxidized cytochrome c_2, which shuffle the electrons through the vesicle interior, can directly be read from the number in the relevant pool **c2ox**. The number of oxidized c_2 that are currently bound to the RCs is queried from the RCs via the observable **RC_c2ox** and the oxidation state of the c_1 domains of the bc1 dimers is obtained via the observable **bc1dimer2_c1ox**. These two observables loop over all proteins and call the corresponding methods of the RCs and *bc*1s, respectively, which return for each protein the current state.

To connect the proteins, a number of pools are required, which are defined in the following part of the setup file. Check the setup file for specific comments to each pool. Finally externally determined changes in the pool concentrations are listed. Here, for this single-flash experiment, the light intensity is switched on for 100 μs after an initial "thermalization" phase of 30 ms which makes sure that the simulation is really in a dark-adapted state.

Next we need to run the simulation. For such a fast flash scenario a time step of 1 μs and an output interval of 1 ms are required, whereas the total simulation time can be as short as 300 ms. The corresponding command is thus

```
   $> ../vesimulus  0.3  0.001  1  Vesicle_a7.ves
```

From the output you can see that at 0.03 seconds the Light is switched on shortly. Subsequently, a number of oxidized c_2 show up transiently first in the pool and then bound to the RCs. The oxidation "signal" is then seen for some more time in the *bc*1s before it decays back to its dark state level. From these three data columns the fraction of oxidized cytochrome *c* can be summed up and then compared to the experiment (see the following figure from the original article). Correspondingly, the transmembrane potential increases very fast during the first millisecond and then slower to reach its maximum with the next 10 ms before it decays again.



Repeat this simulation a few times to get a feeling for the stochastic variations of the observables. There are two things to notice. First, with the small numbers involved, the variations are very large and second, the simulation results look more like discrete changes than the continuous experimental traces. We thus have to run the simulation a few times and average over the simulation outputs before we can compare them to the experiments. Have a look at the supplied shell script runner_a7.sh. It uses some temporary files ("fish*") to store and combine the simulation results and returns the averaged response in the file fish_avg.txt. Start it with the setup file as argument:

```
$> ./runner_a7.sh  Vesicle_a7.ves
```

To compare the simulation results to the experiment the simulation data has to be shifted by 30 ms and rescaled. In **Gnuplot** this can directly be done as follows for ΔΦ and the overall cytochrome *c* oxidation state, respectively (the actual scaling factors may vary a bit even after averaging over 40 simulation runs):

```
> plot [-0.01:0.25]  "fish_avg.txt" u ($1-0.03):($2/85), "Barz95a7a_dPhi.txt"
> plot [-0.01:0.05]  "fish_avg.txt" u ($1-0.03):($3/20), "Barz95a7b_cox.txt"
```

You can see that ΔΦ is reproduced quite well during the fast transients of the first five to ten milliseconds and then decays slightly slower. The cytochrome c oxidation state, on the other hand, decays faster than in the experiment. One problem with the experiment was that the actual time course of the flash light is not documented. The agreement between simulation and experiment can now, e.g., be improved by letting the flash bulb cool down slower. As a guess modify the **changes:** section of the input file as follows and re-run the simulations.

```
change:   Light   0.03     1500
change:   Light   0.0301   1000
change:   Light   0.0302   700
change:   Light   0.0303   400
change:   Light   0.0304   250
change:   Light   0.0305   100
change:   Light   0.0306   0
```

Now the cytochrome *c* oxidation state does not decay that fast any more. Note that this light profile is only a guess (or a fit) to improve the agreement. Implementing the actual scoring of the simulation results vs. the experimental data is left as an exercise to the reader :-)

Identifying the Bottleneck Reaction of a Protein

When all kinetic parameters for a certain protein have been obtained either directly from literature or from an optimization against experimental data, each of these parameters can be scanned individually to find out which of them, e.g., limits the steady state throughput. In the following example we consider one dimeric core complex of the bacterial photosynthetic apparatus under saturating light intensity.

The RC takes up reduced cytochrome c_2, oxidized quinones, and protons from the outside of the vesicle and produces oxidized c_2 and quinol (QH2). To model steady state conditions, the three input pools are set to infinite. Then the turnover of the RC can be directly observed from the increase of the particle numbers in the output pools (this works because the unbinding reactions are insensitive to the product concentrations, i.e., product rebinding is ignored in the model. Such a setup is defined in the configuration file **`oneRC.ves`** with the optimized kinetic parameters for the LHC and the two RCs. A simulation of this setup for 100 seconds at a time step of 10 μs reveals that the two RCs can oxidize about 2500 cytochrome c_2 in 100 seconds, i.e. that one RC has a turnover of about 12.5 c_2 per second.

Now reduce for example the unbinding rate of the reduced QH2 by a factor of ten or hundred and re-run the simulation. This will result in turnovers of about 7.5 and 1.5 oxidized c_2 per second per RC. For a more thoroughly sampled characteristic we can again use a script which modifies the setup file and runs the simulation. Have a look at **`miniScan.sh`**. It uses another script **`confFile-variator.pl`** to create a modified copy (fish.ves) with—in this case—the tenth parameter of the RC replaced by a new value. The values of this sequence were chosen to be equidistant on a logarithmic scale. Run the script **`miniScan.sh`** and save the output:

```
$> ./miniScan.sh | tee RCscan.txt
```

Now plot the resulting data with logarithmic scales and verify that it follows a Michaelis-Menten characteristic (again). You should find that half of the maximal turnover is reached with k_{off}(QH2) ≈ 6 s^{-1} which is about one order of magnitude slower than the optimized value. This optimized value is thus just high enough to not be throughput limiting. Can you identify which reaction limits the turnover in the current parametrization?

