ecell3

Release 3.2.3pre2

December 08, 2015

Contents

1 Legal **3**
1.1 Feedback . . . 4

2 Introduction **5**
2.1 What is ECELL . . . 5
2.2 Organization of This Manual . . . 5

3 Getting Started **7**
3.1 Preparing Simulation . . . 7
3.2 Starting APP . . . 8

4 Modeling with ECELL **9**
4.1 Objects In The Model . . . 9
4.2 E-Cell Model (EM) File Basics . . . 13
4.3 Structure Of The Model . . . 16
4.4 Modeling Schemes . . . 21
4.5 Modeling Convensions . . . 30

5 Modeling Tutorial **31**
5.1 Running the model . . . 31
5.2 Using Gillespie algorithm . . . 31
5.3 Using Deterministic Differential Equations . . . 34
5.4 Making the Model Switchable Between Algorithms . . . 35
5.5 A Simple Deterministic / Stochastic Composite Simulation . . . 36
5.6 Custom equations . . . 38
5.7 Other Modeling Schemes . . . 39

6 Scripting A Simulation Session **41**
6.1 Session Scripting . . . 41
6.2 Running ECELL Session Script . . . 41
6.3 Writing ECELL Session Script . . . 42
6.4 Handling Data Files . . . 47
6.5 Manipulating Model Files . . . 49
6.6 Other Methods . . . 49
6.7 Advanced Topics . . . 49
6.8 ECELL Python Library API . . . 51

7 Creating New Object Classes **53**
7.1 About Dynamic Modules . . . 53

7.2 Defining a new class . . . 53
7.3 PropertySlot . . . 57
7.4 Defining a new Process class . . . 62
7.5 Defining a new Stepper class . . . 63
7.6 Defining a new Variable class . . . 63
7.7 Defining a new System class . . . 63

8 Standard Dynamic Module Library **65**
8.1 Steppers . . . 65
8.2 Process classes . . . 67
8.3 Variable classes . . . 68

9 Simulation Mechanism of E-Cell **69**

10 Empy Module Manual **71**

11 index **99**

Koichi Takahashi
ktakahashi@riken.jp
RIKEN Quantitative Biology Center, Furuedai, Suita, Osaka 565-0874, Japan

CHAPTER 1

Legal

1.1 Feedback

To report a bug or make a suggestion regarding the E-Cell Simulation Environment application or this manual, send an email to <shafi at e-cell.org>

CHAPTER 2

Introduction

2.1 What is ECELL

The APP is a software environment for simulation of various cellular phenomena.

2.2 Organization of This Manual

This manual has the following chapters.

1. Introduction – you are here
2. Getting Started
3. Modeling with E-Cell
4. Scripting E-Cell
5. Creating New Object Classes
6. Standard Dynamic Module Library

If you are just to start using APP Version 3 as a *user*, read chapters 2, 3, and try it in your project. Read the rest parts if you feel it is necessary. Especially you may want to browse chapter 6 to know what classes are available to create your model. If you cannot find what you need in that chapter, you may want to develop your own module by reading chapter 5. Reading chapter 4 will help you to automate your simulation runs. If you are a *frontend module developer* of APP written in PYTHON, read mainly chapter 4. You may also want to read chapters 2 and 3 if you are not already familiar with how the system is organized and used. If you are a C++ *Dynamic Module developer*, (for example, to create a new algorithm module) you need to read at least the chapter 5. Read chapters 2 and 3 if you are not already familiar with ECELL. Also, you may want to read the chapter 6 to know how existing classes are designed.

CHAPTER 3

Getting Started

By reading this chapter, you can get information about the following items.

What types of files are needed to run the simulator. How to prepare the files needed to run the simulator. How to run the simulation with APP.

3.1 Preparing Simulation

To start the simulation, you need to have these types of files:

- A model file in EML format.
- (optionally) shared object files (.so in Linux operating system), if you are using special classes of object in the model file which is not provided by the system by default.
- (optionally) a script file (ECELL Session Script, or ESS) to automate the simulation session.

3.1.1 Converting EM to EML

Simulation models for ECELL is often written in EM format. To convert EM (.em) files to EML (.eml) files, type the following command.

`ecell3-em2eml` filename.em

You can obtain the full description of the command line options giving -h option to `ecell3-em2eml`.

```
ecell3-eml2em -- convert eml to em

Usage:
        ecell3-eml2em [-h] [-f] [-o outfile] infile


Options:
        -h or --help    :  Print this message.
        -f or --force   :  Force overwrite even if outfile already exists.
        -o or --outfile=:  Specify output file name.  '-' means stdout.
```

3.1.2 Compiling C++ Dynamic Modules

You might have some Dynamic Modules (or DM in short) specifically made for the simulation model in the form of C++ source code. If that is the case, those files have to be compiled and linked to form shared module files (usually

suffixed by `.so` on Unix-like platforms, `.dylib` on Mac OS X or `.dll` on Windows) before running the simulation. You will also need to set `ECELL3_DM_PATH` environment variable to the appropriate value to use the DMs (discussed below).

To compile and like DMs, `ecell3-dmc` command is provided for convenience.

The arguments given before the file name (‘‘[command options]‘‘ are interpreted as options to the `ecell3-dmc` command itself.

The arguments after the file name are passed to a backend compiler (such as g++) as-is. The backend compiler used is the same as that used to build the system itself.

To inspect what the command actually does inside, enable verbose mode by specifying `-v` option.

To get a full list of available `ecell3-dmc` options, invoke the command with `-h` option, and without the input file. Here is the help message shown by issuing `ecedll3-dmc -h`. Compile dynamic modules for E-Cell Simulation Environment Versin 3. Usage: ecell3-dmc [ecell3-dmc options] sourcefile [compiler options] ecell3-dmc -h|–help ecell3-dmc options: –no-stdinclude Don’t set standard include file path. –no-stdlibdir Don’t set standard include file path. –ldflags=[ldflags] Specify options to the linker. –cxxflags=[cxxflags] Override the default compiler options. –dmcompile=[path] Specify dmcompile path. -v or –verbose Be verbose. -h or –help Print this message.

3.2 Starting APP

You can start APP either in scripting mode and GUI mode.

3.2.1 GUI mode

To start APP in GUI mode, type the following command.

```
&
```

This will invoke an instance of the simulator with Osogo Session Manager attached as a GUI frontend.

3.2.2 Scripting mode

To start APP in scripting mode, type the following command:

where filename.ess is the name of the Python script file you want to execute.

If filename.ess is omitted, the interpreter starts up in interactive mode.

See chapter 5 for the scripting feature.

3.2.3 DM search path and `ECELL3_DM_PATH` environment variable

If your model makes use of non-standard DMs that you had to build using `ecell3-dmc`, then you need to specify the directory where the DMs are placed in `ECELL3_DM_PATH` environment variable. `ECELL3_DM_PATH` can have multiple directory names separated by either `:` (colon) on Unix-like platform or `;` (semicolon) on Windows.

The following is an example of setting `ECELL3_DM_PATH` before launching `ecell3-session-monitor`:

Note that up to E-Cell SE 3.1.105, the current working directory was implicitly treated as if it was included in `ECELL3_DM_PATH`. This quirk is removed since 3.1.106.

CHAPTER 4

Modeling with ECELL

By reading this chapter, you can get information about:

How an ECELL's simulation model is organized. How to create a simulation model. How to write a model file in EM format.

4.1 Objects In The Model

ECELL's simulation model is fully object-oriented. That is, the simulation model is actually a set of *objects* connected each other. The objects have *properties*, which determine characteristics of the objects (such as a reaction rate constant if the object represent a chemical reaction) and relationships between the objects.

4.1.1 Types Of The Objects

A simulation model of APP consists of the following types of objects.

- Usually more than one ENTITY objects
- One or more STEPPER object(s)

ENTITY objects define the structure of the simulation model and represented phenomena (such as chemical reactions) in the model. STEPPER objects implement specific simulation algorithms.

Entity objects

The ENTITY class has three subclasses:

- VARIABLE

  This class of objects represent state variables. A VARIABLE object holds a scalar real-number value. A set of values of all VARIABLE objects in a simulation model defines the state of the model at a certain point in time.

- PROCESS

  This class of objects represent phenomena in the simulation model that result in changes in the values of one or more VARIABLE objects. The way of change of the VARIABLE values can be either discrete or continuous.

- SYSTEM

  This class of objects define overall structure of the model. A SYSTEM object can contain sets of these three types of ENTITY, VARIABLE, PROCESS, and SYSTEM objects. A SYSTEM can contain other SYSTEMs, and can form a tree-like structure.

Stepper objects

A model must have one or more STEPPER object(s). Each PROCESS and SYSTEM object must be connected with a STEPPER object in the same model. In other words, STEPPER objects in the model have non-overlapping sets of PROCESS and SYSTEM objects.

STEPPER is a class which implement a specific simulation algorithm. If the model has more than one STEPPER objects, the system conducts a multi-stepper simulation. In addition to the lists of PROCESS and SYSTEM objects, a STEPPER has a list of VARIABLE objects that can be read or written by its PROCESS objects. It also has a time step interval as a positive real-number. The system schedules STEPPER objects according to the step intervals, and updates the current time.

When called by the system, a STEPPER object integrates values of related VARIABLE objects to the current time (if the model has a differential component), calls zero, one or more PROCESS objects connected with the STEPPER in an order determined by its implementation of the algorithm, and determines the next time step interval. See the following chapters for details of the simulation procedure.

4.1.2 Object Identifiers

APP uses several types of identifier strings to specify the objects, such as the ENTITY and STEPPER objects, in a simulation model.

ID (ENTITYID and STEPPERID)

Every ENTITY and STEPPER object has an *ID*. ID is a character string of arbitrary length starting from an alphabet or '_' with succeeding alphabet, '_', and numeric characters. APP treats IDs in a case-sensitive way.

If the ID is used to indicate a STEPPER object, it is called a STEPPERID. The ID points to an ENTITY object is refered to as ENTITYID, or just *ID*.

(need EBNF here)

Examples: `_P3`, `ATP`, `GlucoKinase`

SystemPath;

The SYSTEMPATH identifies a SYSTEM from the tree-like hierarchy of SYSTEM objects in a simulation model. It has a form of ENTITYID strings joined by a character '/' (slash). As a special case, the SYSTEMPATH of the root system is `/`. For instance, if there is a SYSTEM `A`, and `A` has a subsystem `B`, a SYSTEMPATH `/A/B` specifies the SYSTEM object `B`. It has three parts: (1) the root system (`/`), (2) the SYSTEM `A` directly under the root system, and (3) the SYSTEM `B` just under `A`.

A SYSTEMPATH can be relative. The relative SYSTEMPATH does not point at a SYSTEM object unless the current SYSTEM is given. A SYSTEMPATH is relative if (1) it does not start with the leading `/` (the root system), or (2) it contains '`.`' (the current system) or '`..`' (the super-system).

Examples: `/A/B`, `../A`, `.`, `/CELL/ER1/../CYTOSOL`

FullID

A FULLID (FULLy qualified IDentifier) identifies a unique ENTITY object in a simulation model. A FULLID comprises three parts, (1) a ENTITYTYPE, (2) a SYSTEMPATH, and (3) an ENTITYID, joined by a character '`:`' (colon).

```
::
```

The ENTITYTYPE is one of the following class names:

- SYSTEM
- PROCESS
- VARIABLE

For example, the following FULLID points to a PROCESS object of which ENTITYID is '`P`', in the SYSTEM '`CELL`' immediately under the root system (`/`). Process:/CELL:P

FullPN

FULLPN (FULLy qualified Property Name) specifies a unique *property* (see the next section) of an ENTITY object in the simulation model. It has a form of a FULLID and the name of the property joined by a character '`:`' (colon).

```
:
```

or,

```
:::
```

The following FULLPN points to 'Value' property of the VARIABLE object `Variable:/CELL:S`. Variable:/CELL:S:Value

4.1.3 Object Properties

ENTITY and STEPPER objects have *properties*. A property is an attribute of a certain object associated with a name. Its value can be get from and set to the object.

Types of object properties

A value of a property has a *type*, which is one of the followings.

- REAL number

  (ex. `3.33e+10`, `1.0`)

- INTEGER number

  (ex. `3`, `100`)

- STRINGTYPE

  STRINGTYPE has two forms: quoted and not quoted. A quoted STRINGTYPE can contain any ASCII characters except the quotation characters (' or "). Quotations can be omitted if the string has a form of a valid object identifier (ENTITYID, STEPPERID, SYSTEMPATH, FULLID, or FULLPN).

  If the STRINGTYPE is triple-quoted (by `'''` or `"""`), it can contain new-line characters. (The current version still has some problems processing this.)

  (ex. `_C10_A`, `Process:/A/B:P1`, ''"It can include spaces if double-quoted."'',

  '''single-quote is available too, if you want to use "double-quotes" inside.''')

- List

  The list can contain REAL, INTEGER, and STRINGTYPE values. This list can also contain other lists, that is, the list can be nested. A list must be surrounded by brackets (`[` and `]`), and the elements must be separated by space characters. In some cases outermost brackets are omitted (such as in EM files, see below).

  (ex. ‘‘[A 10 [1.0 “a string” 1e+10]]‘‘)

Dynamic type adaptation of property values

The system automatically convert the type of the property value if it is different from what the object in the simulator (such as PROCESS and VARIABLE) expects to get. That is, the system does not necessary raise an error if the type of the given value differs from the type the backend object accepts. The system tries to convert the type of the value given in the model file to the requested type by the objects in the simulator. The conversion is done by the objects in the simulator, when it gets a property value. See also the following sections.

The conversion is done in the following manner.

- From a numeric value (REAL or INTEGER)
  - To a STRINGTYPE

    The number is simply converted to a character string. For example, a number 12.3 is converted to a STRINGTYPE `'12.3'`.
  - To a list

    A numeric value can be converted to a length-1 list which has that number as the first item. For example, 12.3 is equivalent to ‘[12.3]’.
- From a STRINGTYPE
  - To a numeric value (REAL or INTEGER)

    The initial portion of the STRINGTYPE is converted to a numeric value. The number can be represented either in a decimal form or a hexadecimal form. Leading white space characters are ignored. ‘INF’ and ‘NAN’ (case-insensitive) are converted to an infinity and a NaN (not-a-number), respectively. If the initial portion of the STRINGTYPE cannot be converted to a numeric value, it is interpreted as a zero (0.0 or 0). This conversion procedure is equivalent to C functions `strtol` and `strtod`, according to the destined type.
  - To a list

    A STRINGTYPE can be converted to a length-1 list which has that STRINGTYPE as the first item. For example, ‘string’ is equivalent to ‘[‘string’]’.
- From a list
  - To a numeric or a STRINGTYPE value

    It simply takes the first item of the list. If necessary the taken value is further converted to the destined types.

    Note

    When converting from a REAL number to an INTEGER, or from a STRINGTYPE to a numeric value, overflow and underflow can occur during the conversion. In this case an exception (TYPE??) is raised when the backend object attempts the conversion.

4.2 E-Cell Model (EM) File Basics

Now you know the ECELL's simulation model consists of what types of objects, and the objects have their properties. The next thing to understand is how the simulation model is organized: the structure of the model. But wait, learn the syntax of the ECELL model (EM) file before proceeding to the next section would help you very much to understand the details of the structure of the model, because most of the example codes are in EM.

4.2.1 What Is EM?

In APP, the standard file format of model description and exchange is XML-based EML (E-Cell Model description Language). Although EML is an ideal means of integrating E-Cell with other software components such as GUI model editors and databases, it is very tedious for human users to write and edit by hand.

E-Cell Model (EM) is a file format with a programming language-like syntax and a powerful embedded EMPY preprocessor, which is designed to be productive and intuitive especially when handled by text editors and other text processing programs. Semantics of EM and EML files are almost completely equivalent to each other, and going between these two formats is meant to be possible with no loss of information (some exceptions are comments and directions to the preprocessor in EM). The file suffix of EM files is ".em".

Why and when use EM?

Although E-Cell Modeling Environment (which is under development) will provide means of more sophisticated, scalable and intelligent model construction on the basis of EML, learning syntax and semantics of EM may help you get the idea of how object model inside ECELL is organized and how it is driven to conduct simulations. Furthermore, owing to the nature of the plain programming language-like syntax, EM can be used as a simple and intuitive tool to communicate with other ECELL users. In fact, this manual uses EM to illustrate how the model is constructed in ECELL

EM files can be viewed as EML generator scripts.

4.2.2 EM At A Glance

Before getting into the details of EM syntax, let's have a look at a tiny example. It's very simple, but you do not need to understand everything for the moment.

```
Stepper ODEStepper( ODE_1 )
{
        # no property
}

System System( / )
{
        StepperID       ODE_1;

        Variable Variable( SIZE )
        {
                Value   1e-18;
        }

        Variable Variable( S )
        {
                Value   10000;
        }
```

```
        Variable Variable( P )
        {
                Value   0;
        }

        Process MassActionFluxProcess( E )
        {
                Name  "A mass action from S to P."
                k     1.0;

                VariableReferenceList [ S0 :.:S -1 ]
                                      [ P0 :.:P 1 ];
        }

}
```

This example is a model of a mass-action differential equation. In this example, the model has a STEPPER `ODE_1` of class ODEStepper, which is a generic ordinary differential equation solver. The model also has the root system (`/`). The root sytem has the StepperID property, and four ENTITY objects, VARIABLEs `SIZE`, `S` and `P`, and the PROCESS `E`. `SIZE` is a special name of the VARIABLE, that determines the size of the compartment. If the compartment is three-dimensional, it means the volume of the compartment in [L] (liter). That value is used to calculate concentrations of other VARIABLEs. These ENTITY objects have their property values of several different types. For example, `StepperID` of the root system is the string without quotes (`ODE_1`). The initial value given to Value property of the VARIABLE `S` is an integer number `10000` (and this is automatically converted to a real number `10000.0` when the VARIABLE gets it because the type of the Value property is REAL). Name property of the PROCESS `E` is the quoted string `"A mass action from S to P"`, and 'k' of it is the real number `1.0`. VariableReferenceList property of `E` is the list of two lists, which contain strings (such as `S0`), and numbers (such as `-1`). The list contain relative FULLIDs (such as `:.:S`) without quotes.

4.2.3 General Syntax Of EM

Basically an EM is (and thus an EML is) a list of just one type of directives: *object instantiation*. As we have seen, ECELL's simulation models have only two types of 'objects'; STEPPER and ENTITY. After creating an object, property values of the object must be set. Therefore the object instantiation has two steps: (1) creating the object and (2) setting properties.

General form of object instantiation statements

The following is the general form of definition (instantiation) of an object in EM:

```
TYPE CLASSNAME( ID )
"""INFO ()"""
{
        PROPERTY_NAME_1 PROPERTY_VALUE_1;
        PROPERTY_NAME_2 PROPERTY_VALUE_2;
        ...
        PROPERTY_NAME_n PROPERTY_VALUE_n;
}
```

where:

- TYPE

  The type of the object, which is one of the followings:

  - STEPPER

  - VARIABLE
  - PROCESS
  - SYSTEM

- ID

  This is a *StepperID* if the object type is STEPPER. If it is SYSTEM, put a SYSTEMPATH here. Fill in an ENTITYID if it is a VARIABLE or a PROCESS.

- CLASSNAME

  The classname of this object. This class must be a subclass of the baseclass defined by *TYPE*. For example, if the *TYPE* is PROCESS, *CLASSNAME* must be a subclass of PROCESS, such as MassActionFluxProcess.

- INFO

  An annotation for this object. This field is optional, and is not used in the simulation. A quoted single-line ("string") or a multi-line string (""" multi-line string """) can be put here.

- PROPERTY

  An object definition has zero or more properties.

  The property starts with an unquoted property name string, followed by a property value, and ends with a semi-colon (`;`). For example, if the property name is Concentration and the value is `10.0`, it may look like: Concentration 10.0;

  REAL, INTEGER, STRINGTYPE, and List are allowed as property value types (See the Object Properties section above).

  If the value is a List, outermost brackets are omitted. For example, to put a list

```
[ 10 "string" [ LIST ] ]
```

  into a property slot `Foo`, write a line in the object definition like this: Foo 10 "string" [LIST];

  Note

  All property values are lists, even if it is a scalar REAL number. Remember a number '1.0' is interconvertible with a length-1 list '[1.0]'. Therefore the system can correctly interpret property values without the brackets.

  In other words, if the property value is bracketed, for example, the following property value

```
Foo [ 10 [ LIST ] ];
```

  is interpreted by the system as a length-1 List

```
[ [ 10 [ LIST ] ] ]
```

  of which the first item is a list

```
[ 10 [ LIST ] ]
```

  This may or may not be what you intend to have.

4.2.4 Macros And Preprocessing

Before converting to EML, `ecell3-em2eml` command invokes the EMPY program to preprocess the given EM file.

By using EMPY, you can embed any PYTHON expressions and statements after '@' in an EM file. Put a PYTHON expression inside '@(python expression)', and the macro will be replated with an evaluation of the expression. If the

expression is very simple, '()' can be ommited. Use '@{ pytyon statements }' to embed PYTHON statements. For example, the following code:

```
@(AA='10')
@AA
```

is expanded to:

```
10
```

Of course the statement can be multi-line. This code

```
@{
  def f( str ):
      return str + ' is true.'
}

@f( 'Video Games Boost Visual Skills' )
```

is expanded to

```
Video Games Boost Visual Skills is true.
```

EMPY can also be used to include other files. The following line is replaced with the content of the file `foo.em` immediately before the EM file is converted to an EML:

```
@include( 'foo.em' )
```

Use `-E` option of `ecell3-em2eml` command to see what happens in the preprocessing. With this option, it outputs the result of the preprocessing to standard output and stops without creating an EML file.

