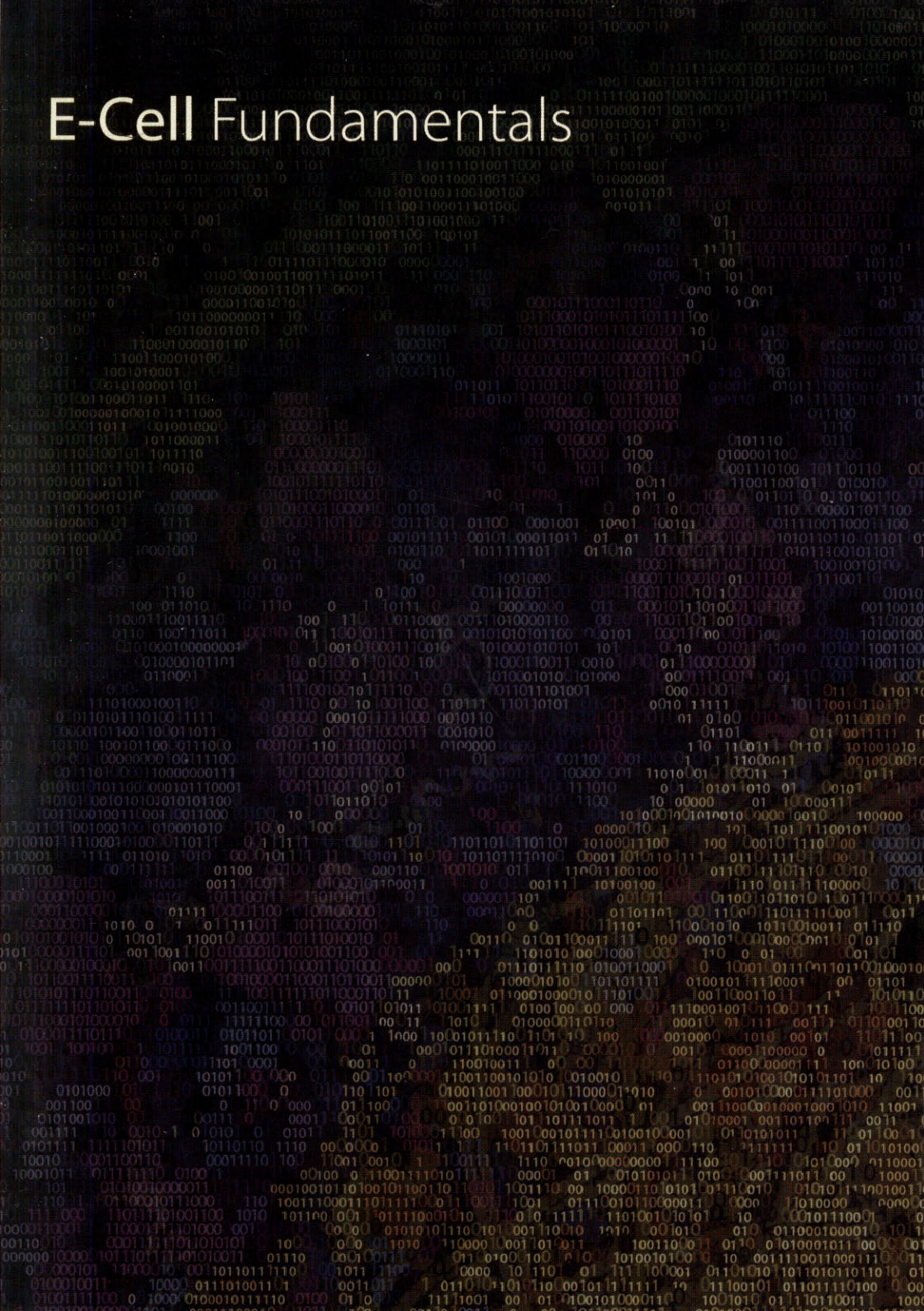
Title	E-cell fundamentals
Sub Title	
Author	慶應義塾大学湘南藤沢キャンパス先端生命科学研究会(Keio gijuku daigaku Shonan Fujisawa kyanpasu sentan seimei kagaku kenkyukai) 内藤, 泰宏(Naito, Yasuhiro)
Publisher	慶應義塾大学湘南藤沢学会
Publication year	2010-03
Jtitle	リサーチメモ
JaLC DOI	
Abstract	
Notes	
Genre	Technical Report
URL	https://koara.lib.keio.ac.jp/xoonips/modules/xoonips/detail.php?koara_id=0302-0000-0629