Similar scans could now be performed for the other proteins under steady state conditions or with the flash setup from above. Then one finds that for different dynamic scenarios different parameters are important.

Connectivity Matters: Dimeric vs. Monomeric Core Complexes

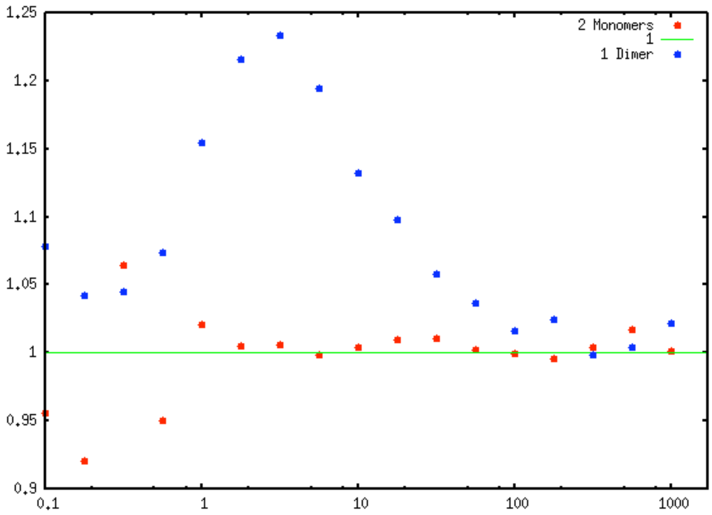
A similar scan as performed above for the kinetic parameters of the core complexes can also be performed for varying light intensities. Copy miniScan.sh and adapt the aruments to the confFile-Variator.pl script. Replace the "protein" label by the label of the light pool ("Light"), tell it to vary parameter 3, and scan light intensities of 0.1 to 1000 W/m^2 and save the output:

```
$> ./miniScan.2.sh | tee lightScan_dimer.txt
```

Now add another LHC and a second exciton pool E2 in the setup file oneRC.ves. Each LHC now has an absorption cross section of half of the initial value, i.e., of only 3.11 W^{-1} m^2 s^{-1} and each LHC is connected to only one RC as shown in the setup file **`twoMonomers.ves`**. Now perform the same scan of the turnover vs. the light intensity and compare the two scenarios.

When you now plot both characteristics in one plot you will see that the monomeric setup follows the familiar Michaelis-Menten kinetics while the dimeric setup has a higher turnover in the regime of intermediate light intensities of 1…10 W m^{-2}. When the fluctuations are too large increase the lengths of the individual simulations.

This advantage of the dimeric core complex vs. two monomers of up to 25% can be seen even better, when both simulation results are normalized against the Michaelis-Menten kinetics as shown in the following screen shot.



The different turnovers at intermediate light intensities are due to the dead times of the RC upon arrical of an exciton. When two RCs are connected to a (larger) LHC then chances are higher that two excitons which arrive shortly after the other will both be processed, whereas the second would be lost in the monomeric scenario. In the bacteria there is an additional coupling between adjacent core complexes which would allow for an even more efficient use of the captured photons at lower light intensities without the need for large antenna systems that are useless in high light conditions. To investigate this effect you can also play with setups where one large LHC is connected to four, eight, or even more RCs or where a connector protein allows for an exciton exchange between different exciton pools.

Note that this is a long-time steady state simulation where conventional wisdom would predict that stochastic effects can be ignored and a rate equation model will yield the same result. Indeed, a rate equation model predicts the Michaelis-Menten characteristic of the monomers, but cannot capture the behavior of the more efficient dimers. Similar differences between the here presented molecular-stochastic modeling and a classical rate equation approach will be visible in, for example, signaling with its typically few receptors.

Contact Information And Further Reading

The most up-to-date version of vesimulus and the documentation, lists of changes and fixed bugs, and (hopefully soon) models for more types of proteins can be found on our server at

http://service.bioinformatik.uni-saarland.de/vesimulus

If you have any questions, suggestions, etc, or if you are willing to share protein models set up by you with others please contact the main author by email:

tihamer.geyer@bioinformatik.uni-saarland.de

The reconstruction of the bacterial chromatophore vesicles is documented in the following two publications. This steady state reconstruction is compiled from a wide variety of published information..

T. Geyer and V. Helms, "A spatial model of the chromatophore vesicles of *Rhodobacter sphaeroides* and the position of the cytochrome bc_1 complex", *Biophys. J.* **91** (2006) 921-6

T. Geyer and V. Helms, "Reconstruction of a kinetic model of the chromatophore vesicles from *Rhodobacter sphaeroides*", *Biophys. J.* **91** (2006) 927-37

The molecular-stochastic pools-and-proteins model was introduced first in:

T. Geyer, F. Lauck, and V. Helms, "Molecular stochastic simulations of chromatophore vesicles from *Rb. sphaeroides*", *J. Biotech.* **129** (2007) 212-28

How the model of the bacterial photosynthetic apparatus was then parameterized is documented in:

T. Geyer, X. Mol, S. Blaß, and V. Helms, "Bridging the Gap: Linking Molecular Simulations and Systemic Descriptions of Cellular Compartments", *PLoS ONE* **5** (2010) e14070

An online version of the chromatophore simulations which has successfully been used in classrooms to familiarize students with stochastic effects can be accessed at the following URL. There also documentation and tutorials can be found.

http://service.bioinformatik.uni-saarland.de/vesiweb