It has many more nice features. See the appendix A for the full description of the EMPY program.

4.2.5 Comments

The comment character is a sharp '#'. If a line contains a '#' outside a quoted-string, anything after the character is considered a comment, and not processed by the `ecell3-em2eml` command.

This is processed differently from the EMPY comments (@#). This comment character is processed by the EMPY as a usual character, and does not have an effect on the preprocessor. That is, the part of the line after '#' is not ignored by EMPY preprocessor. To comment out an EMPY macro, the EMPY comment (@#) must be used.

4.3 Structure Of The Model

4.3.1 Top Level Elements

Usually an EM has one or more STEPPER and one or more SYSTEM statements. These statements are top-level elements of the file. General structure of an EM file may look like this:

```
STEPPER_0
STEPPER_1
...
STEPPER_n

SYSTEM_0 # the root system ( '/' )
SYSTEM_1
```

```
...
SYSTEM_m
```

`STEPPER_?` is a STEPPER statement and `SYSTEM_?` is a SYSTEM statement.

4.3.2 Systems

The root system

The model must have a SYSTEM with a SYSTEMPATH '/'. This SYSTEM is called the *root system* of the model.

```
System System( / )
{
    # ...
}
```

The class of the root system is always System, no matter what class you specify. This is because the simulator creates the root sytem when it starts up, before loading the model file. That is, the statement does not actually create the root system object when loading the EML file, but just set its property values. Consequently the class name specified in the EML is ignored. The model file must always have this root system statement, even if you have no property to set.

Constructing the system tree

If the model has more than one SYSTEM objects, it must form a tree which starts from the root system (/). For example, the following is *not* a valid EM.

```
System System( / )
{
}

System System( /CELL0/MITOCHONDRION0 )
{
}
```

This is invalid because these two SYSTEM objects, `/` and `/CELL0/MITOCHONDRION0` are not connected to each other, nor form a single tree. Adding another SYSTEM, `/CELL0`, makes it valid.

```
System System( / )
{
}

System System( /CELL0 )
{
}

System System( /CELL0/MITOCHONDRION0 )
{
}
```

Of course a SYSTEM can have arbitrary number of sub-systems.

```
System System( / )
{
}

System System( /CELL1 ) {}
System System( /CELL2 ) {}
```

```
System System( /CELL3 ) {}
# ...

**Note**

In future versions, the system will support composing a model from
multiple model files (EMs or EMLs). This is not the same as the EM's
file inclusion by EMPY preprocessor.
```

Sizes of the Systems

If you want to define the size of a SYSTEM, create a VARIABLE with an ID 'SIZE'. If the SYSTEM models a three-dimensional compartment, the SIZE here means the volume of that compartment. The unit of the volume is [L] (liter). In the next example, size of the root system is 1e-18.

```
System System( / )
{
    Variable Variable( SIZE )    # the size (volume) of this compartment
    {
        Value   1e-18;
    }
}
```

If a System has no 'SIZE' VARIABLE, then it shares the SIZE VARIABLE with its supersystem. The root system always has its SIZE VARIABLE. If it is not given by the model file, then the simulator automatically creates it with the default value 1.0. The following example has four SYSTEM objects, and two of them (/ and /COMPARTMENT) have their own SIZE variables. Remaining two (/SUBSYSTEM and its subsystem /SUBSYSTEM/SUBSUBSYSTEM) share the SIZE VARIABLE with the root system.

```
System System( / )                       # SIZE == 1.0 (default)
{
    # no SIZE
}

System System( /COMPARTMENT )            # SIZE == 2.0e-15
{
    Variable Variable( SIZE )
    {
        Value 2.0e-15
    }
}

System System( /SUBSYSTEM )              # SIZE == SIZE of the root sytem
{
    # no SIZE
}

System System( /SUBSYSTEM/SUBSUBSYSTEM ) # SIZE == SIZE of the root system
{
    # no SIZE
}

**Note**

Behavior of the system when zero or negative number is set to SIZE
is undefined.
```

```
**Note**

Currently, the unit of the SIZE is (10 cm)^\ *d*, where d is
dimension of the SYSTEM. If d is 3, it is (10 cm)^3 == liter. This
specification is still under discussion, and is subject to change in
future versions.
```

4.3.3 Variables And Processes

A SYSTEM statement has zero, one or more VARIABLE and PROCESS statements in addition to its properties.

```
System System( / )
{
    # ... properties of this System itself comes here..

    Variable Variable( V0 ) {}
    Variable Variable( V1 ) {}
    # ...
    Variable Variable( Vn ) {}

    Process SomeProcess( P0 )  {}
    Process SomeProcess( P1 )  {}
    # ...
    Process OtherProcess( Pm ) {}
}
```

Do not put a SYSTEM statement inside SYSTEM.

4.3.4 Connecting Steppers With Entity Objects

Any PROCESS and VARIABLE object in the model must be connected with a STEPPER by setting its StepperID property. If the StepperID of a PROCESS is omitted, it defaults to that of its supersystm (the SYSTEM the PROCESS belongs to). StepperID of SYSTEM cannot be omitted.

In the following example, the root sytem is connected to the STEPPER `STEPPER0`, and the PROCESS `P0` and `P1` belong to STEPPERs `STEPPER0` and `STEPPER1`, respectively.

```
Stepper SomeClassOfStepper( STEPPER0 )    {}
Stepper AnotherClassOfStepper( STEPPER1 ) {}

System System( / )  # connected to STEPPER0
{
    StepperID     STEPPER0;

    Process AProcess( P0 )     # connected to STEPPER0
    {
        # No StepperID specified.
    }

    Process AProcess( P1 )     # connected to STEPPER1
    {
        StepperID     STEPPER1;
    }
}
```

Connections between STEPPERs and VARIABLEs are automatically determined by the system, and cannot be specified manually. See the next section.

4.3.5 Connecting Variable Objects With Processes

A PROCESS object changes values of VARIABLE object(s) according to a certain procedure, such as the law of mass action. What VARIABLE objects the PROCESS works on cannot be determined when it is programmed, but it must be specified by the modeler when the PROCESS takes part in the simulation. VariableReferenceList property of the PROCESS relates some VARIABLE objects with the PROCESS.

VariableReferenceList is a list of *VARIABLEREFERENCEs*. A VARIABLEREFERENCE, in turn, is usually a list of the following four elements:

```
[      ]
```

The last two fields can be omitted:

```
[     ]
```

or,

```
[    ]
```

These elements have the following meanings.

1. Reference name

   This field gives a local name inside the PROCESS to this VARIABLEREFERENCE. Some PROCESS classes use this name to identify particular instances of VARIABLEREFERENCE.

   Currently, this reference name must be set for all VARIABLEREFERENCEs, even if the PROCESS does not use the name at all.

   Lexical rule for this field is the same as the ENTITYID; leading alphabet or '_' with trailing alphabet, '_', and numeric characters.

2. FULLID

   This FULLID specifies the VARIABLE that this VARIABLEREFERENCE points to.

   The SYSTEMPATH of this FULLID can be relative. Also, ENTITYTYPE can be omitted. That is, writing like this is allowed:

   ```
   :.:S0
   ```

   instead of

   ```
   Variable:/CELL:S0
   ```

   , if the PROCESS exists in the SYSTEM `/CELL`.

3. Coefficient (*optional*)

   This coefficient is an integer value that defines weight of the connection between the PROCESS and the VARIABLE that this VARIABLEREFERENCE points to.

   If this value is a non-zero integer, then this VARIABLEREFERENCE is said to be a *mutator VARIABLEREFERENCE*, and the PROCESS can change the value of the VARIABLE. If the value is zero, this VARIABLEREFERENCE is not a mutator, and the PROCESS should not change the value of the VARIABLE.

   If the PROCESS represents a chemical reaction, this value is usually interpreted by the PROCESS as a stoichiometric constant. For example, if the coefficient is -1, the value of the VARIABLE is decreased by 1 in a single occurence of the forward reaction.

   If omitted, *this field defaults to zero*.

4. *isAccessor* flag (*optional*)

   This is a binary flag; set either 1 (true) or 0 (false). If this *isAccessor* flag is false, it indicates that the behavior of PROCESS is not affected by the VARIABLE that this VARIABLEREFERENCE points to. That is, the PROCESS never reads the value of the VARIABLE. The PROCESS may or may not change the VARIABLE regardless of the value of this field.

   Some PROCESS objects automatically sets this information, if it knows it never changes the value of the VARIABLE of this VARIABLEREFERENCE. Care should be taken when you set this flag manually, because many PROCESS classes do not check this flag when actually read the value of the VARIABLE.

   The default is 1 (true). This field is often omitted.

   Note

   In multi-stepper simulations, this information sometimes helps the system to run efficiently. If the system knows, for example, all PROCESS objects in the STEPPER `A` do not change any VARIABLE connected to the other STEPPER `B`, it can give `B` more chance to have larger stepsizes, rather than always checking whether STEPPER `A` changed some of the VARIABLE objects. This flag is mainly used when there are more than one STEPPERs.

Consider a reaction PROCESS in the root system, `R`, consumes the VARIABLE `S` and produces the VARIABLE `P`, taking `E` as the enzyme. This class of PROCESS requires to give the enzyme as a VARIABLEREFERENCE of name `ENZYME`. All the VARIABLE objects are in the root system. In EM, VariableReferenceList of this PROCESS may appear like this:

```
System System( / )
{
    # ...
    Variable Variable( S ) {}
    Variable Variable( P ) {}
    Variable Variable( E ) {}

    Process SomeReactionProcess( R )
    {
        # ...
        VariableReferenceList [ S0     :.:S -1 ]
                              [ P0     :.:P  1 ]
                              [ ENZYME :.:E  0 ];

    }
}
```

4.4 Modeling Schemes

ECELL is a multi-algorithm simulator. It can run any kind of simulation algorithms, both discrete and continuous, and these simulation algorithms can be used in any combinations. This section exlains how you can find appropriate set of object classes for your modeling and simulation projects. This section does not give a complete list of available object classes nor detailed usage of those classes. Read the chapter “Standard Dynamic Module Library” for more info.

4.4.1 Discrete Or Continuous ?

ECELL can model both discrete and continuous processes, and these can be mixed in simulation. The system models discrete and continuous systems by discriminating two different types of PROCESS and STEPPER objects: discrete PROCESS / STEPPER and continuous PROCESS / STEPPER.

Note

VARIABLE and SYSTEM do not have special discrete and continuous classes. The base VARIABLE class supports both discrete and continous operations, because it can be connected to any types of PROCESS and STEPPER objects. SYSTEM objects do not do any computation that needs to discriminate discrete and continuos.

Discrete classes

A PROCESS object that models discrete changes of one or more VARIABLE objects is called a *discrete PROCESS*, and it must be used in conjunction with a *discrete STEPPER*. A discrete PROCESS directly changes the *values* of related VARIABLE objects when its STEPPER requests to do so.

There are two types of discrete PROCESS / STEPPER classes: discrete and discrete event.

- Discrete

  A discrete PROCESS changes values of connected VARIABLE objects (i.e. appear in its VariableReferenceList property) discretely. In the current version, there is no special class named DiscreteProcess, because the base PROCESS class is already a discrete PROCESS by default. The manner of the change of VARIABLE values is determined from values of its accessor VARIABLEREFERENCEs, its property values, and sometimes the current time of the STEPPER. Unlike discrete event PROCESS, which is explained in the next item, it does not necessary specify when the discrete changes of VARIABLE values occur. Instead, it is unilaterally determined and fired by a discrete STEPPER.

  A STEPPER that requires all PROCESS objects connected is discrete PROCESS objects is call a discrete STEPPER. The current version has no special class DiscreteStepper, because the base STEPPER class is already discrete.

- Discrete event

  Discrete event is a special case of discreteness. The system provides DiscreteEventStepper and DiscreteEventProcess classes for discrete-event modeling. In addition to the ordinary firing method (fire() method) of the base PROCESS class, the DiscreteEventProcess defines a method to calculate *when* is the next occurrence of the event (the discrete change of VARIABLE values that this discrete event PROCESS models) from values of its accessor VARIABLEREFERENCEs, its property values, and the current time of the STEPPER. DiscreteEventStepper uses information given by this method to determine when each of discrete event PROCESS should be fired. DiscreteEventStepper is instantiatable. See the chapter Standard Dynamic Module Library for more detailed description of how DiscreteEventStepper works.

Continuous classes

On the other hand, a PROCESS that calculates continuous changes of VARIABLE objects is called a *continuous PROCESS*, and is used in combination with a *continuous STEPPER*. Continuous PROCESS objects simulate the phenomena that represents by setting *velocities* of connected VARIABLE objects, rather than directly changing their values in the case of discrete PROCESS objects. A continuous STEPPER integrates the values of VARIABLE objects from the velocities given by the continuous PROCESS objects, and determines when the velocities should be recalculated by the PROCESS objects. A typical application of continuous PROCESS and STEPPER objects is to implement differential equations and differential equation solvers, respectively, to form a simulation system of the system of differential equations.

4.4.2 Some Available Discrete Classes

Followings are some available discrete classes.

NRStepper and GillespieProcess (Gillespie-Gibson pair)

An example of discrete-event simulation method provided by ECELL is a variant of Gillespie's stochastic algorithm, the Next Reaction Method, or Gillespie-Gibson algorithm. NRStepper class implements this algorithm. When this STEPPER is used in conjunction with GillespieProcess objects, which is a subclass of DiscreteEventProcess and calculates a time of the next occurence of the reaction using Gillespie's reaction probability equation and a random number, ECELL conducts a Gillespie-Gibson stochastic simulation of elementary chemical reactions. In fact, the Next Reaction Method is nothing but a standard discrete event simulation algorithm, and NRStepper is just an alias of the DiscreteEventStepper class.

Usage of this pair of classes of objects is simple: just set the StepperID, VariableReferenceList and the rate constant property k of those GillespieProcess objects.

DiscreteTimeStepper

A type of discrete STEPPER that is provided by the system is *DiscreteTimeStepper*. This class of STEPPER, when instantiated, calls all discrete PROCESS objects with a fixed user-specified time-interval. For example, if the model has a DiscreteTimeStepper with 0.001 (second) of StepInterval property, it fires all of its PROCESS objects every milli-second. DiscreteTimeStepper is discrete time because it does not have time between steps; it ignores a signal from other STEPPER objects (*STEPPER interruption*) that notifies a change of system state (values of VARIABLE objects) that may affect its PROCESS objects. Such a change is reflected in the next step.

PassiveStepper

Another class of discrete STEPPER is PassiveStepper. This can partially be seen as a DiscreteTimeStepper with an infinite StepInterval, but there is a difference. Unlike DiscreteTimeStepper, this does *not* ignore STEPPER interruptions, which notify change in the system state that may affect this STEPPER's PROCESS objects.

This STEPPER is used when some special procedures (coded in discrete PROCESS objects) must be invoked when other STEPPER object may have changed a value or a velocity of at least one VARIABLE that this STEPPER's PROCESS objects accesses.

PythonProcess

PythonProcess allows users to script a PROCESS object in full PYTHON syntax.

initialize() and fire() methods can be scripted with InitializeMethod and FireMethod properties, respectively.

PythonProcess can be either discrete or continuous. This 'operation mode' can be specified by setting IsContinuous property. The default is false (0), or discrete. To switch to the continuous mode, set 1 to the property:

```
Process PythonProcess( PY1 )
{
    IsContinuous 1;
}
```

In addition to regular PYTHON constructs, the following objects, methods, and attributes are available in both of the method properties (InitializeMethod and FireMethod):

- Properties

  PythonProcess accepts arbitrary names of properties. For example, the following code creates two new properties.

```
Process PythonProcess( PY1 )
{
    NewProperty "new property";
    KK          3.0;
}
```

These properties can be use in PYTHON methods:

```
Process PythonProcess( PY1 )
{
    # ... NewProperty and KK are set

    InitializeMethod "print NewProperty";

    FireMethod '''
KK += 1.0
print KK
''';
}
```

A new property can also be created within PYTHON methods.

```
InitializeMethod "A = 3.0"; # A is created
FireMethod "print A * 2";   # A can be used here
```

These properties are treated as a global variable.

- Objects
  - `self`

    This is the PROCESS object itself. This has the following attributes:

    * Activity

      The current value of Activity property of this PROCESS.

    * addValue(`value`)

      Add each VARIABLEREFERENCE the `value` multiplied by the coefficient of the VARIABLEREFERENCE.

      Using this method implies that this PROCESS is discrete. Check that IsContinuous property is false.

    * getSuperSystem()

      This method gets the super system of this PROCESS. See below for the attributes of SYSTEM objects.

    * Priority

      The Priority property of this PROCESS.

    * setFlux(`value`)

      Add each VARIABLEREFERENCE's velocity the `value` multiplied by the coefficient of the VARIABLEREFERENCE.

      Using this method implies that this PROCESS is continuous. Check that IsContinuous property is true.

    * StepperID

      StepperID of this PROCESS.

- VARIABLEREFERENCE

  VARIABLEREFERENCE instances given in the VariableReferenceList property of this PROCESS can be used in the PYTHON methods. Each instance has the following attributes:

  * addFlux(`value`)

    Multiply the `value` by the Coefficient of this VARIABLEREFERENCE, and add that to the VARIABLE's velocity.

  * addValue(`value`)

    Add the `value` to the Value property of the VARIABLE.

  * addVelocity(`value`)

    Add the `value` to the Velocity property of the VARIABLE.

  * Coefficient

    The coefficient of the VARIABLEREFERENCE

  * getSuperSystem()

    Get the super system of the VARIABLE. A SYSTEM object is returned.

  * MolarConc

    The concentration of the VARIABLE in Molar [M].

  * Name

    The name of the VARIABLEREFERENCE.

  * NumberConc

    The concentration in number [num / size of the VARIABLE's super system.]

  * IsFixed

    Zero if the Fixed property of the VARIABLE is false. Otherwise a non-zero integer.

  * IsAccessor

    Zero if the IsAccessor flag of the VARIABLEREFERENCE is false. Otherwise a non-zero integer.

  * TotalVelocity

    The total current velocity. Usually of no use.

  * Value

    The value of the VARIABLE

  * Velocity

    The provisional velocity given by the currently stepping STEPPER. Usually of no use.

- SYSTEM

  A SYSTEM object has the following attributes.

  * getSuperSystem()

    Get the super system of the SYSTEM. A SYSTEM object is returned.

  * Size

    The size of the SYSTEM.

  * SizeN_A

Equivalent to ‘‘Size * N_A‘‘, where N_A is a Avogadro’s number.

* StepperID

  The StepperID of the SYSTEM.

Here is an example uses of PythonProcess.

```
Process PythonProcess( PY1 )
{
    # IsContinuous 0; -- default
    FireMethod "S1.Value = S2.Value + S3.Value";
    VariableReferenceList [(S1)] [(S2)] [(S3)];
}
```

PythonEventProcess

This class enables users PYTHON scripting of time-events. In addition to initialize() and fire(), updateStepInterval() method can be scripted with this class. Use UpdateStepIntervalMethod property to set this.

In addition to those of PythonProcess, the `self` object of PythonEventProcess has some more attributes:

- StepInterval

  The most recent StepInterval calculated by the updateStepInterval() method.

- DependentProcessList

  This attribute holds a tuple of IDs of dependent PROCESSes of this PROCESS.

This class of objects must be used with a DiscreteEventStepper.

This class is under development.

Other discrete classes

STEPPER classes for explicit and implicit tau leaping algorithms are under development.

A flux-distribution method for hybrid dynamic/static simulation of biochemical pathways is available with the following classes: FluxDistributionStepper, FluxDistributionProcess, QuasiDynamicFluxProcess. Usage of this scheme is to be described.

4.4.3 Some Available Continuous Classes

ECELL supports both Ordinary Differential Equation (ODE) and Differential-Algebraic Equation (DAE) models, and has STEPPER classes for each type of formalisms.

Also, the system is shipped with some continuous PROCESS classes. For example, MassActionFluxProcess calculates a reaction rate according to the law of mass action. ExpressionFluxProcess allows users to describe arbitrary rate equations in model files. PythonProcess and PythonFluxProcess are used to script PROCESS objects in PYTHON. Some enzyme kinetics rate laws are also available.

Generic ordinary differential Steppers

If your model is a system of ODEs, then in this version of the software (version APPVERSION) the recommended choice is ODEStepper. This STEPPER is a high-performance replacement of ODE45Stepper, which was the choice for the previous versions.

ODEStepper is implemented so that it can adaptively switch the solving method between the implicit one (Radau IIA) and the explicit one (Dormand-Prince), according to the current stiffness of the input.

Some other available ODE STEPPER classes are ODE23Stepper, which employes a lower (the second) order integration algorithm, and FixedODE1Stepper that implements the simplest Euler algorithm without an adaptive step sizing mechanism.

These ODE STEPPER classes except for the FixedODE1Stepper have some common property slots for user-specifiable parameters. Here is a partial list:

- Tolerance

  An error tolerance in local truncation error. Giving this smaller numbers forces the STEPPER to take smaller step sizes, and slows down the simulation. Greater numbers results in faster run with sacrifice of accuracy. A typical number is 1e-6.