E-Cell Fundamentals



E-Cell
Fundamentals

慶應義塾大学湘南藤沢キャンパス
先端生命科学研究会

慶應義塾大学 湘南藤沢学会

目　次

はじめに

1

E-Cellとは

E-Cell Simulation Environment（E-Cell SE）は、さまざまな細胞内現象のシミュレーションを実行するためのソフトウェア環境です。

本書の構成

本書は以下の各章より構成されます。

1. はじめに（本章）
2. E-Cell をはじめよう
3. E-Cellによるモデル作成
4. モデル作成のチュートリアル
5. スクリプトによるセッションの操作
6. 新規オブジェクトクラスの作成
7. 標準ダイナミックモジュール ライブラリ
8. E-Cell のシミュレーション機構
9. E-Cell SE について

Appendix

1. EmPy モジュール
2. サンプルモデル
3. システムのファイル構成
4. セッションモニタ マニュアル

ユーザとしてE-Cellを使いはじめたいなら、2、3、4章を読み、プロジェクトをはじめてください。必要に応じて他の章も読み進めてください。モデルを作成するために用いることができるクラスに関する情報を得るには7章をご覧ください。独自のモジュールを開発するためには6章を読んでください。シミュレーションの実行を自動

化するためには５章が役立つでしょう。
E-Cell SE のフロントエンドモジュール（Python 言語で記述します）の開発に関心があるなら５章を中心に読んでください。システムがどのように構成され、利用されるかについてよく知らないなら、２、３章も読んでください。
C++ダイナミックモジュール（新規アルゴリズムモジュールなど）を開発するのであれば６章を読む必要があります。E-Cell についてよく知らなければ、２、３章を、すでに提供されているクラスの設計について知りたければ７章を読んでください。
E-Cell SEがシミュレーションを実行する仕組みについて関心があるなら８章を読んでください。

Appendix

モデルファイル中で利用できるマクロ EmPy の簡易マニュアルを Appendix-1 に収録します。
システムとともに提供されるサンプルモデルの内容については Appendix-2 をご覧ください。
E-Cell SE がインストールするファイルの構成については Appendix-3 をご覧ください。
E-Cell の GUI であるセッションモニタの簡易マニュアルを Appendix-4 に収録します。

必要となる知識

本書は、以下に挙げるような内容については、前提知識として特に詳細な説明をしません。それらについては、それぞれの教科書、解説書等で学んでください。

UNIX 系プラットホームの使い方

E-Cell SE は、基本的に UNIX 系プラットホームのソフトウェアとして開発されています。セッションモニタなどの GUI を備えていますが、UNIX のコマンドラインの操作ぬきに E-Cell を使いこなすのは不可能です。
UNIX 系プラットホームの基本的な使い方については、本書では説明しません。

オブジェクト指向とはなにか

E-Cell はオブジェクト指向のプログラム言語を用いて書かれたプログラムです。そして、モデルファイルで定義されるモデルの構造もまたオブジェクト指向であり、モデルファイルがシステムに読み込まれると、モデルファイルに表現された構造をそのまま持ったオブジェクト群としてコンピュータ上に構成されます。
このように、E-Cell は完全なオブジェクト指向に則って細胞シミュレーションを

実行します。ですから、オブジェクト指向について理解していることによって、E-Cell の設計、動作に関する理解が格段に深まります。

本書では、オブジェクト指向プログラミングの一般的な用語（オブジェクト、インスタンス、コンストラクタなど）を、特に説明することなく使っています。

Python 言語

E-Cell は、主に C++ と Python の 2 つのプログラミング言語で書かれています。中でも Python は、ユーザが E-Cell にアクセスするためのプログラム言語なので、E-Cell を利用する上で Python 言語の基本的な使い方を知っていることはほぼ必須といえます。

例えば、E-Cell による作業を自動化するスクリプトは、Python 言語そのもので書くことになります。また、E-Cell のモデル言語である EM は Python の書式を多く流用しています。また、モデルファイルの中では、Python 言語による文を書く局面も多くあり、Python 言語を書ければ、モデル作成の効率が向上します。

Python 言語については、多くの優れた解説書がありますので、それらを利用して学んでください。

C++言語

E-Cell を使う上で、Python に加えて C++言語を使えることによる利益は、シミュレーションの高速化、効率化であるといえます。Python を使えれば、一般的な E-Cell の機能をすべて利用することができるので、C++を知らないからできないということはほとんどありません。ただ、Python だけでもできることのいくつかは、C++を駆使することによって、より簡単に、より高速に実現できます。

C++は、E-Cell のコアライブラリを書くのに使われています。Python とは異なり、C++を知らなくても E-Cell を使うことは可能です。C++で書かれたシステムコアが提供する機能のうち一般的なユーザが必要とするものは、すべて Python を介して利用可能になっているからです。

一方、新しいダイナミックモジュールの作成といった一部の作業では、C++のコードそのものを書くことになります。C++を使ってこれらの機能を利用することができれば、シミュレーションを効率化するのに役立ちます。

C++言語については、多くの優れた解説書がありますので、それらを利用して学んでください。

E-Cell をはじめよう

2

本章は以下の項目について書かれています。

- シミュレーションを実行するために必要なファイルとその形式
- シミュレーションを実行するために必要なファイルの作り方
- E-Cell SEを使ってシミュレーションを実行する方法

シミュレーションの準備

シミュレーションを開始するには以下の形式のファイルが必要です。

- EML形式で書かれたモデルファイル
- （省略可能）E-Cell SEが提供しない独自のオブジェクトクラスをモデル中で使用する場合、その共有オブジェクトファイル（ファイルの拡張子は通常、UNIX系プラットホームでは`.so`、Mac OS Xでは`.dylib`、Windowsでは`.dll`です。）
- （省略可能）シミュレーションのセッションを自動的に実行する場合、そのためのスクリプトファイル（E-Cellセッションスクリプト、ESS）

EMからEMLへの変換

E-Cellのシミュレーションモデルは多くの場合 EM 形式で書かれています。EM（`.em`）ファイルをEML（`.eml`）ファイルに変換するには次のコマンドを実行します。

```
$ ecell3-em2eml filename.em
```

C++ダイナミックモジュールのコンパイル

シミュレーションモデルを作成するために、C++言語で書かれたダイナミックモジュール（DM）のソースコードを要する場合があります。その場合、シミュレーションを実行する前に、ソースコードファイルを予めコンパイルおよびリンクして共有モジュールファイルを作成しておく必要があります。また、DM を利用するには環

境変数 ECELL3_DM_PATH を適切に設定する必要があります（後述）。

DM をコンパイルおよびリンクするには、ecell3-dmc コマンドを用いるのが簡単です。

```
$ ecell3-dmc [options] filename.cpp [compiler options]
```

ファイル名 *filename.cpp* の前の引数 [*options*] は、ecell3-dmc コマンド自身のオプションとして処理されます。ファイル名の後の引数 [*compiler options*] はバックエンドで実行されるコンパイラ（g++ など）に引き渡されます。バックエンドのコンパイラは、システムそのものをビルドする際に用いたものと同じです。

コマンド内で実行されている処理を確認するためには、-v オプション（冗長モード）を用います。

入力ファイルを指定せず、 -h オプションとともに ecell3-dmc を実行することで、すべてのオプションのリストを得ることができます。ecell3-dmc -h で表示されるヘルプメッセージは以下の通りです。

```
Compile dynamic modules for E-Cell Simulation Environment Version 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.
```