- MinStepInterval

  Species the minimum value of step width. This limit precedes the Tolerance property above.

  These properties can also be useful to completely disable the adaptive step size control mechanism: set the same number to both of the property slots.

- MaxStepInterval

  This property is no longer supported and has no specific effect if it is set

MassActionFluxProcess

MassActionFluxProcess is a class of PROCESS for simple mass-actions. This class calculates a flux rate according to the irreversible mass-action. Use a property k to specify a rate constant.

ExpressionFluxProcess

ExpressionFluxProcess is designed for easy and efficient representations of continuous flux rate equations.

Expression property of this class accepts a plain text rate expression. The expression must be evaluated to give a flux rate in [number / second]. (Note that this is a number per second, not concentration per second.) Here is an example use of ExpressionFluxProcess:

```
Process ExpressionFluxProcess( P1 )
{
    k 0.1;
    Expression "k * S.Value";

    VariableReferenceList [ S :.:S -1 ] [ P :.:P 1 ];
}
```

Compared to PythonProcess or PythonFluxProcess below, it runs significantly faster with sacrifice of some flexibility in scripting.

The following shows elements those can be used in the Expression property. The set of available arithmetic operators and mathematical functions are meant to be equivalent to SBML level 2, except control structures.

- Constants

  Numbers (e.g. 10, 10.33, 1.33e-5), `true`, `false` (equivalent to zero), `pi` (Pi), `NaN` (Not-a-Number), `INF` (Infinity), `N_A` (Avogadro's number), `exp` (the base of natural logarithms).

- Arithmetic operators

  +, −, *, /, ^ (power; this can equivalently be written as `pow( x, y )`).

- Built-in functions

  `abs`, `ceil`, `exp`, `*fact`, `floor`, `log`, `log10`, `pow` `sqrt`, `*sec`, `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `coth`, `*csch`, `*sech`, `*asin`, `*acos`, `*atan`, `*asec`, `*acsc`, `*acot`, `*asinh`, `*acosh`, `*atanh`, `*asech`, `*acsch`, `*acoth`. (Functions with astarisk '*' are currently not available on the Windows version.)

  All functions but `pow` are unary functions. `pow` is a binary function.

- Properties

  Similar to PythonProcess, ExpressionFluxProcess accepts arbitrary name properties in the model. Unlike PythonProcess, however, these properties of this class can hold only REAL values.

- Objects
  - `self`

    This PROCESS object itself. This has the following attribute which is a sub set of that of PythonProcess:

    * getSuperSystem()

  - VARIABLEREFERENCE

    VARIABLEREFERENCE instances given in the VariableReferenceList property of this PROCESS can be used in the expression. Each instance has the following set of attributes, which is a sub set of that of PythonProcess:

    * Value
    * MolarConc
    * NumberConc
    * TotalVelocity
    * Velocity

  - SYSTEM

    A SYSTEM object has the following two attributes.

    * Size
    * SizeN_A

Below is an example of the basic Michaelis-Menten reaction programmed with the ExpressionFluxProcess.

```
Process ExpressionFluxProcess( P )
{
    Km    1.0;
    Kcat  10;

    Expression "E.Value * Kcat * S.MolarConc / ( S.MolarConc + Km )";

    VariableReferenceList [ S :.:S -1 ] [ P :.:P 1 ] [ E :.:E 0 ];
}
```

Some pre-defined reaction rate classes

See the standard dynamic module library reference for availability of some enzyme kinetics PROCESS classes.

PythonFluxProcess

PythonFluxProcess is almost the same as PythonProcess, except that (1) it takes just a PYTHON expression (instead of statements) to its Expression property, and (2) similar to ExpressionFluxProcess, the evaluated value of the expression is implicitly passed to the setFlux() method.

Generic differential-algebraic Steppers

For DAE models, use DAEStepper. The model must form a valid index-1 DAE system. When a DAE STEPPER detects one or more discrete PROCESS objects, it assumes that these are *algebraic PROCESS* objects. Thus, all discrete PROCESS objects in a DAE STEPPER must be algebraic. See below for what is algebraic PROCESS.

> **Note**
>
> Because it can be viewed that ODE is a special case of DAE problems which does not have a algebraic equations, but only differential equations, a DAE STEPPER can be used to run an ODE model. However, ODE Steppers are specialized for ODE problems, in terms of both the selection of integration algorithms and implementation issues, and generally use of an ODE STEPPER benefits in performance when the model is a system of ODEs.

Those properties of ODE STEPPER classes described above (such as the Tolerance property) are also available for DAE STEPPER classes.

Algebraic Processes

This is a type of discrete PROCESS, but placed here because it is used with a DAE STEPPER, which is continuous.

In principle, continuous PROCESS objects must be connected with continuous STEPPER instances, and a discrete STEPPER is assumed to take only discrete PROCESS objects. However, there are some exceptions. One of such is the *algebraic processes*. Strangely enough, in DAE simulations, seemingly discrete algebraic equations are solved continuously in conjunction with other differential equations.

Algebraic equations in ECELL has the following form:

```
0 = g( t, x )
```

where t is the time and x is a vector of variable references.

The DAE solver system of ECELL uses Activity property of PROCESS objects to represent the value of the algebraic function `g( t, x )`. An algebraic PROCESS must *not* change values of VARIABLE objects explicitly. The DAE STEPPER does this job of finding a point where the equation `g()` holds.

When modeling, be careful about coefficients of VARIABLEREFERENCEs of an algebraic PROCESS. In most cases, simply set unities. The solver respects these numbers when solving the system. For example, if the coefficient of `A` is zero, it does not change the variable when trying to find the solution, while it is used in the calculation of the equation.

As a means of describing algebraic equations, ExpressionAlgebraicProcess is available. The usage is the same as ExpressionFluxProcess, except that the evaluation of its expression is interpreted as the value of the algebraic function `g()`.

The following examble describes an equation

```
a * A + B = 10,  a = 1.5
```

```
Stepper DAEStepper( DAE1 ) {}

Process ExpressionAlgebraicProcess( P )
{
```

```
    StepperID DAE1;

    a    1.5;

    Expression "( a * A + B ) - 10";

    VariableReferenceList [ A :.:A 1 ] [ B :.:B 1 ];
}
```

To use C++ or PythonProcess for algebraic equations, call setActivity() method to set the value of the equation. The following is an example with a PythonProcess:

```
Process PythonProcess( PY )
{
    a    1.5;

    FireMethod "self.setActivity( ( a * A + B ) - 10 )";

    VariableReferenceList [ A :.:A 1 ] [ B :.:B 1 ];
}
```

Power-law canonical DEs (S-System and GMA)

ESSYNSStepper supports S-System and GMA simulations by using the ESSYNS algorithm. A ESSYNSStepper must be connected with either a SSystemProcess or a GMAProcess as its sole VARIABLEREFERENCE. Use SSystemMatrix or GMAMatrix property to set the system parameters.

A sample model under the directory `doc/sample/ssystem/` gives an example usage.

These modules are still under development. More descriptions to come...

4.5 Modeling Convensions

4.5.1 Units

In APP, the following units are used. This standard is meant only for the simulator's internal representation, and any units can be used in the process of modeling. However, it must be converted to these standard units before loaded by the simulator.

- Time

  s (second)

- Volume

  L (liter)

- Concentration

  Molar concentration (M, or molar per L (liter), used for example in MolarConc property of a VARIABLE object) or,

  Number concentration (number per L (liter), NumberConc property of VARIABLE has this unit).

CHAPTER 5

Modeling Tutorial

This chapter is a simple modeling tutorial using ECELL.

5.1 Running the model

All the examples in this section can be run by the following procedure.

1. Create and save the model file (for example, `simple-gillespie.em`) with a text editor.
2. Convert the EM file to an EML file by `ecell3-em2eml` command.
3. Run it in GUI mode with `gecell` command.

   or, in the script mode with `ecell3-session` command (see the following chapter):

5.2 Using Gillespie algorithm

APP comes with a set of classes for simulations using Gillespie's stochastic algorithm.

5.2.1 A Trivial Reaction System

At the very first, let us start with the simplest possible stable system of elementary reactions, which has two variables (in this case the numbers of molecules of molecular species) and a couple of elementary reaction processes. Because elementary reactions are irreversible, at least two instances of the reactions are needed for the system to be stable. The reaction system looks like this: – P1 –> S1 S2 <– P2 – `S1` and `S2` are molecular species, and `P1` and `P2` are reaction processes. Rate constants of both reactions are the same: 1.0 [1/s]. Initial numbers of molecules of `S1` and `S2` are 1000 and 0, respectively. Because rate constants are the same, the system has a steady state at `S1 == S2 == 500`.

5.2.2 Specifying the Next Reaction method

NRStepper class implements Gibson's efficient variation of the Gillespie algorithm, or the Next Reaction (NR) method.

To use the NRStepper in your simulation model, write like this in your EM file:

```
Stepper NRStepper( NR1 )
{
    # no property
}
```

In this example, the NRStepper has the StepperID 'NR1'. For now, no property needs to be specified for this object.

5.2.3 Defining the compartment

Next, define a compartment, and connect it to the STEPPER NR1. Because this model has only a single compartment, we use the root sytem (/). Although this model does not depend on size of the compartment because all reactions are first order, it is a good idea to always define the size explicitly rather than letting it defaults to 1.0. Here we set it to 1e-15 [L].

```
System System( / )
{
    StepperID       NR1;

    Variable Variable( SIZE ) { Value 1e-15; }

    # ...
}
```

5.2.4 Defining the variables

Now define the main variables S1 and S2. Use 'Value' properties of the objects to set the initial values.

```
System System( / )
{
    # ...

    Variable Variable( S1 )
    {
        Value   1000;
    }

    Variable Variable( S2 )
    {
        Value   0;
    }

    # ...
}
```

5.2.5 Defining reaction processes

Lastly, create reaction process instances P1 and P2. GillespieProcess class works together with the NRStepper to simulate elementary reactions.

Two different types of properties, k and VariableReferenceList, must be set for each of the GillespieProcess object. k is the rate constant parameter in [1/sec] if the reaction is first-order, or [1/sec/M] if it is a second-order reaction. (Don't forget to define SIZE VARIABLE if there is a second-order reaction.) Set VariableReferenceList properties so that P1 consumes S1 and produce S2, and P2 uses S2 to create S1.

```
System System( / )
{
    # ...

    Process GillespieProcess( P1 )              # the reaction S1 --> S2
    {
```

```
        VariableReferenceList   [ S :.:S1 -1 ]
                                [ P :.:S2 1 ];
        k       1.0;                            # the rate constant
    }

    Process GillespieProcess( P2 )              # the reaction S2 --> S1
    {
        VariableReferenceList   [ S :.:S2 -1 ]
                                [ P :.:S1 1 ];
        k       1.0;
    }
}
```

5.2.6 Putting them together

Here is the complete EM of the model that really works. Run this model with `gecell` and open a TracerWindow to plot trajectories of `S1` and `S2`. You will see those two VARIABLEs immediately reaching the steady state around 500.0. If you zoom around the trajectories, you will be able to see stochastic fluctuations.

```
Stepper NRStepper( NR1 )
{
    # no property
}

System System( / )
{
    StepperID       NR1;

    Variable Variable( SIZE ) { Value 1e-15; }

    Variable Variable( S1 )
    {
        Value   1000;
    }

    Variable Variable( S2 )
    {
        Value   0;
    }

    Process GillespieProcess( P1 )              # the reaction S1 --> S2
    {
        VariableReferenceList   [ S :.:S1 -1 ]
                                [ P :.:S2 1 ];
        k       1.0;                            # the rate constant
    }

    Process GillespieProcess( P2 )              # the reaction S2 --> S1
    {
        VariableReferenceList   [ S :.:S2 -1 ]
                                [ P :.:S1 1 ];
        k       1.0;
    }
}
```

5.3 Using Deterministic Differential Equations

The previous section described how to create a model that runs with the stochastic Gillespie's algorithm. ECELL is a multi-algorithm simulator, and different algorithms can be used to run the model. This section explains a way to use a deterministic differential equation solver to run the system of simple mass-action reactions.

5.3.1 Choosing Stepper and Process classes

In the current version, we recommend ODE45Stepper class as a general-purpose STEPPER for differential equation systems. This STEPPER implements an explicit fourth order numerical integration algorithm with a fifth-order error control.

MassActionFluxProcess is the continuous differential equation conterpart of the discrete-event GillespieProcess. Unlike GillespieProcess, MassActionFluxProcess does not have limitation in the order of the reaction mechanism. For example, it can handle the reaction like this: `S0 + S1 + 2 S2 --> P0 + P1`.

5.3.2 Converting the model

Converting the trivial reaction system model for Gillespie to use differential equations is very easy; just replace NRStepper with ODE45Stepper, and change the classname of GillespieProcess to MassActionFluxProcess.

The following is the model of the trivial model that runs on the differential ODE45Stepper. You will get similar simulation result than the stochastic model in the previous section. However, if you zoom, you would notice that the stochastic fluctuation can no longer be observed because the model is turned from stochastic to deterministic.

```
Stepper ODE45Stepper( ODE1 )
{
    # no property
}

System System( / )
{
    StepperID       ODE1;

    Variable Variable( SIZE ) { Value 1e-15; }

    Variable Variable( S1 )
    {
        Value   1000;
    }

    Variable Variable( S2 )
    {
        Value   0;
    }

    Process MassActionFluxProcess( P1 )
    {
        VariableReferenceList   [ S0 :.:S1 -1 ]
                                [ P0 :.:S2 1 ];
        k       1.0;
    }

    Process MassActionFluxProcess( P2 )
    {
```

```
        VariableReferenceList   [ S0 :.:S2 -1 ]
                                [ P0 :.:S1 1 ];
        k       1.0;
    }
}
```

5.4 Making the Model Switchable Between Algorithms

Fortunately, at least as far as the model has only elementary reactions, switching between these stochastic and deterministic algorithms is just to switch between NRStepper/GillespieProcess pair and ODE45Stepper/MassActionFluxProcess pair of classnames. Both PROCESS classes takes the same property 'k' with the same unit.

Some use of EMPY macros makes the model generic. In the following example, setting the PYTHON variable `TYPE` to `ODE` makes it run in deterministic differential equation mode, and setting `TYPE` to `NR` turns it stochastic.

```
@{ALGORITHM='ODE'}

@{
if ALGORITHM == 'ODE':
    STEPPER='ODE45Stepper'
    PROCESS='MassActionFluxProcess'
elif ALGORITHM == 'NR':
    STEPPER='NRStepper'
    PROCESS='GillespieProcess'
else:
    raise 'unknown algorithm: %s' % ALGORITHM
}

Stepper @(STEPPER)( STEPPER1 )
{
    # no property
}

System System( / )
{
    StepperID       STEPPER1;

    Variable Variable( SIZE ) { Value 1e-15; }

    Variable Variable( S1 )
    {
        Value   1000;
    }

    Variable Variable( S2 )
    {
        Value   0;
    }

    Process @(PROCESS)( P1 )
    {
        VariableReferenceList   [ S0 :.:S1 -1 ]
                                [ P0 :.:S2 1 ];
        k       1.0;
    }
```

```
    Process @(PROCESS)( P2 )
    {
        VariableReferenceList   [ S0 :.:S2 -1 ]
                                [ P0 :.:S1 1 ];
        k       1.0;
    }
}
```

5.5 A Simple Deterministic / Stochastic Composite Simulation

ECELL can drive a model with multiple STEPPER objects. Each STEPPER can implement different simulation algorithms, and have different time scales. This framework of multi-algorithm, multi-timescale simulation is quite generic, and virtually any combination of any number of different types of sub-models is possible. This section exemplifies a tiny model of coupled ODE and Gillespie reactions.

5.5.1 A tiny multi-timescale reactions model

Consider this tiny model of four VARIABLEs and six reaction PROCESSes: – P1 –> – P4 –> S1 S2 – P3 –> S3 S4 ^ <– P2 – <– P5 – | | | \ _____ P6 _____/ Although it may look complicated at first glance, this system consists of two instances of the 'trivial' system we have modeled in the previous sections coupled together: Sub-model 1: – P1 –> S1 S2 <– P2 – and Sub-model 2: – P4 –> S3 S4 <– P5 – These two sub-models are in turn coupled by reaction processes `P3` and `P6`. Because time scales of `P3` and `P6` are determined by `S2` and `S4`, respectively, `P3` belongs to the sub-model 1, and `P6` is a part of the sub-model 2. Sub-model 1: S2 – P3 –> S3 S1 <– P6 –> S4 :Sub-model 2 Rate constants of the main reactions, `P1`, `P2`, `P4`, and `P5` are the same as the previous model: `1.0` [1/sec]. But the 'bridging' reactions are slower than the main reactions: `1e-1` for `P3` and `1e-3` for `P6`. Consequently, sub-models 1 and 2 would have approximately `1e-1 / 1e-3 == 1e-2` times different steady-state levels. Because the rate constants of the main reactions are the same, this implies time scale of both sub-models are different.

5.5.2 Writing model file

The following code implements the multi-time scale model. The first two lines specify algorithms to use for those two parts of the model. `ALGORITHM1` variable specifies the algorithm to use for the sub-model 1, and `ALGORITHM2` is for the sub-model 2. Values of these variables can either be `'NR'` or `'ODE'`.

For example, to try pure-stochastic simulation, set these variables like this:

```
@{ALGORITHM1='NR'}
@{ALGORITHM2='NR'}
```

Setting `ALGORITHM1` to `'NR'` and `ALGORITHM2` to `'ODE` would be an ideal configuration. This runs a magnitude faster than the pure-stochastic configuration.

```
@{ALGORITHM1='NR'}
@{ALGORITHM2='ODE'}
```

Also try pure-deterministic run.

```
@{ALGORITHM1='ODE'}
@{ALGORITHM2='ODE'}
```

In this particular model, this configuration runs very fast because the system easily reaches the steady-state and stiffness of the model is low. However, this does not necessary mean pure-ODE is always the fastest. Under some situations NR/ODE composite simulation exceeds both pure-stochastic and pure-deterministic (reference?).

```
@{ALGORITHM1= ['NR' or 'ODE']}
@{ALGORITHM2= ['NR' or 'ODE']}


# a function to give appropriate class names.
@{
def getClassNamesByAlgorithm( anAlgorithm ):
    if anAlgorithm == 'ODE':
        return 'ODE45Stepper', 'MassActionFluxProcess'
    elif anAlgorithm == 'NR':
        return 'NRStepper', 'GillespieProcess'
    else:
        raise 'unknown algorithm: %s' % ALGORITHM1
}

# get classnames
@{
STEPPER1, PROCESS1 = getClassNamesByAlgorithm( ALGORITHM1 )
STEPPER2, PROCESS2 = getClassNamesByAlgorithm( ALGORITHM2 )
}

# create appropriate steppers.
# stepper ids are the same as the ALGORITHM.
@('Stepper %s ( %s ) {}' % ( STEPPER1, ALGORITHM1 ))

# if we have two different algorithms, one more stepper is needed.
@(ALGORITHM1 != ALGORITHM2 ? 'Stepper %s( %s ) {}' % ( STEPPER2, ALGORITHM2 ))



System CompartmentSystem( / )
{
    StepperID   @(ALGORITHM1);

    Variable Variable( SIZE ) { Value 1e-15; }

    Variable Variable( S1 )
    {
        Value   1000;
    }

    Variable Variable( S2 )
    {
        Value   0;
    }

    Variable Variable( S3 )
    {
        Value   1000000;
    }

    Variable Variable( S4 )
    {
        Value   0;
    }
```

```
    Process @(PROCESS1)( P1 )
    {
        VariableReferenceList   [ S0 :.:S1 -1 ] [ P0 :.:S2 1 ];
        k       1.0;
    }

    Process @(PROCESS1)( P2 )
    {
        VariableReferenceList   [ S0 :.:S2 -1 ] [ P0 :.:S1 1 ];
        k       1.0;
    }

    Process @(PROCESS1)( P3 )
    {
        VariableReferenceList   [ S0 :.:S2 -1 ] [ P0 :.:S3 1 ];
        k       1e-1;
    }

    Process @(PROCESS2)( P4 )
    {
        StepperID @(ALGORITHM2);

        VariableReferenceList   [ S0 :.:S3 -1 ] [ P0 :.:S4 1 ];
        k       1.0;
    }

    Process @(PROCESS2)( P5 )
    {
        StepperID @(ALGORITHM2);

        VariableReferenceList   [ S0 :.:S4 -1 ] [ P0 :.:S3 1 ];
        k       1.0;
    }

    Process @(PROCESS2)( P6 )
    {
        StepperID @(ALGORITHM2);

        VariableReferenceList   [ S0 :.:S4 -1 ] [ P0 :.:S1 1 ];
        k       1e-4;
    }

}
```

5.6 Custom equations

5.6.1 Complex flux rate equations

The simplest way to script custom rate equations is to use ExpressionFluxProcess. Here is an example taken from the Drosophila sample model which you can find under `${datadir}/doc/ecell/samples/Drosophila` [1] In this expression, Size * N_A of the supersystem of the PROCESS is used to keep the unit of the expression [num /

[1] `${datadir}` refers to the directory either given to `--datadir` option of `configure` script or `${prefix}/share`. On Windows, `${prefix}` would be the directory to which the application is installted.

second].

```
Process ExpressionFluxProcess( R_toy1 )
{
    vs      0.76;
    KI      1;
    Expression "(vs*KI) / (KI + C0.MolarConc ^ 3)
                                        * self.getSuperSystem().SizeN_A";

    VariableReferenceList   [ P0 :.:M 1 ] [ C0 :.:Pn 0 ];
}
```

FIXME: some more examples

5.6.2 Algebraic equations

Use of ExpressionAlgebraicProcess is the easiest method to describe algebraic equations.

Be careful about the coefficients of the VARIABLEREFERENCEs. (Usually just set unities.)

FIXME: some more examples here

5.7 Other Modeling Schemes

5.7.1 Discrete events

CHAPTER 6

Scripting A Simulation Session

By reading this chapter, you can get information about the following items: What is ECELL Session Script (ESS)., How to run ESS in scripting mode., How to use ESS in GUI mode., How to automate a simulation run by writing an ESS file., How to write frontend software components for ECELL in PYTHON.

6.1 Session Scripting

An ECELL Session Script (ESS) is a PYTHON script which is loaded by a ECELL SESSION object. A SESSION instance represents a single run of a simulation.

An ESS is used to automate a single run of a simulation session. A simple simulation run typically involves the following five stages:

1. Loading a model file.

   Usually an EML file is loaded.

2. Pre-simulation setup of the simulator.

   Simulator and model parameters, such as initial values of VARIABLE objects and property values of PROCESS objects, are set and/or altered. Also, data LOGGERs may be created in this phase.

3. Running the simulation.

   The simulation is run for a certain length of time.

4. Post-simulation data processing.

   In this phase, the resulting state of the model after the simulation and the data logged by the LOGGER objects are examined. The simulation result may be numerically processed. If necessary, go back to the previous step and run the simulation for more seconds.

5. Data saving.

   Finally, the processed and raw simulation result data are saved to files.

An ESS file usually has an extension '`.py`'.

6.2 Running ECELL Session Script

There are three ways to execute ESS;

- Execute the script from the operating system's command line (the shell prompt).

- Load the script from frontend software such as OSOGO.
- Use SESSIONMANAGER to automate the invokation of the simulation sessions itself. This is usually used to write mathematical analysis scripts, such as parameter tuning, which involves multiple runs of the simulator.

6.2.1 Running ESS in command line mode

An ESS can be run by using ECELL3-SESSION command either in *batch mode* or in *interactive mode*.

Batch mode

To execute an ESS file without user interaction, type the following command at the shell prompt:

ECELL3-SESSION command creates a simulation SESSION object and executes the ESS file `ess.py` on it. The option [-e] can be omitted. Optionally, if [-f model.eml] is given, the EML file `model.eml` is loaded immediately before executing the ESS.

Interactive mode

To run the ECELL3-SESSION in interactive mode, invoke the command without an ESS file.

```
ecell3-session [ for E-Cell SE Version 3, on Python Version 2.2.1 ]
Copyright (C) 1996-2014 Keio University.
Send feedback to Koichi Takahashi
```

The banner and the prompt shown here may vary according to the version you are using. If the option [-f model.eml] is given, the EML file `model.eml` is loaded immediately before prompting.

Giving parameters to the script

Optionally *session parameters* can be given to the script. Given session parameters can be accessible from the ESS script as global variables (see the following section).

To give the ESS parameters from the ECELL3-SESSION command, use either `-D` or `--parameters=` option.

Both ways, `-D` and `--parameters`, can be mixed.

6.2.2 Loading ESS from OSOGO

To manually load an ESS file from the GUI, use File->loadScript menu button.

GECELL command accepts `-e` and `-f` options in the same way as the ECELL3-SESSION command.

6.2.3 Using SessionManager

(a separate chapter?)

6.3 Writing ECELL Session Script

The syntax of ESS is a full set of PYTHON language with some convenient features.

6.3.1 Using Session methods

General rules

In ESS, an instance of SESSION is given, and any methods defined in the class can be used as if it is defined in the global namespace.

For example, to run the simulation for 10 seconds, use run method of the SESSION object. self.run(10) where self. points to the current SESSION object given by the system. Alternatively, you can use `theSession` in place of the `self`. theSession.run(10)

Unlike usual PYTHON script, you can omit the object on which the method is called if the method is for the current SESSION. run(10)

Loading a model

Let's try this in the interactive mode of the ECELL3-SESSION command. On the prompt of the command, load an EML file by using loadModel() method. `ecell3-session>>> ``\ ``loadModel('simple.eml')` Then the prompt changes from ‘‘ecell3-session>>> ‘‘ to ‘‘, t=>>> ‘‘ ‘‘simple.eml, t=0>>> ‘‘

Running the simulation

To proceed the time by executing the simulation, step() and run() methods are used.

step(`n`) conducts `n` steps of the simulation. The default value of `n` is 1.

> **Note**
>
> In above example you may notice that the first call of step() does not cause the time to change. The simulator updates the time at the beginning of the step, and calculates a tentative step size after that. The initial value of the step size is zero. Thus it needs to call step() twice to actually proceed the time. See chapter 6 for details of the simulation mechanism.

To execute the simulation for some seconds, call run method with a duration in seconds. (e.g. run(`10`) for 10 seconds.) run method steps the simulation repeatedly, and stops when the time is proceeded for the given seconds. In other words, the meaning of run(`10`) is to run the simulation *at least* 10 seconds. It always overrun the specified length of time to a greater or less.

The system supports run without an argument to run forever, if and only if both *event checker* and *event handler* are set. If not, it raises an exception. See setEventChecker() in the method list of Session class.

Getting current time

To get the current time of the simulator, getCurrentTime() method can be used.

Printing messages

You may want to print some messages in your ESS. Use message(`message`) method, where `message` argument is a string to be outputed.

By default the message is handled in a way the same as the Python's print statement; it is printed out to the standard out with a trailing new line. This behavior can be changed by using setMessageMethod() method.

An example of using SESSION methods

Here is a tiny example of using SESSION methods which loads a model, run a hundred seconds, and print a short message.

```
loadModel( 'simple.eml' )
run( 100 )
message( 'stopped at %f seconds.' % getCurrentTime() )
```

6.3.2 Getting Session Parameters.

Session parameters are given to an ESS as global variables. Therefore usage of the session parameters is very simple. For example, if you can assume a session parameter `MODELFILE` is given, just use it as a variable: loadModel(MODELFILE)

To check what parameters are given to ESS, use dir() or globals() built-in functions. Session parameters are listed as well as other available methods and variables. To check if a certain ESS parameter or a global variable is given, write an if statement like this: if 'MODELFILE' in globals(): # MODELFILE is given else: # not given

Note

Currently there is no way to distinguish the Session parameters from other global variables from ESS.

6.3.3 Observing and Manipulating the Model with OBJECTSTUBs

What is OBJECTSTUB?

OBJECTSTUB is a proxy object in the frontend side of the system which corresponds to an internal object in the simulator. Any operations on the simulator's internal objects should be done via the OBJECTSTUB.

There are three types of OBJECTSTUB:

- ENTITYSTUB
- STEPPERSTUB
- LOGGERSTUB

each correspond to ENTITY, STEPPER, and LOGGER classes in the simulator, respectively.

Why OBJECTSTUB is needed

OBJECTSTUB classes are actually thin wrappers over the ecell.ecs.Simulator class of the E-Cell Python Library, which provides object-oriented appearance to the flat procedure-oriented API of the class. Although SIMULATOR object can be accessed directly via `theSimulator` property of SESSION class, use of OBJECTSTUB is encouraged.

This backend / frontend isolation is needed because lifetimes of backend objects are not the same as that of frontend objects, nor are their state transitions necessarily synchronous. If the frontend directly manipulates the internal objects of the simulator, consistency of the lifetime and the state of the objects can easily be violated, which must not happen, without some complicated and tricky software mechanism.

Creating an OBJECTSTUB by ID

To get an OBJECTSTUB object, createEntityStub(), createStepperStub(), and createLoggerStub() methods of SESSION class are used.

For example, to get an ENTITYSTUB, call the createEntityStub() method with a *FullID* string:

```
= createEntityStub(  )
```

Similarly, a STEPPERSTUB object and a LOGGERSTUB object can be retrieved with a *StepperID* and a *FullPN*, respectively.

```
= createStepperStub(  )
```

```
= createLoggerStub(  )
```

Creating and checking existence of the backend object

Creating an OBJECTSTUB does not necessarily mean a corresponding object in the simulator backend exists, or is created. In other words, creation of the OBJECTSTUB is purely a frontend operation. After creating an OBJECTSTUB, you may want to check if the corresponding backend object exists, and/or to command the backend to create the backend object.

To check if a corresponding object to an OBJECTSTUB exists in the simulator, use exists() method. For example, the following if statement checks if a Stepper whose ID is `STEPPER_01` exists: aStepperStub = createStepperStub('STEPPER_01') if aStepperStub.exists(): # it already exists else: # it is not created yet

To create the backend object, just call create() method. aStepperStub.create()# Stepper 'STEPPER_01' is created here

Getting the name and a class name from an OBJECTSTUB

To get the name (or an ID) of an OBJECTSTUB, use getName() method.

To get the class name of an ENTITYSTUB or a STEPPERSTUB, call getClassName() method. This operation is not applicable to LOGGERSTUB.

Setting and getting properties

As described in the previous chapters, ENTITY and STEPPER objects has *properties*. This section describes how to access the object properties via OBJECTSTUBs. This section is not applicable to LOGGERSTUBs.

To get a property value from a backend object by using an ENTITYSTUB or a STEPPERSTUB, invoke getProperty() method or access an object attribute with a property name: aValue = aStub.getProperty('Activity') or equivalently, aValue = aStub['Activity']

To set a new property value to an ENTITY or a STEPPER, call setProperty() method or mutate an object attribute with a property name and the new value: aStub.getProperty('Activity', aNewValue) or equivalently, aStub['Activity'] = aNewValue

List of all the properties can be gotten by using getPropertyList method, which returns a list of property names as a Python TUPLE containing string objects. aStub.getPropertyList()

To know if a property is *getable* (accessible) or *setable* (mutable), call getPropertyAttribute() with the name of the property. The method returns a Python TUPLE whose first element is true if the property is setable, and the second element is true if it is getable. Attempts to get a value from an inaccessible property and to set a value to a immutable property result in exceptions. aStub.getPropertyAttribute('Activity')[0] # `true` if setable aStub.getPropertyAttribute('Activity')[1] # `true` if getable

Getting LOGGER data

To get logged data from a LOGGERSTUB, use getData() method.

getData() method has three forms according to requested range and time resolution of the data:

- getData()

  Get the whole data.

- getData(`starttime` [, `endtime`])

  Get a slice of the data from `starttime` to `endtime`. If `endtime` is omitted, the slice includes the tail of the data.

- getData(`starttime`, `endtime`, `interval`)

  Get a slice of the data from `starttime` to `endtime`. This omits data points if a time interval between two datapoints is smaller than `interval`. This is not suitable for scientific data analysis, but optimized for speed.

getData() method returns a rank-2 (matrix) ARRAY object of NUMERICPYTHON module. The ARRAY has either one of the following forms: [[time value average min max] [time value average min max] ...] or [[time value] [time value] ...] The first five-tuple data format has five values in a single datapoint:

- time

  The time of the data point.

- value

  The value at the time point.

- average

  The time-weighted average of the value after the last data point to the time of this data point.

- min

  The minimum value after the last data point to the time of this data point.

- max

  The maximum value after the last data point to the time of this data point.

The two-tuple data format has only time and value.

To know the start time, the end time, and the size of the logged data before getting data, use getStartTime(), getEndTime(), and getSize() methods of LOGGERSTUB. getSize() returns the number of data points stored in the LOGGER.

Getting and changing logging interval

Logging interval of a LOGGER can be checked and changed by using getMinimumInterval() and setMinimumInterval(`interval`) methods of LOGGERSTUB. `interval` must be a zero or positive number in second. If `interval` is a non-zero positive number, the LOGGER skips logging if a simulation step occurs before `interval` second past the last logging time point. If `interval` is zero, the LOGGER logs at every simulation step.

An example usage of an ENTITYSTUB

The following example loads an EML file, and prints the value of ATP VARIABLE in SYSTEM `/CELL` every 10 seconds. If the value is below 1000, it stops the simulation.

```
loadModel( 'simple.eml' )

ATP = createEntityStub( 'Variable:/CELL:ATP' )

while 1:

    ATPValue = ATP[ 'Value' ]

    message( 'ATP value = %s' % ATPValue )

    if ATPValue <= 1000:
        break

    run( 10 )

message( 'Stopped at %s.' % getCurrentTime() )
```

6.4 Handling Data Files

6.4.1 About ECD file

ECELL SE uses ECD (E-Cell Data) file format to store simulation results. ECD is a plain text file, and easily handled by user-written and third-party data processing and plotting software such as gnuplot.

An ECD file can store a matrix of floating-point numbers.

ecell.ECDDataFile class can be used to save and load ECD files. A ECDDataFile object takes and returns a rank-2 ARRAY of NUMERICPYTHON. A 'rank-2' ARRAY is a matrix, which means that Numeric.rank(ARRAY) and len(Numeric.shape(ARRAY)) returns '2'.

6.4.2 Importing ECDDataFile class

To import the ECDDataFile class, import the whole ecell module,

```
import ecell
```

or import ecell.ECDDataFile module selectively.

```
import ecell.ECDDataFile
```

6.4.3 Saving and loading data

To save data to an ECD file, say, `datafile.ecd`, instantiate an ECDDataFile object and use save() method. import ecell aDataFile = ecell.ECDDataFile(DATA) aDataFile.save('datafile.ecd') here `DATA` is a rank-2 ARRAY of NUMERICPYTHON or an equivalent object. The data can also be set by using setData() method after the instantiation. If the data is already set, it is replaced. aDataFile.setData(DATA)

Loading the ECD file is also straightforward. aDataFile = ecell.ECDDataFile() aDataFile.load('datafile.ecd') DATA = aDataFile.getData() The getData() method extracts the data from the ECDDataFile object as an ARRAY.

6.4.4 ECD header information

In addition to the data itself, an ECD file can hold some information in its header.

- Data name

  The name of data. Setting a *FullPN* may be a good idea. Use setDataName(`name`) and getDataName() methods to set and get this field.

- Label

  This field is used to name axes of the data. Use setLabel(`labels`) and getLabel() methods. These methods takes and returns a PYTHON TUPLE, and stored in the file as a space-separated list. The default value of this field is: `( 't', 'value', 'avg', 'min', 'max' )`.

- Note

  This is a free-format field. This can be a multi-line or a single-line string. Use setNote(`note`) and getNote().

The header information is stored in the file like this.

```
#DATA:
#SIZE: 5 1010
#LABEL: t       value   avg     min     max
#NOTE:
#
#----------------------
0 0.1 0.1 0.1 0.1
...
```

Each line of the header is headed by a sharp (#) character. The `'#SIZE:'` line is automatically set when saved to show size of the data. This field is ignored in loading. The header ends with `'#----...'`.

6.4.5 Using ECD outside ECELL SE

For most cases NUMERICPYTHON will offer any necessary functionality for scientific data processing. However, using some external software can enhance the usability.

ECD files can be used as input to any software which supports white space-separated text format, and treats lines with heading sharps (#) as comments.

GNU gnuplot is a scientific presentation-quality plotting software with a sophisticated interactive command system. To plot an ECD file from gnuplot, just use `plot` command. For example, this draws a time-value 2D-graph: `gnuplot> ``\ ``plot 'datafile.ecd' with lines` Use `using` modifier to specify which column to use for the plotting. The following example makes a time-average 2D-plot. `gnuplot> ``\ ``plot 'datafile.ecd' using 1:3 with lines`

Another OpenSource software useful for data processing is GNU Octave. Loading an ECD from Octave is also simplest. `octave:1> load datafile.ecd` Now the data is stored in a matrix variable with the same name as the file without the extension (`datafile`). `octave:2> ``\ ``mean(datafile)` ‘‘ans =

5.0663 51.7158 51.7158 51.2396 52.2386‘‘

6.4.6 Binary format

Currently loading and saving of the binary file format is not supported. However, Numeric Python has an efficient, platform-dependent way of exporting and importing ARRAY data. See the Numeric Python manual.

6.5 Manipulating Model Files

This section describes how to create, modify, and read EML files with the EML module of the ECELL PYTHON library.

6.5.1 Importing EML module

To import the EML module, just import ecell module.

```
import ecell
```

And ecell.Eml class is made available.

6.6 Other Methods

6.6.1 Getting version numbers

getLibECSVersion() method of ecell.ecs module gives the version of the C++ backend library (libecs) as a string. getLibECSVersionInfo() method of the module gives the version as a PYTHON TUPLE. The TUPLE contains three numbers in this order: (`MAJOR_VERSION`, `MINOR_VERSION`, `MICRO_VERSION`)

6.6.2 DM loading-related methods

The search path of DM files can be specified and retrieved by using setDMSearchPath() and getDMSearchPath() methods. These methods gets and returns a colon (:) separated list of directory names. The search path can also be specified by using ECELL3_DM_PATH environment variable. See the previous section for more about DMsearch path.

A list of built-in and already loaded DM classes can be gotten with getDMInfo() method of ecell.ecs.Simulator class. The SIMULATOR instance is available to SESSION as `theSimulator` variable. The method returns a nested PYTHON TUPLE in the form of ((TYPE1, CLASSNAME1, PATH1), (TYPE2, CLASSNAME2, PATH2), ...). TYPE is one of `'Process'`, `'Variable'`, `'System'`, or `'Stepper'`. CLASSNAME is the class name of the DM. PATH is the directory from which the DM is loaded. PATH is an empty string (`''`) if it is a built-in class.

6.7 Advanced Topics

6.7.1 How ECELL3-SESSION runs

ECELL3-SESSION command runs on ECELL3-PYTHON interpreter command. ECELL3-PYTHON command is a thin wrapper to the PYTHON interpreter. ECELL3-PYTHON command simply invokes a PYTHON interpreter command specified at compile time. Before executing PYTHON, ECELL3-PYTHON sets some environment variables to ensure that it can find necessary ECELL PYTHON extension modules and the Standard DM Library. After processing the commandline options, ECELL3-SESSION command creates an ecell.ecs.Simulator object, and then instantiate a ecell.Session object for the simulator object.

Thus basically ECELL3-PYTHON is just a PYTHON interpreter, and frontend components of ECELL SE run on this command. To use the ECELL Python Library from ECELL3-PYTHON command, use

```
import ecell
```

statement from the prompt: `$ ``\ ``ecell3-python Python 2.2.2 (#1, Feb 24 2003, 19:13:11) [GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2 Type "help", "copyright", "credits" or "license" for more information. >>> ``\ ``import ecell` “>>> ‘‘ or, (on UNIX-like systems) execute a file starting with:

```
#!/usr/bin/env ecell3-python
import ecell
[...]
```

6.7.2 Getting information about execution environment

To get the current configuration of ECELL3-PYTHON command, invoke ECELL3-PYTHON command with a `-h` option. This will print values of some variables as well as usage of the command. `$ ``\ ``ecell3-python -h` “[...]

Configurations:

> PACKAGE = ecell VERSION = 3.2.0 PYTHON = /usr/bin/python PYTHONPATH = /usr/lib/python2.2/site-packages: DEBUGGER = gdb LD_LIBRARY_PATH = /usr/lib: prefix = /usr pythondir = /usr/lib/python2.2/site-packages ECELL3_DM_PATH =

[...] ‘‘ The ‘`PYTHON =`‘ line gives the path of the PYTHON interpreter to be used.

6.7.3 Debugging

To invoke ECELL3-PYTHON command in debugging mode, set ECELL_DEBUG environment variable. This runs the command on a debugger software. If found, GNU gdb is used as the debugger. ECELL_DEBUG can be used for any commands which run on ECELL3-PYTHON, including ECELL3-SESSION and GECELL. For example, to run ECELL3-SESSION in debug mode on the shell prompt: `$ ``\ ``ECELL_DEBUG=1 ecell3-session -f foo.eml gdb --command=/tmp/ecell3.0mlQyE /usr/bin/python GNU gdb Red Hat Linux (5.3post-0.20021129.18rh) Copyright 2003 Free Software Foundation, Inc. GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions. There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "i386-redhat-linux-gnu"... [New Thread 1074178112 (LWP 7327)] ecell3-session [E-Cell SE Version 3.2.0, on Python Version 2.2.2] Copyright (C) 1996-2014 Keio University. Send feedback to Koichi Takahashi <shafi@e-cell.org> <foo.eml, t=0>>> ``\ `` Program received signal SIGINT, Interrupt. [Switching to Thread 1074178112 (LWP 7327)] 0xffffe002 in ?? () (gdb)` It automatically runs the program with the commandline options with ‘`--command=`‘ option of gdb. The gdb prompt appears when the program crashes or interrupted by the user by pressing Ctrl C.

ECELL_DEBUG runs gdb, which is operates at the level of C++ code. For debugging of PYTHON layer scripts, see PYTHON Library Reference Manual for Python Debugger.

6.7.4 Profiling

It is possible to run ECELL3-PYTHON command in profiling mode, if the operating system has GNU sprof command, and its C library supports LD_PROFILE environmental variable. Currently it only supports per-shared object profiling. (See GNU C Library Reference Manual)

To run ECELL3-PYTHON in profiling mode, set ECELL_PROFILE environment variable to *SONAME* of the shared object. SONAME of a shared object file can be found by using objdump command, with, for example, `-p` option.