E-Cell SEの起動

E-Cell SE は、スクリプトモードあるいはGUIモードで起動することができます。

GUIモード

E-Cell SEをGUIモードで起動するには、以下のコマンドを用います。

```
$ ecell3-session-monitor &
```

このコマンドは、シミュレータインスタンスを、GUIフロントエンドであるE-Cell セッションモニタとともに立ち上げます。

スクリプトモード

E-Cell SEをスクリプトモードで起動するには、以下のコマンドを用います。

```
$ ecell3-session [filename.ess]
```

filename.ess は実行するPythonスクリプトファイルの名前です。
filename.ess を省略して実行すると、インタプリタがインタラクティヴモードで立ち上がります。

スクリプトについては5章をご覧ください。

DM 探索パスと環境変数 ECELL3_DM_PATH

モデルが、ecell3-dmc でビルドするなどして作成した標準の DM 以外の DM を利用している場合、環境変数 ECELL3_DM_PATH で、それらの DM が格納されているパスを指定する必要があります。
環境変数 ECELL3_DM_PATH は、複数のディレクトリ名を保持できます。その際の区切り文字は、UNIX 系のプラットホームでは：（コロン）、Windows では；（セミコロン）です。
以下に、ecell3-session-monitor を起動する前に ECELL3_DM_PATH を設定する例を示します：

```
$ ECELL3_DM_PATH=.:/home/example/mydms
$ export ECELL3_DM_PATH
$ ecell3-session-monitor
```

E-Cell SE 3.1.105 までは、カレントディレクトリが暗黙のうちに ECELL3_DM_PATH に含まれていました。この不適切な仕様は 3.1.106 以降は取り除かれています。