For example, the following commandline takes a performance profile of Libecs: `$ ''\ ''ECELL_PROFILE=libecs.so.2 ecell3-session [...]` After running, it creates a profiling data file with a filename `SONAME.profile` in the *current directory*. In this case, it is `libecs.so.2.profile`. The binary profiling data can be converted to a text format by using `sprof` command. For example: `$ ''\ ''sprof -p libecs.so.2 libecs.so.2.profile`

6.8 ECELL Python Library API

This section provides a list of some commonly used classes in ECELL Python library and their APIs.

6.8.1 SESSION Class API

Methods of SESSION class has the following five groups.

- Session methods
- Simulation methods
- Stepper methods
- Entity methods
- Logger methods

SESSION-CLASS-API OBJECTSTUB Classes API ————————-

There are three subclasses of OBJECTSTUB

- ENTITYSTUB
- STEPPERSTUB
- LOGGERSTUB

Some methods are common to these subclasses.

OBJECTSTUBS-API ECDDataFile Class API ————————

ECDDataFile class has the following set of methods.

ECDDATAFILE-API

CHAPTER 7

Creating New Object Classes

This section describes how to define your own object classes for use in the simulation.

7.1 About Dynamic Modules

Dynamic Module (DM) is a file containing an object class, especially C++ class, which can be loaded and instantiated by the application. APP uses this mechanism to provide users a way of defining and adding new classes to appear in simulation models without recompiling the whole system. Because the classes are defined in forms of native codes, this is the most efficient way of adding a new code or object class in terms of space and speed.

In APP, subclasses of PROCESS, VARIABLE, SYSTEM and STEPPER classes can be dynamically loaded by the system.

In addition to standard DMs distributed with APP, user-defined DM files can be created from C++ source code files ('.cpp' files) with the `ecell3-dmc` command. The compiled files usually take a form of shared library ('.so') files.

7.2 Defining a new class

A new object class can be defined by writing a C++ source code file with some special usage of C++ macros.

Here is a boilarplate template of a DM file, with which you should feel familiar if you have a C++ experience. Replace `DMTYPE`, `CLASSNAME`, and `BASECLASS` according to your case.

```
#include <libecs/libecs.hpp>
#include <libecs/.hpp>

USE_LIBECS;

LIBECS_DM_CLASS( ,  )
{
public:
    LIBECS_DM_OBJECT( ,  )
    {
      // ( Property definition of this class comes here. )
    }

    () {}// A constructor without an argument
    () {}// A destructor
};
```

```
LIBECS_DM_INIT( ,  );
```

7.2.1 DMTYPE, CLASSNAME and BASECLASS

First of all you have to decide basic attributes of the class you are going to define; such as a DM type (PROCESS, VARIABLE, SYSTEM, or STEPPER), a class name, and a base class.

- DMTYPE

  DMTYPE is one of DM base classes defined in APP PROCESS, STEPPER, VARIABLE, and SYSTEM.

- CLASSNAME

  CLASSNAME is a name of the object class.

  This must be a valid C++ class name, and should end with the DMTYPE name. For example, if you are going to define a new PROCESS class and want to name it Foo, the class name may look like FooProcess.

- BASECLASS

  The class your class inherits from.

  This may or may not be the same as the ‘‘DMTYPE ‘‘, depending on whether it is a direct descendant of the DM base

  class.

7.2.2 Filename

The name of the source file must be the same as the CLASSNAME with a trailing ‘.cpp’ suffix. For example, if the CLASSNAME is FooProcess, the file name must be FooProcess.cpp.

The source code can be divided into header and source files (such as FooProcess.hpp and FooProcess.cpp), but at least the LIBECS_DM_INIT macro must be placed in the source file of the class (FooProcess.cpp).

7.2.3 Include Files

At least the libecs header file (libecs/libecs.hpp) and a header file of the base class (such as libecs/.hpp) must be included in the head of the file.

7.2.4 DM Macros

You may notice that the template makes use of some special macros: USE_LIBECS, LIBECS_DM_CLASS, LIBECS_DM_OBJECT, and LIBECS_DM_INIT.

USE_LIBECS declares use of libecs library, which is the core library of APP, in this file after the line.

LIBECS_DM_CLASS

‘‘LIBECS_DM_OBJECT(,)‘‘ should be placed on the top of the class definition part

(immediately after ‘{‘ of the class). This macro declares that this is a DM class. This macro makes it dynamically instantiable, and automatically defines getClassName() method. Note that this macro specifies public: field inside, and thus anything comes after this is placed in public. For clarity it is a good idea to always write public: explicitly after this macro.

```
LIBECS_DM_OBJECT( DMTYPE, CLASSNAME )
        public:
```

‘‘**LIBECS_DM_INIT(,)**‘‘ exports the class `CLASSNAME` as a DM class of type

`DMTYPE`. This must come after the definition (not just a declaration) of the class to be exported with a `LIBECS_DM_OBJECT` call.

7.2.5 Constructor And Destructor

DM objects are always instantiated by calling the constructor with no argument. The destructor is defined virtual in the base class.

7.2.6 Types And Declarations

Basic types

The following four basic types are available to be used in your code if you included `libecs/libecs.hpp` header file and called the `USE_LIBECS` macro.

- `Real`

  A real number. Usually implemented as a double precision floating point number. It is a 64-bit float on Linux/IA32/gcc platform.

- `Integer`

  A signed integer number. This is a 64-bit `long int` on Linux/IA32/gcc.

- `UnsignedInteger`

  An unsigned integer number. This is a 64-bit `unsigned long int` on Linux/IA32/gcc.

- STRING

  A string equivalent to std::string class of the C++ standard library.

- POLYMORPH

  POLYMORPH is a sort of universal type (actually a class) which can *become* and *be made from* any of `Real`, `Integer`, `String`, and `PolymorphVector`, which is a mixed list of these three types of objects. See the next section for details.

These types are recommended to be used over other C++ standard types such as `double`, `int` and `char*`.

Pointer and reference types

For each types, the following typedefs are available.

- `TYPEPtr`

  Pointer type. (== `TYPE*`)

- `TYPECptr`

  Const pointer type. (== `const TYPE*`)

- `TYPERef`

  Reference type. (== `TYPE&`)

- `TYPECref`

  Const reference type. (== `const TYPE&`)

For example, `RealCref` is equivalent to write `const Real&`. Using these typedefs is recommended.

To declare a new type, use `DECLARE_TYPE` macro. For example,

```
DECLARE_TYPE( double, Real );
```

is called inside the system so that `RealCref` can be used as ‘‘const double&‘‘.

Similary, DECLARE_CLASS can be used to enable the typedefs for a class. Example:

```
DECLARE_CLASS( Process );
```

enables `ProcessCref` `ProcessPtr` etc.. Most classes defined in libecs have these typedefs.

Limits and other attributes of types

To get limits and precisions of these numeric types, use std::numeric_limits<> template class in the C++ standard library. For instance, to get a maximum value that can be represented by the `Real` type, use the template class like this:

```
#include <limits>
numeric_limits<Real>::max();
```

See the C++ standard library reference manual for more.

7.2.7 Polymorph class

A POLYMORPH object can be constructed from and converted to any of `Real`, `Integer`, `String`, types and POLYMORPHVECTOR class.

Construct a Polymorph

To construct a POLYMORPH object, simply call a constructor with a value:

```
Polymorph anIntegerPolymorph( 1 );
Polymorph aRealPolymorph( 3.1 );
Polymorph aStringPolymorph( "2.13e2" );
```

A POLYMORPH object can be constructed (or copied) from a POLYMORPH:

```
Polymorph aRealPolymorph2( aRealPolymorph );
```

Getting a value of a Polymorph

The value of the POLYMORPH objects can be retrieved in any type by using as<>() template method.

```
anIntegerPolymorph.as<Real>();    // == 1.0
aRealPolymorph.as<String>(); // == "3.1"
aStringPolymorph.as<Integer>();  // == 213

**Note**
```

```
If an overflow occurs when converting a very big ``Real`` value to
``Integer``, a ValueError exception?? is thrown. (NOT IMPLEMENTED
YET)
```

Examining and changing the type of Polymorph

getType(), changeType()

PolymorphVector

POLYMORPHVECTOR is a list of POLYMORPH objects.

7.2.8 Other C++ statements

The only limitation is the `DM_INIT` macro, which exports a class as a DM class, can appear only once in a compilation unit which forms a single shared library file.

Except for that, there is no limitation as far as the C++ compiler understands it. There can be any C++ statements inside and outside of the class definition including; other class definitions, nested classes, typedefs, static functions, namespaces, and even template<>.

Be careful, however, about namespace corruptions. You may want to use private C++ namespaces and static functiont when a class or a function declared outside the DM class is needed.

7.3 PropertySlot

7.3.1 What is PropertySlot

PROPERTYSLOT is a pair of methods to access (get) and mutate (set) an *object property*, associated with the name of the property. Values of the object property can either be stored in a member variable of the object, or dynamically created when the methods are called.

All of the four DM base classes, PROCESS, VARIABLE, SYSTEM and STEPPER can have a set of PROPERTYSLOTs, or object *properties*. In other words, these classes inherit PROPERTYINTERFACE common base class.

What is PropertySlot for?

PROPERTYSLOTs can be used from model files (such as EM files) as a means of giving parameter values to each objects in the simulation model (such as ENTITY and STEPPER objects). It can also be ways of dynamic communications between objects during the simulation.

Type of PropertySlot

A type of a PROPERTYSLOT is any one of these four types:

- `Real`
- `Integer`
- `String`
- `Polymorph`

7.3.2 How to define a PropertySlot

To define a PROPERTYSLOT on an object class, you have to:

1. Define set and/or get method(s).
2. If necessary, define a member variable to store the property value.
3. Register the method(s) as a PROPERTYSLOT.

Set method and get method

A PROPERTYSLOT is a pair of object methods, *set method* and *get method*, associated with a property name. Either one of the methods can be ommited. If there is a set method defined for a PROPERTYSLOT, the PROPERTYSLOT is said to be *setable*. If there is a get method, it is *getable.*

A set method must have the following signature to be recognized by the system.

```
void CLASS::* ( const T&)
```

And a get method must look like this:

```
const T CLASS::* ( void ) const
```

where `T` is a property type and `CLASS` is the object class that the PROPERTYSLOT belongs to.

Don't worry, you don't need to memorize these prototypes. The following four macoros can be used to declare and define set/get methods of a specific type and a property name.

- `SET_METHOD( , )`
  - *Expansion:*

```
void set( const &value )
```

  - *Usage:* `SET_METHOD` macro is used to declare or define a property set method, of which the property type is `TYPE` and the property name is `NAME`, in a class definition. The given property value is available as the `value` argument variable.
  - *Example:*

    This code:

```
class FooProcess
{
    SET_METHOD( Real, Flux )
    {
        theFlux = value;
    }

    Real theFlux;
};
```

    will expand to the following C++ program.

```
class FooProcess
{
    void setFlux( const Real& value )
    {
        theFlux = value;
    }
```

```
        Real theFlux;
    };
```

In this example, the given property value is stored in the member variable `theFlux`.

- `GET_METHOD( , )`
  - *Expansion:*

```
const  get() const
```

  - *Usage:* `GET_METHOD` macro is used to declare or define a property get method, of which the property type is `TYPE` and the property name is `NAME`, in a class definition. Definition of the method must return the value of the property as a `TYPE` object.
  - *Example:*

    This code:

```
class FooProcess
{
    GET_METHOD( Real, Flux )
    {
        return theFlux;
    }

    Real theFlux;
};
```

    will expand to the following C++ program.

```
class FooProcess
{
    const Real getFlux() const
    {
        return theFlux;
    }

    Real theFlux;
};
```

- `SET_METHOD_DEF( , , )`
  - *Expansion:*

```
void ::set( const &value )
```

  - *Usage:* `SET_METHOD_DEF` macro is used to define a property set method outside class scope.
  - *Example:*

    `SET_METHOD_DEF` macro is usually used in conjunction with `SET_METHOD` macro. For instance, the following code declares a property setter method with `SET_METHOD` in the class definition, and later defines the actual body of the method using `SET_METHOD_DEF`.

```
class FooProcess
{
    virtual SET_METHOD( Real, Flux );

    Real theFlux;
};
```

```
SET_METHOD_DEF( Real, Flux, FooProcess )
{
    theFlux = value;
}
```

The definition part will expand to the following C++ program.

```
void FooProcess::setFlux( const Real& value )
{
    theFlux = value;
}
```

- GET_METHOD_DEF(, ,)
  - *Expansion:*

```
const  ::get() const
```

  - *Usage:* GET_METHOD_DEF macro is used to define a property get method outside class scope.
  - *Example:* See the example of SET_METHOD_DEF above.

If the property is both setable and getable, and is simply stored in a member variable, the following macro can be used.

```
SIMPLE_SET_GET_METHOD( ,  )
```

This assumes there is a variable with the same name as the property name (NAME), and expands to a code that is equivalent to:

```
SET_METHOD( ,  )
{
   = value;
}

GET_METHOD( ,  )
{
  return ;
}
```

Registering PropertySlots

To register a PROPERTYSLOT on a class, one of these macros in the LIBECS_DM_OBJECT macro of the target class:

- PROPERTYSLOT_SET_GET(,)

  Use this if the property is both setable and getable, which means that the class defines both set method and get method.

  For example, to define a property 'Flux' of type Real on the FooProcess class, write like this in the public area of the class definition:

```
public:

  LIBECS_DM_OBJECT( ,  )
  {
    PROPERTYSLOT_SET_GET( ,  );
  }
```

This registers these methods:

```
void FooProcess::setFlux( const Real& );
```

and

```
const Real FooProcess::getFlux() const;
```

as the set and get methods of 'Flux' property of the class FooProcess, respectively. Signatures of the methods must match with the prototypes defined in the previous section. LIBECS_DM_OBJECT can have any number of properties. It can also be empty.

- PROPERTYSLOT_SET(,)

  This is almost the same as PROPERTYSLOT_SET_GET, but this does not register get method. Use this if only a set method is available.

- PROPERTYSLOT_GET(,)

  This is almost the same as PROPERTYSLOT_SET_GET, but this does not register set method. Use this if only a get method is available.

- PROPERTYSLOT(, , ,)

  If the name of either get or set method is different from the default format (set''NAME''() or getNAME()), then use this macro with explicitly specifying the pointers to the methods.

  For example, the following use of the macro registers setFlux2() and anotherGetMethod() methods of Flux property of the class FooProcess:

```
PROPERTYSLOT( Flux, Real,
              &FooProcess::setFlux2,
              &FooProcess::anotherGetMethod );
```

If more than one PROPERTYSLOTs with the same name are created on an object, the last is taken.

Load / save methods

In addition to set and get methods, load and save methods can be defined. Load methods are called when the model is loaded from the model file. Similarly, save methods are called when the state of the model is saved to a file by saveModel() method of the simulator.

Unless otherwise specified, load and save methods default to set and get methods. This default definition can be changed by using the following some macros.

- PROPERTYSLOT_LOAD_SAVE(, , , , ,)

  This macros is the most generic way to set the property methods; all of set method, get method, load method ans save method can be specified independently. If the LOAD_METHOD is NOMETHOD, it is said to be not *loadable*, and it is not *savable* if SAVE_METHOD is NOMETHOD.

- PROPERTYSLOT_NO_LOAD_SAVE(, , ,)

  Usage of this macro is the same as PROPERTYSLOT in the previous section, but this sets both LOAD_METHOD and SAVE_METHOD to NOMETHOD.

  That is, this macro is equivalent to writing:

- PROPERTYSLOT_SET_GET_NO_LOAD_SAVE(, , ,)

  PROPERTYSLOT_SET_NO_LOAD_SAVE(, ,)

  PROPERTYSLOT_GET_NO_LOAD_SAVE(, ,)

Usage of these macros are the same as: PROPERTYSLOT_SET_GET, PROPERTYSLOT_SET, and PROPERTYSLOT_GET, except that load and save methods are not set instead of default to set and get methods.

Inheriting properties of base class

In most cases you may also want to use properties of base class. To inherit the baseclass properties, use INHERIT_PROPERTIES( ) macro. This macro is usually placed before any property definition macros (such as PROPERTY_SET_GET()).

```
LIBECS_DM_OBJECT( ,  )
{
    INHERIT_PROPERTIES(  );

    PROPERTYSLOT_SET_GET( ,  );
}
```

Here PROPERTY_BASECLASS is usually the same as BASECLASS. An exception is when the BASECLASS does not make use of LIBECS_DM_OBJECT() macro. In this case, choose the nearest baseclass in the class hierarachy that uses LIBECS_DM_OBJECT() for PROPERTY_BASECLASS.

7.3.3 Using PropertySlots In Simulation

(1) Static direct access (using native C++ method) bypassing the PROPERTYSLOT, (2) dynamically-bound access via a PROPERTYSLOT object, (3) dynamically-bound access via PROPERTYINTERFACE.

7.4 Defining a new Process class

To define a new PROCESS class, at least the following two methods need to be defined.

- initialize()
- fire()

initialize() is called when the simulation state needs to be reset. Note that reset can happen anytime during the session, not just at the beginning; especially when the reintegration of the state is requested. fire() is called when the reaction takes place. You have to update the VARIABLEs referred to by your PROCESS according to VARIABLEREFERENCE.

The PROCESS's VARIABLEREFERENCEs are stored in theVariableReferenceVector member variable, sorted by coefficient. Hence references that have negative coefficients are followed by those of zero coefficients, and so by those of positive coefficients. You can get the offset from which the "zero" or positive references start through getZeroVariableReferenceOffset() or getPositiveVariableReferenceOffset(). If you want to look up for a specific VARIABLEREFERENCE by name, use getVariableReference().

```
#include <libecs.hpp>
#include <Process.hpp>

USE_LIBECS;

LIBECS_DM_CLASS( SimpleProcess, Process )
{
public:
    LIBECS_DM_OBJECT( SimpleFluxProcess, Process )
    {
        PROPERTYSLOT_SET_GET( Real, k );
```

```
    }

    SimpleProcess(): k( 0.0 )
    {
    }

    SIMPLE_SET_GET_METHOD( Real, k );

    virtual void initialize()
    {
        Process::initialize();
        S0 = getVariableReference( "S0" );
    }

    virtual void fire()
    {
        // concentration gets reverted to the number of molecules
        // according to the volume of the System where the Process belongs.
        setFlux( k * S0.getMolarConc() * getSuperSystem()->getSize() * N_A );
    }

protected:
    Real k;
    VariableReference const& S0;
};

LIBECS_DM_INIT( SimpleProcess, Process );
```

7.5 Defining a new Stepper class

7.6 Defining a new Variable class

7.7 Defining a new System class

CHAPTER 8

Standard Dynamic Module Library

This chapter overviews:

An incomplete list of classes available as the Standard Dynamic Module Library, and, Some usage the classes in the Standard Dynamic Module Library. This chapter briefly describes the Standard Dynamic Module Library distributed with APP. If the system is installed correctly, the classes provided by the library can be used without any special procedure.

This chapter is not meant to be a complete reference. To know more about the classes defined in the library, see the E-Cell3 Standard Dynamic Module Library Reference Manual (under preparation).

8.1 Steppers

There are three direct sub-classes of STEPPER: DifferentialStepper, DiscreteEventStepper, DiscreteTimeStepper

8.1.1 DifferentialSteppers

General-purpose DifferentialStepper classes

The following STEPPER classes implement general-purpose ordinary differential equation solvers. Basically these classes must work well with any simple continuous PROCESS classes.

- ODE45Stepper

  This STEPPER implements Dormand-Prince 5(4)7M algorithm for ODE systems.

  In most cases this STEPPER is the best general purpose solver for ODE models.

- ODE23Stepper

  This STEPPER implements Fehlberg 2(3) algorithm for ODE systems.

  Try this STEPPER if other part of the model has smaller timescales. This STEPPER can be used for a moderately stiff systems of differential equations.

- FixedODE1Stepper

  A DifferentialStepper without adaptive stepsizing mechanism. The solution of this STEPPER is first order.

  This stepper calls process() method of each PROCESS just once in a single step.

  Although this STEPPER is not suitable for high-accuracy solution of smooth continuous systems of differential equations, its simplicity of the algorithm is sometimes useful.

S-System and GMA Steppers

FIXME: need description here.

8.1.2 DiscreteEventSteppers

- DiscreteEventStepper

  This STEPPER is used to conduct discrete event simulations. This STEPPER should be used in combination with subclasses of DiscreteEventProcess.

  This STEPPER uses its PROCESS objects as event generators. The procedure of this STEPPER for initialize() method is like this:

  1. updateStepInterval() method of its all DiscreteEventProcess objects.
  2. Find a PROCESS with the least *scheduled time* (top process). The scheduled time is calculated as: (current time) + (StepInterval of the process).
  3. Reschedule itself to the scheduled time of the top process.

  step() method of this STEPPER is as follows:

  1. Call process() method of the current top process.
  2. Calls updateStepInterval() method of the top process and all *dependent processes* of the top process, and update scheduled times for those processes to find the new top process.
  3. Lastly the STEPPER reschedule itself to the scheduled time of the new top process.

  The procedure for interrupt() method of this class is the same as that for initialize(). FIXME: need to explain about TimeScale property.

- NRStepper

  This is an alias to the DiscreteEventStepper. This class can be used as an implementation of Gillespie-Gibson algorithm.

  To conduct the Gillespie-Gibson simulation, use this class of STEPPER in combination with GillespieProcess class. GillespieProcess is a subclass of DiscreteEventProcess.

8.1.3 DiscreteTimeStepper

- DiscreteTimeStepper

  This STEPPER steps with a fixed interval. For example, StepInterval property of this STEPPER is set to `0.1`, this STEPPER steps every 0.1 seconds.

  When this STEPPER steps, it calls process() of all of its PROCESS instances. To change this behavior, create a subclass.

  This STEPPER ignores incoming interruptions from other STEPPERs.

8.1.4 PassiveStepper

- PassiveStepper

  This STEPPER never steps spontaneously (step interval = infinity). Instead, this STEPPER steps upon interruption. In other words, this STEPPER steps everytime immediately after a dependent STEPPER steps.

When this STEPPER steps, it calls process() of all of its PROCESS instances. To change this behavior, create a subclass.

8.2 Process classes

8.2.1 Continuous Process classes

Differential equation-based Process classes

The following PROCESS classes are straightforward implementations of differential equations, and can be used with the general-purpose DifferentialSteppers such as ODE45Stepper, ODE23Stepper, and FixedODE1Stepper.

In the current version, most of the classes represent certain reaction rate equations. Of course it is not limited to chemical and biochemical simulations.

- CatalyzedMassActionFluxProcess
- DecayFluxProcess
- IsoUniUniFluxProcess
- MassActionProcess
- MichaelisUniUniProcess
- MichaelisUniUniReversibleProcess
- OrderedBiBiFluxProcess
- OrderedBiUniFluxProcess
- OrderedUniBiFluxProcess
- PingPongBiBiFluxProcess
- RandomBiBiFluxProcess
- RandomBiUniFluxProcess
- RandomUniBiFluxProcess

Other continuous Process classes

- PythonFluxProcess
- SSystemProcess

8.2.2 Discrete Process classes

- GammaProcess

  Under development.

- GillespieProcess

  This PROCESS must be used with a Gillespie-type STEPPER, such as NRStepper.

- RapidEquilibriumProcess

8.2.3 Other Process classes

- PythonProcess

8.3 Variable classes

- Variable

  A standard class to represent a state variable.

CHAPTER 9

Simulation Mechanism of E-Cell

This chapter reveals how APP represents cell models internally, and what happens when the simulation is executed inside the system.

CHAPTER 10

Empy Module Manual

```
   empy

Summary

   A templating system for Python.

Overview

   EmPy is a system for embedding Python expressions and statements in template text; it
   takes an EmPy source file, processes it, and produces output. This is accomplished via
   expansions, which are special signals to the EmPy system and are set off by a special
   prefix (by default the at sign, @). EmPy can expand arbitrary Python expressions and
   statements in this way, as well as a variety of special forms. Textual data not explicitly
   delimited in this way is sent unaffected to the output, allowing Python to be used in
   effect as a markup language. Also supported are "hook" callbacks, recording and playback
   via diversions, and dynamic, chainable filters. The system is highly configurable via
   command line options and embedded commands.

   Expressions are embedded in text with the @(...) notation; variations include conditional
   expressions with @(...?...:...) and the ability to handle thrown exceptions with
   @(...$...). As a shortcut, simple variables and expressions can be abbreviated as
   @variable, @object.attribute, @function(arguments), @sequence [2][index]
   , and combinations. Full-fledged statements are embedded with @{...}. Forms of
   conditional, repeated, and recallable expansion are available via @[...]. A @ followed by
   a whitespace character (including a newline) expands to nothing, allowing string
   concatenations and line continuations. Comments are indicated with @# and consume the rest
   of the line, up to and including the trailing newline. @% indicate "significators," which
   are special forms of variable assignment intended to specify per-file identification
   information in a format which is easy to parse externally. Escape sequences analogous to
   those in C can be specified with @\..., and finally a @@ sequence expands to a single
   literal at sign.

Getting the software

   The current version of empy is 2.3.

   The latest version of the software is available in a tarball here: [3]
   http://www.alcyone.com/pyos/empy/empy-latest.tar.gz.

   The official URL for this Web site is [4]http://www.alcyone.com/pyos/empy/.

Requirements
```

```
   EmPy should work with any version of Python from 1.5.x onward. It has been tested with all
   major versions of CPython from 1.5 up, and Jython from 2.0 up. The included test script is
   intended to run on UNIX-like systems with a Bourne shell.

License

   This code is released under the [5]GPL.

Mailing lists

   There are two EmPy related mailing lists available. The first is a receive-only, very low
   volume list for important announcements (including releases). To subscribe, send an email
   to [6]empy-announce-list-subscribe@alcyone.com.

   The second is a general discussion list for topics related to EmPy, and is open for
   everyone to contribute; announcements related to EmPy will also be made on this list. The
   author of EmPy (and any future developers) will also be on the list, so it can be used not
   only to discuss EmPy features with other users, but also to ask questions of the
   author(s). To subscribe, send an email to [7]empy-list-subscribe@alcyone.com.

Basics

   EmPy is intended for embedding Python code in otherwise unprocessed text. Source files are
   processed, and the results are written to an output file. Normal text is sent to the
   output unchanged, but markups are processed, expanded to their results, and then written
   to the output file as strings (that is, with the str function, not repr). The act of
   processing EmPy source and handling markups is called "expansion."

   Code that is processed is executed exactly as if it were entered into the Python
   interpreter; that is, it is executed with the equivalent of eval (for expressions) and
   exec (for statements). For instance, inside an expression, abc represents the name abc,
   not the string "abc", just as it would in normal Python code.

   By default the embedding token prefix is the at sign (@), which appears neither in valid
   Python code nor commonly in English text; it can be overridden with the -p option (or with
   the empy.setPrefix function). The token prefix indicates to the EmPy interpreter that a
   special sequence follows and should be processed rather than sent to the output untouched
   (to indicate a literal at sign, it can be doubled as in @@).

   When the interpreter starts processing its target file, no modules are imported by
   default, save the empy pseudomodule (see below), which is placed in the globals; the empy
   pseuodmodule is associated with a particular interpreter; it is important that it not be
   removed from that interpreter's globals, nor that it be shared with other interpreters
   running concurrently. The globals are not cleared or reset in any way. It is perfectly
   legal to set variables or explicitly import modules and then use them in later markups,
   e.g., @{import time} ... @time.time(). Scoping rules are as in normal Python, although all
   defined variables and objects are taken to be in the global namespace.

   Activities you would like to be done before any processing of the main EmPy file can be
   specified with the -I, -D, -E, -F, and -P options. -I imports modules, -D executes a
   Python variable assignment, -E executes an arbitrary Python (not EmPy) statement, -F
   executes a Python (not EmPy) file, and -P processes an EmPy (not Python) file. These
   operations are done in the order they appear on the command line; any number of each
   (including, of course, zero) can be used.

Expansions

   The following markups are supported. For concreteness below, @ is taken for the sake of
```

```
argument to be the prefix character, although this can be changed.

@# COMMENT NEWLINE
       A comment. Comments, including the trailing newline, are stripped out completely.
       Comments should only be present outside of expansions. The comment itself is not
       processed in any way: It is completely discarded. This allows @# comments to be
       used to disable markups. Note: As special support for "bangpaths" in UNIX like
       operating systems, if the first line of a file (or indeed any context) begins with
       #!, and the interpreter has a processBangpaths option set to true (default), it is
       treated as a @# comment. A #! sequence appearing anywhere else will be handled
       literally and unaltered in the expansion. Example:

       @# This line is a comment.
       @# This will NOT be expanded: @x.

@ WHITESPACE
       A @ followed by one whitespace character (a space, horizontal tab, vertical tab,
       carriage return, or newline) is expanded to nothing; it serves as a way to
       explicitly separate two elements which might otherwise be interpreted as being the
       same symbol (such as @name@ s to mean '@(name)s'; see below). Also, since a newline
       qualifies as whitespace here, the lone @ at the end of a line represents a line
       continuation, similar to the backslash in other languages. Coupled with statement
       expansion below, spurious newlines can be eliminated in statement expansions by use
       of the @{...}@ construct. Example:

       This will appear as one word: salt@ water.
       This is a line continuation; @
       this text will appear on the same line.

@\ ESCAPE_CODE
       An escape code. Escape codes in EmPy are similar to C-style escape codes, although
       they all begin with the prefix character. Valid escape codes include:

     @\0
              NUL, null

     @\a
              BEL, bell

     @\b
              BS, backspace

     @\d
              three-digital decimal code DDD

     @\e
              ESC, escape

     @\f
              FF, form feed

     @\h
              DEL, delete

     @\n
              LF, linefeed character, newline

     @\oOOO
```

```
               three-digit octal code OOO

      @\qQQQQ
               four-digit quaternary code QQQQ

      @\r
               CR, carriage return

      @\s
               SP, space

      @\t
               HT, horizontal tab

      @\v
               VT, vertical tab

      @\xHH
               two-digit hexadecimal code HH

      @\z
               EOT, end of transmission

      @^X
               the control character ^X

        Unlike in C-style escape codes, escape codes taking some number of digits afterward
        always take the same number to prevent ambiguities. Furthermore, unknown escape
        codes are treated as parse errors to discourage potential subtle mistakes. Unlike
        in C, to represent an octal value, one must use @\o.... Example:

        This embeds a newline.@\nThis is on the following line.
        This beeps!@\a
        There is a tab here:@\tSee?
        This is the character with octal code 141: @\o141.

  @@
        A literal at sign (@). To embed two adjacent at signs, use @@@@, and so on. Any
        literal at sign that you wish to appear in your text must be written this way, so
        that it will not be processed by the system. Note: If a prefix other than @ has
        been chosen via the command line option, one expresses that literal prefix by
        doubling it, not by appending a @. Example:

        The prefix character is @@.
        To get the expansion of x you would write @@x.

  @), @], @}
        These expand to literal close parentheses, close brackets, and close braces,
        respectively; these are included for completeness and explicitness only. Example:

        This is a close parenthesis: @).

  @( EXPRESSION )
        Evaluate an expression, and replace the tokens with the string (via a call to str)
        representation evaluation of that expression. Whitespace immediately inside the
        parentheses is ignored; @( expression ) is equivalent to @(expression). If the
        expression evaluates to None, nothing is expanded in its place; this allows
        function calls that depend on side effects (such as printing) to be called as
```

```
        expressions. (If you really do want a None to appear in the output, then use the
        Python string "None".) Example:

        2 + 2 is @(2 + 2).
        4 squared is @(4**2).
        The value of the variable x is @(x).
        This will be blank: @(None).

@( TEST ? THEN (: ELSE)_opt ($ CATCH)_opt )
        A special form of expression evaluation representing conditional and protected
        evaluation. Evaluate the "test" expression; if it evaluates to true (in the
        Pythonic sense), then evaluate the "then" section as an expression and expand with
        the str of that result. If false, then the "else" section is evaluated and
        similarly expanded. The "else" section is optional and, if omitted, is equivalent
        to None (that is, no expansion will take place).

        If the "catch" section is present, then if any of the prior expressions raises an
        exception when evaluated, the expansion will be substituted with the evaluation of
        the catch expression. (If the "catch" expression itself raises, then that exception
        will be propagated normally.) The catch section is optional and, if omitted, is
        equivalent to None (that is, no expansion will take place). An exception (cough) to
        this is if one of these first expressions raises a SyntaxError; in that case the
        protected evaluation lets the error through without evaluating the "catch"
        expression. The intent of this construct is to catch runtime errors, and if there
        is actually a syntax error in the "try" code, that is a problem that should
        probably be diagnosed rather than hidden. Example:

        What is x? x is @(x ? "true" : "false").
        Pluralization: How many words? @x word@(x != 1 ? 's').
        The value of foo is @(foo $ "undefined").
        The square root of -1 is @(math.sqrt(-1) $ "not real").

@ SIMPLE_EXPRESSION
        As a shortcut for the @(...) notation, the parentheses can be omitted if it is
        followed by a "simple expression." A simple expression consists of a name followed
        by a series of function applications, array subscriptions, or attribute
        resolutions, with no intervening whitespace. For example:

        + a name, possibly with qualifying attributes (e.g., @value, @os.environ).
        + a straightforward function call (e.g., @min(2, 3), @time.ctime()), with no space
          between the function name and the open parenthesis.
        + an array subscription (e.g., '@array[8][index]', '@os.environ[9][name]', with no
          space between the name and the open bracket.
        + any combination of the above (e.g., '@function(args).attr[10][sub].other[11][i]
          (foo)').

        In essence, simple expressions are expressions that can be written ambiguously from
        text, without intervening space. Note that trailing dots are not considered part of
        the expansion (e.g., @x. is equivalent to @(x)., not @(x.), which would be illegal
        anyway). Also, whitespace is allowed within parentheses or brackets since it is
        unambiguous , but not between identifiers and parentheses, brackets, or dots.
        Explicit @(...) notation can be used instead of the abbreviation when concatenation
        is what one really wants (e.g., @(word)s for simple pluralization of the contents
        of the variable word). As above, if the expression evaluates to the None object,
        nothing is expanded. Example:

        The value of x is @x.
        The ith value of a is @a[i].
```

```
        The result of calling f with q is @f(q).
        The attribute a of x is @x.a.
        The current time is @time.ctime(time.time()).
        The current year is @time.localtime(time.time())[0].
        These are the same: @min(2,3) and @min(2, 3).
        But these are not the same: @min(2, 3) vs. @min (2, 3).
        The plural of @name is @(name)s, or @name@ s.

@` EXPRESSION `
        Evaluate a expression, and replace the tokens with the repr (instead of the str
        which is the default) of the evaluation of that expression. This expansion is
        primarily intended for debugging and is unlikely to be useful in actual practice.
        That is, a @`...` is identical to @(repr(...)). Example:

        The repr of the value of x is @`x`.
        This print the Python repr of a module: @`time`.
        This actually does print None: @`None`.

@: EXPRESSION : DUMMY :
        Evaluate an expression and then expand to a @:, the original expression, a :, the
        evaluation of the expression, and then a :. The current contents of the dummy area
        are ignored in the new expansion. In this sense it is self-evaluating; the syntax
        is available for use in situations where the same text will be sent through the
        EmPy processor multiple times. Example:

        This construct allows self-evaluation:
        @:2 + 2:this will get replaced with 4:

@[ noop : IGNORED ]
        The material contained within the substitution is completely ignored. The
        substiution does not expand to anything, and indeed expansion contained within the
        ignored block are not expanded. This is included simply for completeness, and can
        served as a block comment. Example:

        @[noop:
        All this stuff would appear here
        if it weren't for the noop.
        @{
        while 1:
            print "Testing"
        }@
        ]

@[ if EXPRESSION : CODE ]
        Evaluate the Python test expression; if it evaluates to true, then expand the
        following code through the EmPy system (which can contain markups), otherwise,
        expand to nothing. Example:

        @[if x > 0:@x is positive.]
        @# If you want to embed unbalanced right brackets:
        @[if showPrompt:@\x5dINIT HELLO]

@[ while EXPRESSION : CODE ]
        Evaluate the Python expression; if it evaluates to true, then expand the code and
        repeat; otherwise stop expanding. Example:

        @[while i < 10:@ i is @i.@\n]
```

```
@[ for NAME in EXPRESSION : CODE ]
       Evaluate the Python expression and treat it as a sequence; iterate over the
       sequence, assigning each element to the provided name in the globals, and expanding
       the given code each time. Example:

       @[for i in range(5):@ The cube of @i is @(i**3).@\n]

@[ macro SIGNATURE : CODE ]
       Define a "macro," which is a function-like object that causes an expansion whenever
       it is called. The signature defines the name of the function and its parameter
       list, if any -- just like normal Python functions, macro signatures can include
       optional arguments, keyword arguments, etc. When defined, calling the macro results
       in the given code to be expanded, with the function arguments involved as the
       locals dictionary in the expansion. Additionally, the doc string of the function
       object that is created corresponds to the expansion. Example:

       @[macro f(n):@ @[for i in range(n):@ @i**2 is @(i**2)@\n]]

@{ STATEMENTS }
       Execute a (potentially compound) statement; statements have no return value, so the
       expansion is not replaced with anything. Multiple statements can either be
       separated on different lines, or with semicolons; indentation is significant, just
       as in normal Python code. Statements, however, can have side effects, including
       printing; output to sys.stdout (explicitly or via a print statement) is collected
       by the interpreter and sent to the output. The usual Python indentation rules must
       be followed, although if the statement consists of only one statement, leading and
       trailing whitespace is ignored (e.g., @{ print time.time() } is equivalent to
       @{print time.time()}). Example:

       @{x = 123}
       @{a = 1; b = 2}
       @{print time.time()}
       @# Note that extra newlines will appear above because of the
       @# newlines trailing the close braces.  To suppress them
       @# use a @ before the newline:
       @{
       for i in range(10):
           print "i is %d" % i
       }@
       @{print "Welcome to EmPy."}@

@% KEY (WHITESPACE VALUE)_opt NEWLINE
       Declare a significator. Significators consume the whole line (including the
       trailing newline), and consist of a key string containing no whitespace, and than
       optional value prefixed by whitespace. The key may not start with or contain
       internal whitespace, but the value may; preceding or following whitespace in the
       value is stripped. Significators are totally optional, and are intended to be used
       for easy external (that is, outside of EmPy) identification when used in large
       scale environments with many EmPy files to be processed. The purpose of
       significators is to provide identification information about each file in a
       special, easy-to-parse form so that external programs can process the significators
       and build databases, independently of EmPy. Inside of EmPy, when a significator is
       encountered, its key, value pair is translated into a simple assignment of the form
       __KEY__ = VALUE , where "__KEY__" is the key string with two underscores on either
       side and "VALUE" is a Python expression. Example:

       @%title     "Nobody knows the trouble I've seen"
       @%keywords  ['nobody', 'knows', 'trouble', 'seen']
```

```
          @%copyright [2000, 2001, 2002]

Substitutions

   Supported are conditional and repeated substitutions, which involve testing or iterating
   over Python expressions and then possibly expanding EmPy code. These different from normal
   Python if, for, and while statements since the result is an EmPy expansion, rather than
   the execution of a Python statement; the EmPy expansion may, of course, contain further
   expansions. This is useful for in-place conditional or repeated expansion of similar text;
   as with all expansions, markups contained within the EmPy code are processed. The simplest
   form would consist something like:
        @[if x != 0:x is @x]

   This will expand x is @x if x is greater than zero. Note that all characters, including
   whitespace and newlines, after the colon and before the close bracket are considered part
   of the code to be expanded; to put a space in there for readability, you can use the
   prefix and a whitespace character:
        @[if x != 0:@ x is @x]

   Iteration via while is also possible:
        @{i = 0}@[while i < 10:@ i is @i@\n@{i = i + 1}]

   This is a rather contrived example which iterates i from 0 to 9 and then prints "i is
   (value)" for each iteration.

   A more practical example can be demonstrated with the for notation:
        <table>@[for x in elements:@ <tr><td>@x</td></tr>]</table>

   This EmPy fragment would format the contents of elements into an HTML table, with one
   element per row.

   The macro substitution doesn't get replaced with anything, but instead defines a "macro,"
   or recallable expansion, which looks and behaves like a function. When called, it expands
   its contents. The arguments to the function -- which can be defined with optional,
   remaining, and keyword arguments, just like any Python function -- can be referenced in
   the expansion as local variables. For concreteness, the doc string of the macro function
   is the original expansion. An macro substitution of the form @[macro SIGNATURE:CODE] is
   equivalent to the following Python code:
        def SIGNATURE:
            repr(CODE) # so it is a doc string
            empy.string(repr(CODE), '<macro>', locals())

   This can be used to defer the expansion of something to a later time:
        @[macro header(title='None'):<head><title>@title</title></head>]

   Note that all text up to the trailing bracket is considered part of the EmPy code to be
   expanded. If one wishes a stray trailing brackets to appear in the code, one can use an
   escape code to indicate it, such as @\x5d. Matching open and close bracket pairs do not
   need to be escaped, for either bracket pairs in an expansion or even for further
   substitutions:
        @[if something:@ This is an unbalanced close bracket: @\x5d]
        @[if something:@ This is a balanced bracket pair: [word]]
        @[if something:@ @[if somethingElse:@ This is nested.]]

Significators

   Significators are intended to represent special assignment in a form that is easy to
   externally parse. For instance, if one has a system that contains many EmPy files, each of
```

```
   which has its own title, one could use a title significator in each file and use a simple
   regular expression to find this significator in each file and organize a database of the
   EmPy files to be built. This is an easier proposition than, for instance, attempting to
   grep for a normal Python assignment (inside a @{...} expansion) of the desired variable.

   Significators look like the following:
        @%KEY VALUE

   including the trailing newline, where "key" is a name and "value" is a Python expression,
   and are separated by any whitespace. This is equivalent to the following Python code:
        __KEY__ = VALUE

   That is to say, a significator key translates to a Python variable consisting of that key
   surrounded by double underscores on either side. The value may contain spaces, but the key
   may not. So:
        @%title "All Roads Lead to Rome"

   translates to the Python code:
        __title__ = "All Roads Lead to Rome"

   but obviously in a way that easier to detect externally than if this Python code were to
   appear somewhere in an expansion. Since significator keys are surrounded by double
   underscores, significator keys can be any sequence of alphanumeric and underscore
   characters; choosing 123 is perfectly valid for a significator (although straight), since
   it maps to the name __123__ which is a legal Python identifier.

   Note the value can be any Python expression. The value can be omitted; if missing, it is
   treated as None.

   Significators are completely optional; it is totally legal for a EmPy file or files to be
   processed without containing any significators.

   A regular expression string designed to match significators (with the default prefix) is
   available as empy.SIGNIFICATOR_RE_STRING, and also is a toplevel definition in the em
   module itself.

Diversions

   EmPy supports an extended form of m4-style diversions, which are a mechanism for deferring
   and recalling output on demand. Multiple "streams" of output can be diverted and
   undiverted in this manner. A diversion is identified with a name, which is any immutable
   object such an integer or string. When recalled, diverted code is not resent through the
   EmPy interpreter (although a filter could be set up to do this).

   By default, no diversions take place. When no diversion is in effect, processing output
   goes directly to the specified output file. This state can be explicitly requested at any
   time by calling the empy.stopDiverting function. It is always legal to call this function.

   When diverted, however, output goes to a deferred location which can then be recalled
   later. Output is diverted with the empy.startDiversion function, which takes an argument
   that is the name of the diversion. If there is no diversion by that name, a new diversion
   is created and output will be sent to that diversion; if the diversion already exists,
   output will be appended to that preexisting diversion.

   Output send to diversions can be recalled in two ways. The first is through the
   empy.playDiversion function, which takes the name of the diversion as an argument. This
   recalls the named diversion, sends it to the output, and then erases that diversion. A
   variant of this behavior is the empy.replayDiversion, which recalls the named diversion
```

```
   but does not eliminate it afterwards; empy.replayDiversion can be repeatedly called with
   the same diversion name, and will replay that diversion repeatedly. empy.createDiversion
   create a diversion without actually diverting to it, for cases where you want to make sure
   a diversion exists but do not yet want to send anything to it.

   The diversion object itself can be retrieved with empy.retrieveDiversion. Diversions act
   as writable file-objects, supporting the usual write, writelines, flush, and close
   methods. The data that has been diverted to them can be retrieved in one of two ways;
   either through the asString method, which returns the entire contents of the diversion as
   a single strong, or through the asFile method, which returns the contents of the diversion
   as a readable (not writable) file-like object.

   Diversions can also be explicitly deleted without recalling them with the
   empy.purgeDiversion function, which takes the desired diversion name as an argument.

   Additionally there are three functions which will apply the above operations to all
   existing diversions: empy.playAllDiversions, empy.replayAllDiversions, and
   empy.purgeAllDiversions. All three will do the equivalent of a empy.stopDiverting call
   before they do their thing.

   The name of the current diversion can be requested with the empy.getCurrentDiversion
   function; also, the names of all existing diversions (in sorted order) can be retrieved
   with empy.getAllDiversions.

   When all processing is finished, the equivalent of a call to empy.playAllDiversions is
   done.

Filters

   EmPy also supports dynamic filters. Filters are put in place right "before" the final
   output file, and so are only invoked after all other processing has taken place (including
   interpreting and diverting). Filters take input, remap it, and then send it to the output.

   The current filter can be retrieved with the empy.getFilter function. The filter can be
   cleared (reset to no filter) with empy.resetFilter and a special "null filter" which does
   not send any output at all can be installed with empy.nullFilter. A custom filter can be
   set with the empy.setFilter function; for convenience, specialized forms of filters
   preexist and can be accessed with shortcuts for the empy.setFilter argument:
     * None is a special filter meaning "no filter"; when installed, no filtering whatsoever
       will take place. empy.setFilter(None) is equivalent to empy.resetFilter().
     * 0 (or any other numeric constant equal to zero) is another special filter that
       represents the null filter; when installed, no output will ever be sent to the
       filter's sink.
     * A filter specified as a function (or lambda) is expected to take one string argument
       and return one string argument; this filter will execute the function on any input and
       use the return value as output.
     * A filter that is a string is a 256-character table is substituted with the result of a
       call to string.translate using that table.
     * A filter can be an instance of a subclass of empy.Filter. This is the most general
       form of filter. (In actuality, it can be any object that exhibits a Filter interface,
       which would include the normal file-like write, flush, and close methods, as well as
       next, attach, and detach methods for filter-specific behavior.)
     * Finally, the argument to empy.setFilter can be a Python list consisting of one or more
       of the above objects. In that case, those filters are chained together in the order
       they appear in the list. An empty list is the equivalent of 'None'; all filters will
       be uninstalled.

   Filters are, at their core, simply file-like objects (minimally supporting write, flush,
```

```
   and close methods that behave in the usual way) which, after performing whatever
   processing they need to do, send their work to the next file-like object or filter in
   line, called that filter's "sink." That is to say, filters can be "chained" together; the
   action of each filter takes place in sequence, with the output of one filter being the
   input of the next. Additionally, filters support a _flush method (note the leading
   underscore) which will always flush the filter's underlying sink; this method should be
   not overridden.

   Filters also support three additional methods, not part of the traditional file interface:
   attach, which takes as an argument a file-like object (perhaps another filter) and sets
   that as the filter's "sink" -- that is, the next filter/file-like object in line. detach
   (which takes no arguments) is another method which flushes the filter and removes its
   sink, leaving it isolated. Finally, next is an accessor method which returns the filter's
   sink -- or None, if the filter does not yet have a sink attached.

   To create your own filter, you can create an object which supports the above described
   interface, or simply derive from the empy.Filter class and override its write and possibly
   flush methods. You can chain filters together by passing them as elements in a list to the
   empy.setFilter function, or you can chain them together manually with the attach method:
        firstFilter.attach(secondFilter)
        empy.setFilter(firstFilter)

   or just let EmPy do the chaining for you:
        empy.setFilter([firstFilter, secondFilter])

   In either case, EmPy will walk the filter chain and find the end and then hook that into
   the appropriate interpreter stream; you need not do this manually.

   Subclasses of empy.Filter are already provided with the above null, function, and string
   functionality described above; they are NullFilter, FunctionFilter, and StringFilter,
   respectively. In addition, a filter which supports buffering, BufferedFilter, is provided.
   Several variants are included: SizeBufferedFilter, a filter which buffers into fixed-sized
   chunks, LineBufferedFilter, a filter which buffers by lines, and MaximallyBufferedFilter,
   a filter which completely buffers its input.

Hooks

   The EmPy system also allows for the usage of "hooks," which are callbacks that can be
   registered with an interpreter to get information on the current state of activity and act
   upon it.

   Hooks are associated with names, which are merely strings; these strings represent a state
   of the interpreter. Any number of hooks can be associated with a given name, and are
   registered with the empy.addHook function call. Hooks are callable objects which take two
   arguments: first, a reference to the interpreter that is running; and second, a dictionary
   that contains contextual information about the point at which the hook is invoked; the
   contents of this dictionary are dependent on the hook name.

   Hooks can perform any reasonable action, with one caveat: When hooks are invoked,
   sys.stdout may not be properly wrapped and so should be considered unusable. If one wishes
   to really write to the actually stdout stream (not the interpreter), use
   sys.__stdout__.write. If one wishes to send output to the interpreter, then use
   interpreter.write. Neither references to sys.stdout nor print statements should ever
   appear in a hook.

   The hooks associated with a given name can be retrieved with empy.getHooks. All hooks
   associated with a name can be cleared with empy.clearHooks, and all hooks associated with
   all names can be cleared with empy.clearAllHooks. A hook added with empy.addHook can be
```

```
removed with empy.removeHook. Finally, hooks can be manually invoked via empy.invokeHook.

The following hooks are supported; also listed in curly braces are the keys contained in
the dictionary argument:

at_shutdown
       The interpreter is shutting down.

at_handle {meta}
       An exception is being handled; meta is the exception (an instance of MetaError).
       Note that this hook is invoked when the exception is handled by the EmPy system,
       not when it is thrown.

before_include {name, file}
       An empy.include call is about to be processed; name is the context name of the
       inclusion and file is the actual file object associated with the include.

after_include
       An empy.include was just completed.

before_expand {string, locals}
       An empy.expand call is about to be processed. string is the actual data that is
       about to be processed; locals is the locals dictionary or None.

after_expand
       An empy.expand was just completed.

at_quote {string}
       An empy.quote call is about to be processed; string is the string to be quoted.

at_escape {string}
       An empy.escape call is about to be processed; string is the string to be escaped.

before_file {name, file}
       A file object is just about to be processed. name is the context name associated
       with the object and file is the file object itself.

after_file
       A file object has just finished processing.

before_string {name, string}
       A standalone string is just about to be processed. name is the context name
       associated with it and string is the string itself.

after_string
       A standalone string has just finished being processed.

at_parse {scanner}
       A parsing pass is just about to be performed. scanner is the scanner associated
       with the parsing pass.

before_evaluate {expression, locals}
       A Python expression is just about to be evaluated. expression is the (string)
       expression, and locals is the locals dictionary or None.

after_evaluate
       A Python expression was just evaluated.
```

```
   before_execute {statements, locals}
          A chunk of Python statements is just about to be evaluated. statements is the
          (string) statement block, and locals is the locals dictionary or None.

   before_single {source, locals}
          A single interactive source code fragment (just as in the Python interpreter) is
          about to be executed via Interpreter.single. source is the code (expression or
          statement) to execute, and locals is the locals directory or None.

   after_single
          A single has just taken place.

   before_substitute {substitution}
          A @[...] substitution is just about to be done. substitution is the substitution
          string itself.

   after_substitute
          A substitution just took place.

   before_significate {key, value}
          A significator is just about to be processed; key is the key and value is the
          value.

   after_significate
          A significator was just processed.

   As a practical example, this sample Python code would print a pound sign followed by the
   name of every file that is included with 'empy.include':
        def includeHook(interpreter, keywords):
            interpreter.write("# %s\n" % keywords['name'])
        empy.addHook('before_include', includeHook)

   Note that this snippet properly uses a call to interpreter.write instead of executing a
   print statement.

Data flow

   input -> interpreter -> diversions -> filters -> output

   Here, in summary, is how data flows through a working EmPy system:

    1. Input comes from a source, such an .em file on the command line, or via an
       empy.include statement.
    2. The interpreter processes this material as it comes in, expanding token sequences as
       it goes.
    3. After interpretation, data is then sent through the diversion layer, which may allow
       it directly through (if no diversion is in progress) or defer it temporarily.
       Diversions that are recalled initiate from this point.
    4. Any filters in place are then used to filter the data and produce filtered data as
       output.
    5. Finally, any material surviving this far is sent to the output stream. That stream is
       stdout by default, but can be changed with the -o or -a options, or may be fully
       buffered with the -B option (that is, the output file would not even be opened until
       the entire system is finished).

  Pseudomodule contents

   The empy pseudomodule (available only in an operating EmPy system) contains the following
```

```
functions and objects (and their signatures, with a suffixed opt indicating an optional
argument):

First, basic identification:

VERSION
       A constant variable which contains a string representation of the EmPy version.

SIGNIFICATOR_RE_STRING
       A constant variable representing a regular expression string that can be used to
       find significators in EmPy code.

interpreter
       The instance of the interpreter that is currently being used to perform execution.

argv
       A list consisting of the name of the primary EmPy script and its command line
       arguments, in analogue to the sys.argv list.

args
       A list of the command line arguments following the primary EmPy script; this is
       equivalent to empy.argv[1:].

identify() -> string, integer
       Retrieve identification information about the current parsing context. Returns a
       2-tuple consisting of a filename and a line number; if the file is something other
       than from a physical file (e.g., an explicit expansion with empy.expand, a
       file-like object within Python, or via the -E or -F command line options), a string
       representation is presented surrounded by angle brackets. Note that the context
       only applies to the EmPy context, not the Python context.

setName(name)
       Manually set the name of the current context.

setLine(line)
       Manually set the line number of the current context; line must be a numeric value.
       Note that afterward the line number will increment by one for each newline that is
       encountered, as before.

atExit(callable)
       Register a callable object (or function) taking no arguments which will be called
       at the end of a normal shutdown. Callable objects registered in this way are called
       in the reverse order in which they are added, so the first callable registered with
       empy.atExit is the last one to be called. Note that although the functionality is
       related to hooks, empy.atExit does no work via the hook mechanism, and you are
       guaranteed that the interpreter and stdout will be in a consistent state when the
       callable is invoked.

Globals manipulation:

getGlobals()
       Retrieve the globals dictionary for this interpreter. Unlike when calling globals()
       in Python, this dictionary can be manipulated and you can expect changes you make
       to it to be reflected in the interpreter that holds it.

setGlobals(globals)
       Reseat the globals dictionary associated with this interpreter to the provided
       mapping type.
```

```
updateGlobals(globals)
       Merge the given dictionary into this interpreter's globals.

clearGlobals(globals_opt)
       Clear out the globals (restoring, of course, the empy pseudomodule). Optionally,
       instead of starting with a refresh dictionary, use the dictionary provided.

Filter classes:

Filter
       The base Filter class which can be derived from to make custom filters.

NullFilter
       A null filter; all data sent to the filter is discarded.

FunctionFilter
       A filter which uses a function taking a string and returning another to perform the
       filtering.

StringFilter
       A filter which uses a 256-character string table to map any incoming character.

BufferedFilter
       A filter which does not modify its input, but instead holds it until it is told to
       flush (via the filter's flush method). This also serves as the base class for the
       other buffered filters below.

SizeBufferedFilter
       A filter which buffers into fixed-size chunks, with the possible exception of the
       last chunk. The buffer size is indicated as the sole argument to the constructor.

LineBufferedFilter
       A filter which buffers into lines, with the possible exception of the last line
       (which may not end with a newline).

MaximallyBufferedFilter
       A filter which does not flush any of its contents until it is closed. Note that
       since this filter ignores calls to its flush method, this means that installing
       this filter and then replacing it with another can result in loss of data.

The following functions allow direct execution; optional locals arguments, if specified,
are treated as the locals dictionary in evaluation and execution:

evaluate(expression, locals_opt)
       Evaluate the given expression.

execute(statements, locals_opt)
       Execute the given statement(s).

single(source, locals_opt)
       Interpret the "single" source code, just as the Python interactive interpreter
       would.

substitute(substitution, locals_opt)
       Perform the given substitution.

significate(key, value_opt)
       Do a manual signification. If value is not specified, it is treated as None.
```

```
The following functions relate to source manipulation:

include(file_or_filename, locals_opt)
       Include another EmPy file, by processing it in place. The argument can either be a
       filename (which is then opened with open in text mode) or a file object, which is
       used as is. Once the included file is processed, processing of the current file
       continues. Includes can be nested. The call also takes an optional locals
       dictionary which will be passed into the evaluation function.

expand(string, locals_opt) -> string
       Explicitly invoke the EmPy parsing system to process the given string and return
       its expansion. This allows multiple levels of expansion, e.g., @(empy.expand("@(2 +
       2)")). The call also takes an optional locals dictionary which will be passed into
       the evaluation function. This is necessary when text is being expanded inside a
       function definition and it is desired that the function arguments (or just plain
       local variables) are available to be referenced within the expansion.

quote(string) -> string
       The inverse process of empy.expand, this will take a string and return a new string
       that, when expanded, would expand to the original string. In practice, this means
       that appearances of the prefix character are doubled, except when they appear
       inside a string literal.

escape(string, more_opt) -> string
       Given a string, quote the nonprintable characters contained within it with EmPy
       escapes. The optional more argument specifies additional characters that should be
       escaped.

flush()
       Do an explicit flush on the underlying stream.

string(string, name_opt, locals_opt)
       Explicitly process a string-like object. This differs from empy.expand in that the
       string is directly processed into the EmPy system, rather than being evaluated in
       an isolated context and then returned as a string.

Changing the behavior of the pseudomodule itself:

flatten(keys_opt)
       Perform the equivalent of from empy import ... in code (which is not directly
       possible because empy is a pseudomodule). If keys is omitted, it is taken as being
       everything in the empy pseudomodule. Each of the elements of this pseudomodule is
       flattened into the globals namespace; after a call to empy.flatten, they can be
       referred to simple as globals, e.g., @divert(3) instead of @empy.divert(3). If any
       preexisting variables are bound to these names, they are silently overridden. Doing
       this is tantamount to declaring an from ... import ... which is often considered
       bad form in Python.

Prefix-related functions:

getPrefix() -> char
       Return the current prefix.

setPrefix(char)
       Set a new prefix. Immediately after this call finishes, the prefix will be changed.
       Changing the prefix affects only the current interpreter; any other created
       interpreters are unaffected.
```

```
Diversions:

stopDiverting()
       Any diversions that are currently taking place are stopped; thereafter, output will
       go directly to the output file as normal. It is never illegal to call this
       function.

createDiversion(name)
       Create a diversion, but do not begin diverting to it. This is the equivalent of
       starting a diversion and then immediately stopping diversion; it is used in cases
       where you want to make sure that a diversion will exist for future replaying but
       may be empty.

startDiversion(name)
       Start diverting to the specified diversion name. If such a diversion does not
       already exist, it is created; if it does, then additional material will be appended
       to the preexisting diversions.

playDiversion(name)
       Recall the specified diversion and then purge it. The provided diversion name must
       exist.

replayDiversion(name)
       Recall the specified diversion without purging it. The provided diversion name must
       exist.

purgeDiversion(name)
       Purge the specified diversion without recalling it. The provided diversion name
       must exist.

playAllDiversions()
       Play (and purge) all existing diversions in the sorted order of their names. This
       call does an implicit empy.stopDiverting before executing.

replayAllDiversions()
       Replay (without purging) all existing diversions in the sorted order of their
       names. This call does an implicit empy.stopDiverting before executing.

purgeAllDiversions()
       Purge all existing diversions without recalling them. This call does an implicit
       empy.stopDiverting before executing.

getCurrentDiversion() -> diversion
       Return the name of the current diversion.

getAllDiversions() -> sequence
       Return a sorted list of all existing diversions.

Filters:

getFilter() -> filter
       Retrieve the current filter. None indicates no filter is installed.

resetFilter()
       Reset the filter so that no filtering is done.

nullFilter()
       Install a special null filter, one which consumes all text and never sends any text
```

```
        to the output.

  setFilter(filter)
        Install a new filter. A filter is None or an empty sequence representing no filter,
        or 0 for a null filter, a function for a function filter, a string for a string
        filter, or an instance of empy.Filter. If filter is a list of the above things,
        they will be chained together manually; if it is only one, it will be presumed to
        be solitary or to have already been manually chained together. See the "Filters"
        section for more information.

  Hooks:

  enableHooks()
        Enable invocation of hooks. By default hooks are enabled.

  disableHooks()
        Disable invocation of hooks. Hooks can still be added, removed, and queried, but
        invocation of hooks will not occur (even explicit invocation with empy.invokeHook).

  areHooksEnabled()
        Return whether or not hooks are presently enabled.

  getHooks(name)
        Get a list of the hooks associated with this name.

  clearHooks(name)
        Clear all hooks associated with this name.

  clearAllHooks(name)
        Clear all hooks associated with this name.

  addHook(name, hook, prepend_opt)
        Add this hook to the hooks associated with this name. By default, the hook is
        appended to the end of the existing hooks, if any; if the optional insert argument
        is present and true, it will be prepended to the list instead.

  removeHook(name, hook)
        Remove this hook from the hooks associated with this name.

  invokeHook(name_, ...)
        Manually invoke all the hooks associated with this name. The remaining arguments
        are treated as keyword arguments and the resulting dictionary is passed in as the
        second argument to the hooks.

 Invocation

  Basic invocation involves running the interpreter on an EmPy file and some optional
  arguments. If no file are specified, or the file is named -, EmPy takes its input from
  stdin. One can suppress option evaluation (to, say, specify a file that begins with a
  dash) by using the canonical -- option.

  -a/--append (filename)
        Open the specified file for append instead of using stdout.

  -f/--flatten
        Before processing, move the contents of the empy pseudomodule into the globals,
        just as if empy.flatten() were executed immediately after starting the interpreter.
        That is, e.g., empy.include can be referred to simply as include when this flag is
```

```
       specified on the command line. This can also be specified through the existence of
       the EMPY_FLATTEN environment variable.

-h/--help
       Print usage and exit.

-H/--extended-help
       Print extended usage and exit. Extended usage includes a rundown of all the legal
       expansions, escape sequences, pseudomodule contents, used hooks, and supported
       environment variables.

-i/--interactive
       After the main EmPy file has been processed, the state of the interpreter is left
       intact and further processing is done from stdin. This is analogous to the Python
       interpreter's -i option, which allows interactive inspection of the state of the
       system after a main module is executed. This behaves as expected when the main file
       is stdin itself. This can also be specified through the existence of the
       EMPY_INTERACTIVE environment variable.

-k/--suppress-errors
       Normally when an error is encountered, information about its location is printed
       and the EmPy interpreter exits. With this option, when an error is encountered
       (except for keyboard interrupts), processing stops and the interpreter enters
       interactive mode, so the state of affairs can be assessed. This is also helpful,
       for instance, when experimenting with EmPy in an interactive manner. -k implies -i.

-o/--output (filename)
       Open the specified file for output instead of using stdout. If a file with that
       name already exists it is overwritten.

-p/--prefix (prefix)
       Change the prefix used to detect expansions. The argument is the one-character
       string that will be used as the prefix. Note that whatever it is changed to, the
       way to represent the prefix literally is to double it, so if $ is the prefix, a
       literal dollar sign is represented with $$. Note that if the prefix is changed to
       one of the secondary characters (those that immediately follow the prefix to
       indicate the type of action EmPy should take), it will not be possible to represent
       literal prefix characters by doubling them (e.g., if the prefix were unadvisedly
       changed to # then ## would already have to represent a comment, so ## could not
       represent a literal #). This can also be specified through the EMPY_PREFIX
       environment variable.

-r/--raw-errors
       Normally, EmPy catches Python exceptions and prints them alongside an error
       notation indicating the EmPy context in which it occurred. This option causes EmPy
       to display the full Python traceback; this is sometimes helpful for debugging. This
       can also be specified through the existence of the EMPY_RAW_ERRORS environment
       variable.

-B/--buffered-output
       Fully buffer processing output, including the file open itself. This is helpful
       when, should an error occur, you wish that no output file be generated at all (for
       instance, when using EmPy in conjunction with make). When specified, either the -o
       or -a options must be specified (and the -B option must precede them; full
       buffering does not work with stdout. This can also be specified through the
       existence of the EMPY_BUFFERED_OUTPUT environment variable.

-D/--define (assignment)
```

```
        Execute a Python assignment of the form variable = expression. If only a variable
        name is provided (i.e., the statement does not contain an = sign), then it is taken
        as being assigned to None. The -D option is simply a specialized -E option that
        special cases the lack of an assignment operator. Multiple -D options can be
        specified.

 -E/--execute (statement)
        Execute the Python (not EmPy) statement before processing any files. Multiple -E
        options can be specified.

 -F/--execute-file (filename)
        Execute the Python (not EmPy) file before processing any files. This is equivalent
        to -E execfile("filename") but provides a more readable context. Multiple -F
        options can be specified.

 -I/--import (module)
        Imports the specified module name before processing any files. Multiple modules can
        be specified by separating them by commas, or by specifying multiple -I options.

 -P/--preprocess (filename)
        Process the EmPy file before processing the primary EmPy file on the command line.

 -V/--version
        Print version and exit.

Environment variables

 EmPy also supports a few environment variables to predefine certain behaviors. The
 settings chosen by environment variables can be overridden via command line arguments. The
 following environment variables have meaning to EmPy:

 EMPY_OPTIONS
        If present, the contents of this environment variable will be treated as options,
        just as if they were entered on the command line, before the actual command line
        arguments are processed. Note that these arguments are not processed by the shell,
        so quoting, filename globbing, and the like, will not work.

 EMPY_PREFIX
        If present, the value of this environment variable represents the prefix that will
        be used; this is equivalent to the -p command line option.

 EMPY_FLATTEN
        If defined, this is equivalent to including -f on the command line.

 EMPY_RAW_ERRORS
        If defined, this is equivalent to including -r on the command line.

 EMPY_INTERACTIVE
        If defined, this is equivalent to including -i on the command line.

 EMPY_BUFFERED_OUTPUT
        If defined, this is equivalent to including -B on the command line.

Examples and testing EmPy

 See the sample EmPy file sample.em which is included with the distribution. Run EmPy on it
 by typing something like (presuming a UNIX-like operating system):
       ./em.py sample.em
```

```
and compare the results and the sample source file side by side. The sample content is
intended to be self-documenting.

The file sample.bench is the benchmark output of the sample. Running the EmPy interpreter
on the provided sample.em file should produce precisely the same results. You can run the
provided test script to see if your EmPy environment is behaving as expected:
     ./test.sh

By default this will test with the first Python interpreter available in the path; if you
want to test with another interpreter, you can provide it as the first argument on the
command line, e.g.:
     ./test.sh python2.1
     ./test.sh /usr/bin/python1.5
     ./test.sh jython

Embedding EmPy

Embedding EmPy into your application is quite simple. The relative complexity of the
em.invoke function is due to handling every possible combination of options (via the
command line and environment variables). An EmPy interpreter can be created with as code
as simple as:
     import em
     interpreter = em.Interpreter()
     # The following prints the results to stdout:
     interpreter.string("@{x = 123}@x\n")
     # This expands to the same thing, but puts the results as a
     # string in the variable result:
     result = interpreter.expand("@{x = 123}@x\n")
     # Process an actual file (and output to stdout):
     interpreter.file('/path/to/some/file')

When you are finished with your interpreter, it is important to call its shutdown method:
     interpreter.shutdown()

This will ensure that the interpreter cleans up all its overhead, entries in the
sys.stdout proxy, and so forth. It is usually advisable that this be used in a
try...finally clause:
     interpreter = em.Interpreter(...)
     try:
         ...
     finally:
         interpreter.shutdown()

The em.Interpreter constructor takes the following arguments; all are optional:

output
       The output file which the interpreter will be sending all its processed data to.
       This need only be a file-like object; it need not be an actual file. If omitted,
       sys.__stdout__ is used.

argv
       An argument list analogous to sys.argv, consisting of the script name and zero or
       more arguments. These are available to executing interpreters via empy.argv and
       empy.args. If omitted, a non-descript script name is used with no arguments.

prefix
       The single character prefix. Defaults to @.
```

```
 options
        A dictionary of options that can override the default behavior of the interpreter.
        The names of the options are constant names ending in _OPT and their defaults are
        given in Interpreter.DEFAULT_OPTIONS.

 globals
        By default, interpreters begin with a pristine dictionary of globals (except, of
        course, for the empy pseudomodule). Specifying this argument will allow the globals
        to start with more.

 Many things can be done with EmPy interpreters; for the full developer documentation, see
 the generated documentation for the em module.

Interpreter options

 The following options (passed in as part of the options dictionary to the Interpreter
 constructor) have the following meanings. The defaults are shown below and are also
 indicated in an Interpreter.DEFAULT_OPTIONS dictionary.

 BANGPATH_OPT
        Should a bangpath (#!) as the first line of an EmPy file be treated as if it were
        an EmPy comment? Note that #! sequences starting lines or appearing anywhere else
        in the file are untouched regardless of the value of this option. Default: true.

 BUFFERED_OPT
        Should an abort method be called upon failure? This relates to the fully-buffered
        option, where all output can be buffered including the file open; this option only
        relates to the interpreter's behavior after that proxy file object has been
        created. Default: false.

 RAW_OPT
        Should errors be displayed as raw Python errors (that is, the exception is allowed
        to propagate through to the toplevel so that the user gets a standard Python
        traceback)? Default: false.

 EXIT_OPT
        Upon an error, should execution continue (although the interpreter stacks will be
        purged)? Note that even in the event this is set, the interpreter will halt upon
        receiving a KeyboardInterrupt. Default: true.

 FLATTEN_OPT
        Upon initial startup, should the empy pseudomodule namespace be flattened, i.e.,
        should empy.flatten be called? Note this option only has an effect when the
        interpreter is first created; thereafter it is ignored. Default: false.

Known issues and caveats

   * EmPy was primarily intended for static processing of documents, rather than dynamic
     use, and hence speed of processing was not a major consideration in its design.
   * EmPy is not threadsafe.
   * Expressions (@(...)) are intended primarily for their return value; statements
     (@{...}) are intended primarily for their side effects, including of course printing.
     If an expression is expanded that as a side effect prints something, then the printing
     side effects will appear in the output before the expansion of the expression value.
   * Due to Python's curious handling of the print keyword -- particularly the form with a
     trailing comma to suppress the final newline -- mixing statement expansions using
     prints inline with unexpanded text will often result in surprising behavior, such as
     extraneous (sometimes even deferred!) spaces. This is a Python "feature," and occurs
```

```
      in non-EmPy applications as well; for finer control over output formatting, use
      sys.stdout.write or empy.interpreter.write (these will do the same thing) directly.
    * To function properly, EmPy must override sys.stdout with a proxy file object, so that
      it can capture output of side effects and support diversions for each interpreter
      instance. It is important that code executed in an environment not rebind sys.stdout,
      although it is perfectly legal to invoke it explicitly (e.g., @sys.stdout.write("Hello
      world\n")). If one really needs to access the "true" stdout, then use sys.__stdout__
      instead (which should also not be rebound). EmPy uses the standard Python error
      handlers when exceptions are raised in EmPy code, which print to sys.stderr.
    * The empy "module" exposed through the EmPy interface (e.g., @empy) is an artificial
      module. It cannot be imported with the import statement (and shouldn't -- it is an
      artifact of the EmPy processing system and does not correspond to any accessible .py
      file).
    * For an EmPy statement expansion all alone on a line, e.g., @{a = 1}, note that this
      will expand to a blank line due to the newline following the closing curly brace. To
      suppress this blank line, use the symmetric convention @{a = 1}@.
    * When using EmPy with make, note that partial output may be created before an error
      occurs; this is a standard caveat when using make. To avoid this, write to a temporary
      file and move when complete, delete the file in case of an error, use the -B option to
      fully buffer output (including the open), or (with GNU make) define a .DELETE_ON_ERROR
      target.
    * empy.identify tracks the context of executed EmPy code, not Python code. This means
      that blocks of code delimited with @{ and } will identify themselves as appearing on
      the line at which the } appears, and that pure Python code executed via the -D, -E and
      -F command line arguments will show up as all taking place on line 1. If you're
      tracking errors and want more information about the location of the errors from the
      Python code, use the -r command line option, which will provide you with the full
      Python traceback.

Wish list

 Here are some random ideas for future revisions of EmPy. If any of these are of particular
 interest to you, your input would be appreciated.
    * Some real-world examples should really be included for demonstrating the power and
      expressiveness of EmPy first-hand.
    * A "trivial" mode, where all the EmPy system does is scan for simple tokens replace
      them with evaluations/executions, rather than having to do the contextual scanning it
      does now. This has the down side of being much less configurable and powerful but the
      upside of being extremely efficient. Perhaps this need not be a separate mode, but an
      additional prefix something of the form @<(...)>, @<{...}>, and possibly @<[12][...]>?
      Setting the trivial mode might simply disallow other expansions.
    * A "debug" mode, where EmPy prints the contents of everything it's about to evaluate
      (probably to stderr) before it does?
    * The ability to funnel all code through a configurable RExec for user-controlled
      security control. This would probably involve abstracting the execution functionality
      outside of the interpreter.
    * Optimized handling of processing would be nice for the possibility of an Apache module
      devoted to EmPy processing.
    * An EmPy emacs mode.
    * An "unbuffered" option which would lose contextual information like line numbers, but
      could potentially be more efficient at processing large files.
    * An optimization of offloading diversions to files when they become truly huge.
    * Unicode support, particularly for filters. (This may be problematic given Python 1.5.2
      support.)
    * Support for mapping filters (specified by dictionaries).
    * Support for some sort of batch processing, where several EmPy files can be listed at
      once and all of them evaluated with the same initial (presumably expensive)
      environment.
```

```
    * A more elaborate interactive mode, perhaps with a prompt and readline support.
    * A toplevel run function, which invoke delegates to, that accepts arguments similar to
      the command line as keyword arguments. Perhaps also a simplified wrapper just for
      doing basic processing, e.g., interpreter.simple?
    * A tool to collect significator information from a hierarchy of .em files and put them
      in a database form available for individual scripts would be extremely useful.
    * A StructuredText and/or reStructuredText filter would be quite useful, as would
      SGML/HTML/XML, s-expression, Python, etc. auto-indenter filters.
    * A caching system that stores off the compilations of repeated evaluations and
      executions so that in a persistent environment the same code does not have to be
      repeatedly evaluated/executed. This would probably be a necessity in an Apache
      module-based solution.
    * An option to change the format of the standard EmPy messages in a traceback.
    * An "binary" option to have EmPy process incoming data in chunks, rather than by lines,
      for handling of non-textual data or data which may not contain predictably short
      lines.
    * Support for some manner of implicitly processed /etc/empyrc and/or ~/.empyrc file, and
      of course an option to inhibit its processing. This can already be accomplished via an
      explicit EMPY_OPTIONS, but still ...
    * More uniform handling of the preprocessing directives (-I, -D, -E, -F, and -P),
      probably mapping directly to methods in the Interpreter class.
    * distutils support.

Author's notes

 I originally conceived EmPy as a replacement for my [13]Web templating system which uses
 [14]m4 (a general macroprocessing system for UNIX).

 Most of my Web sites include a variety of m4 files, some of which are dynamically
 generated from databases, which are then scanned by a cataloging tool to organize them
 hierarchically (so that, say, a particular m4 file can understand where it is in the
 hierarchy, or what the titles of files related to it are without duplicating information);
 the results of the catalog are then written in database form as an m4 file (which every
 other m4 file implicitly includes), and then GNU make converts each m4 to an HTML file by
 processing it.

 As the Web sites got more complicated, the use of m4 (which I had originally enjoyed for
 the challenge and abstractness) really started to become an impediment to serious work;
 while I am very knowledgeable about m4 -- having used it for for so many years -- getting
 even simple things done with it is awkward and difficult. Worse yet, as I started to use
 Python more and more over the years, the cataloging programs which scanned the m4 and
 built m4 databases were migrated to Python and made almost trivial, but writing out huge
 awkward tables of m4 definitions simply to make them accessible in other m4 scripts
 started to become almost farcical -- especially when coupled with the difficulty in
 getting simple things done in m4.

 It occurred to me what I really wanted was an all-Python solution. But replacing what used
 to be the m4 files with standalone Python programs would result in somewhat awkward
 programs normally consisting mostly of unprocessed text punctuated by small portions where
 variables and small amounts of code need to be substituted. Thus the idea was a sort of
 inverse of a Python interpreter: a program that normally would just pass text through
 unmolested, but when it found a special signifier would execute Python code in a
 persistent environment. After considering between choices of signifiers, I settled on @
 and EmPy was born.

 As I developed the tool, I realized it could have general appeal, even to those with
 widely varying problems to solve, provided the core tool they needed was an interpreter
 that could embed Python code inside templated text. As I continue to use the tool, I have
```

```
been adding features as unintrusively as possible as I see areas that can be improved.

A design goal of EmPy is that its feature set should work on several levels; at each
level, if the user does not wish or need to use features from another level, they are
under no obligation to do so. If you have no need of substitutions, for instance, you are
under no obligation to use them. If significators will not help you organize a set of EmPy
scripts globally, then you need not use them. New features that are being added are
whenever possible transparently backward compatible; if you do not need them, their
introduction should not affect you in any way. The use of unknown prefix sequences results
in errors, guaranteeing that they are reserved for future use.

Release history

   * 2.3; 2003 Feb 20. Proper and full support for concurrent and recursive interpreters;
     protection from closing the true stdout file object; detect edge cases of interpreter
     globals or sys.stdout proxy collisions; add globals manipulation functions
     empy.getGlobals, empy.setGlobals, and empy.updateGlobals which properly preserve the
     empy pseudomodule; separate usage info out into easily accessible lists for easier
     presentation; have -h option show simple usage and -H show extened usage; add NullFile
     utility class.
   * 2.2.6; 2003 Jan 30. Fix a bug in the Filter.detach method (which would not normally be
     called anyway).
   * 2.2.5; 2003 Jan 9. Strip carriage returns out of executed code blocks for DOS/Windows
     compatibility.
   * 2.2.4; 2002 Dec 23. Abstract Filter interface to use methods only; add @[noop: ...]
     substitution for completeness and block commenting.
   * 2.2.3; 2002 Dec 16. Support compatibility with Jython by working around a minor
     difference between CPython and Jython in string splitting.
   * 2.2.2; 2002 Dec 14. Include better docstrings for pseudomodule functions; segue to a
     dictionary-based options system for interpreters; add empy.clearAllHooks and
     'empy.clearGlobals'; include a short documentation section on embedding interpreters;
     fix a bug in significator regular expression.
   * 2.2.1; 2002 Nov 30. Tweak test script to avoid writing unnecessary temporary file; add
     Interpreter.single method; expose evaluate, execute, substitute, and single methods to
     the pseudomodule; add (rather obvious) EMPY_OPTIONS environment variable support; add
     empy.enableHooks and 'empy.disableHooks'; include optimization to transparently
     disable hooks until they are actually used.
   * 2.2; 2002 Nov 21. Switched to -V option for version information; empy.createDiversion
     for creating initially empty diversion; direct access to diversion objects with
     'empy.retrieveDiversion'; environment variable support; removed --raw long argument
     (use --raw-errors instead); added quaternary escape code (well, why not).
   * 2.1; 2002 Oct 18. empy.atExit registry separate from hooks to allow for normal
     interpreter support; include a benchmark sample and test.sh verification script;
     expose empy.string directly; -D option for explicit defines on command line; remove
     ill-conceived support for @else: separator in @[if ...] substitution; handle nested
     substitutions properly; @[macro ...] substitution for creating recallable expansions.
   * 2.0.1; 2002 Oct 8. Fix missing usage information; fix after_evaluate hook not getting
     called; add empy.atExit call to register values.
   * 2.0; 2002 Sep 30. Parsing system completely revamped and simplified, eliminating a
     whole class of context-related bugs; builtin support for buffered filters; support for
     registering hooks; support for command line arguments; interactive mode with -i;
     significator value extended to be any valid Python expression.
   * 1.5.1; 2002 Sep 24. Allow @] to represent unbalanced close brackets in @[...] markups
     [now defunct; use escape codes instead].
   * 1.5; 2002 Sep 18. Escape codes (@\...); conditional and repeated expansion
     substitutions via @[if E:...], @[for X in E:...], and @[while E:...] notations; fix a
     few bugs involving files which do not end in newlines.
   * 1.4; 2002 Sep 7. Fix bug with triple quotes; collapse conditional and protected
```

```
      expression syntaxes into the single generalized @(...) notation; empy.setName and
      empy.setLine functions; true support for multiple concurrent interpreters with
      improved sys.stdout proxy; proper support for empy.expand to return a string evaluated
      in a subinterpreter as intended; merged Context and Parser classes together, and
      separated out Scanner functionality.
    * 1.3; 2002 Aug 24. Pseudomodule as true instance; move toward more verbose (and clear)
      pseudomodule functions; fleshed out diversion model; filters; conditional expressions;
      protected expressions; preprocessing with -P (in preparation for possible support for
      command line arguments).
    * 1.2; 2002 Aug 16. Treat bangpaths as comments; empy.quote for the opposite process of
      'empy.expand'; significators (@%... sequences); -I option; -f option; much improved
      documentation.
    * 1.1.5; 2002 Aug 15. Add a separate invoke function that can be called multiple times
      with arguments to simulate multiple runs.
    * 1.1.4; 2002 Aug 12. Handle strings thrown as exceptions properly; use getopt to
      process command line arguments; cleanup file buffering with AbstractFile; very slight
      documentation and code cleanup.
    * 1.1.3; 2002 Aug 9. Support for changing the prefix from within the empy pseudomodule.
    * 1.1.2; 2002 Aug 5. Renamed buffering option to -B, added -F option for interpreting
      Python files from the command line, fixed improper handling of exceptions from command
      line options (-E, -F).
    * 1.1.1; 2002 Aug 4. Typo bugfixes; documentation clarification.
    * 1.1; 2002 Aug 4. Added option for fully buffering output (including file opens),
      executing commands through the command line; some documentation errors fixed.
    * 1.0; 2002 Jul 23. Renamed project to EmPy. Documentation and sample tweaks; added
      empy.flatten. Added -a option.
    * 0.3; 2002 Apr 14. Extended "simple expression" syntax, interpreter abstraction, proper
      context handling, better error handling, explicit file inclusion, extended samples.
    * 0.2; 2002 Apr 13. Bugfixes, support non-expansion of Nones, allow choice of alternate
      prefix.
    * 0.1.1; 2002 Apr 12. Bugfixes, support for Python 1.5.x, add -r option.
    * 0.1; 2002 Apr 12. Initial early access release.

  Author

   This module was written by [15]Erik Max Francis. If you use this software, have
   suggestions for future releases, or bug reports, [16]I'd love to hear about it.

   Even if you try out EmPy for a project and find it unsuitable, I'd like to know what
   stumbling blocks you ran into so they can potentially be addressed in a future version.

  Version

   Version 2.3 $Date$ $Author$

   Modules and Packages
   [17]em

   A system for processing Python as markup embedded in text.
     ____________________________________________________________________________

   [18]Table of Contents
   This document was automatically generated on Thu Feb 20 03:56:27 2003 by [19]HappyDoc
   version 2.0.1

References

   Visible links
```

```
  1. file://localhost/home/shafi/pak/empy-2.3/doc/index.html
  2. file://localhost/home/shafi/pak/empy-2.3/doc/index.html#refindex
  3. http://www.alcyone.com/pyos/empy/empy-latest.tar.gz
  4. http://www.alcyone.com/pyos/empy/
  5. http://www.gnu.org/copyleft/gpl.html
  6. mailto:empy-announce-list-subscribe@alcyone.com
  7. mailto:empy-list-subscribe@alcyone.com
  8. file://localhost/home/shafi/pak/empy-2.3/doc/index.html#refindex
  9. file://localhost/home/shafi/pak/empy-2.3/doc/index.html#refname
 10. file://localhost/home/shafi/pak/empy-2.3/doc/index.html#refsub
 11. file://localhost/home/shafi/pak/empy-2.3/doc/index.html#refi
 12. file://localhost/home/shafi/pak/empy-2.3/doc/index.html#ref...
 13. http://www.alcyone.com/max/info/m4.html
 14. http://www.seindal.dk/rene/gnu/
 15. http://www.alcyone.com/max/
 16. mailto:pyos@alcyone.com
 17. file://localhost/home/shafi/pak/empy-2.3/doc/em.py.html
 18. file://localhost/home/shafi/pak/empy-2.3/doc/index.html
 19. http://happydoc.sourceforge.net/

  Hidden links:
 20. file://localhost/home/shafi/pak/empy-2.3/doc/index.html#index
```

CHAPTER 11

index

- genindex
- modindex
- search