その他のコマンド

EML から EM への変換

ecell3-eml2em コマンドを使って、EMLファイルをEMファイルに変換できます。

```
ecell3-eml2em -- convert a EML file to a EM file

Usage:
    ecell3-eml2em [-h] [-f] [-o EMFILE] infile.eml

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

E-Cell によるモデル作成

3

本章は以下の項目について書かれています。

- E-Cell のシミュレーションモデルの構成
- シミュレーションモデルの作り方
- EM 形式でのモデルファイルの書き方

モデル中のオブジェクト

E-Cell では、シミュレーションモデルを完全にオブジェクト指向で記述します。すなわち、シミュレーションモデルは相互に結合した**オブジェクト**の集合となっています。オブジェクトは**属性**（Property）を持ち、属性はオブジェクトの性質（化学反応であれば反応速度定数）およびオブジェクト間の関係を決定づけます。

オブジェクト指向による生命現象の表現

生命現象をオブジェクト指向で表現するために、E-Cell では、モデル化の対象とする生命システムを、システムが作動する場：System、システムを構成する要素（モノ）：Variable、システム内で起こる動き（要素間相互作用）：Process の 3 つの概念の集合として捉えます。
どのような生命現象も、このコンセプトで捉えることができます。
例えば、真核細胞の解糖系は、以下のように記述できるでしょう：

System：　細胞質
Variable：　グルコース、G6P、F6P、……ピルビン酸、ATP、ADP、……
Process：　グルコキナーゼ、ホスホフルクトキナーゼ、……

E-Cell ではこのコンセプトに沿ったオブジェクトを用意しており、オブジェクト指向で記述したモデルを、そのまま、コンピュータ上に構築することができます。

オブジェクトの型

E-Cell SE のシミュレーションモデルは以下の型のオブジェクトによって構成されます。

- 通常、２つ以上の Entity オブジェクト
- １つ以上の Stepper オブジェクト

Entity オブジェクトはシミュレーションモデルの構造を定義し、モデル中の現象（化学反応など）を表現します。Stepper オブジェクトはそれぞれシミュレーションアルゴリズムを実装しています。

Entity オブジェクト

Entity には３つの下位クラス（派生クラス）があります。

System

System オブジェクトは、モデルの全体構造を定義します。
System オブジェクトは、その中に他の Entity オブジェクト（System, Variable, Process）を持つことができます。
複数の System によって木構造をつくることができます。
一般には、反応の起こる場を表す、細胞質やオルガネラの内腔のような空間、細胞膜のような区画を表現するのに用います。

Variable

状態変数を表すオブジェクトです。
ひとつの Variable オブジェクトは、ひとつのスカラー実数値（Value 属性）を持っています。
モデルに含まれるすべての Variable の値によって、ある時刻におけるモデルの状態を定義できます。
一般には、分子など物質の量を表します。

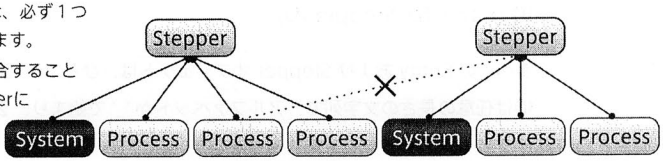
Process

モデル中で生じる現象を記述します。
Process は、ひとつ以上の Variable の値を時間発展に応じて変更します。
一般には、酵素反応や物質の移動などを表します。

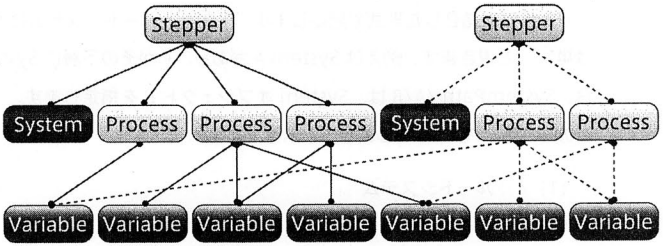
Stepper オブジェクト

モデルは、１つ以上の Stepper オブジェクトを含んでいなければなりません。ひとつのモデル中のすべての Process、System オブジェクトは、それぞれ特定の１つの Stepper オブジェクトに結合されていなければなりません。換言すれば、モデル中の Stepper オブジェクト（群）は、重複なく Process、System オブジェクトと結合しています。

Process、Systemは、必ず1つのStepperと結合します。2つのStepperに結合することはなく（×）、Stepperに結合しないこともありません。



Stepper は、特定のシミュレーションアルゴリズムを実装したクラスです。モデルが2つ以上の Stepper オブジェクトを持っていると、複数 Stepper シミュレーションとなります。Process、System オブジェクトのリストに加え、Stepper は、読み込み／書き込み可能な Variable のリストを持っています。Variable のリストは、Stepper が持つ Process オブジェクトから与えられます。また、Stepperは、時間ステップ間隔を正の実数で持っています。システムはステップ間隔に従って Stepper オブジェクトをスケジュールし、現在時刻を更新します。



Stepper は、結合する Process を介して、関連する Variable のリストを得ます。Variable はそれぞれ複数の Process、Stepper と結合することができます。

上の図の右の Stepper は、破線で結ばれた System、Process に結合しており、それらの Process が結合する（破線）Variable を自らが関連する Variable としてリスト化します。

システムから呼ばれると、Stepperオブジェクトは、（モデルが微分の要素を含んでいる場合）関連づけられたVariableオブジェクトの値を現在時刻まで積分し、Stepper に結合した Process をアルゴリズムによって決められた順序に従って呼びだし、次の時間ステップ間隔を決定します。シミュレーション手順の詳細については次章以降を読んでください。

オブジェクト識別子

E-Cell SEでは、モデル中のEntityやStepperを指定するために、いくつかの識別子（ID）を用います。

ID (Entity ID, Stepper ID)

すべての Entity および Stepper オブジェクトは、ひとつの ID を持ちます。
IDは任意の長さの文字列で、アルファベットか '_' で始まり、２文字目以降には、アルファベット、'_' に加え数字も使えます。
アルファベットの大文字と小文字は区別されます。
ID を Stepper オブジェクトに対して用いる場合、Stepper ID といいます。Entity オブジェクトに対して用いる場合は、Entity ID あるいは単に ID といいます。

例：`_P3, ATP, GlucoKinase`

SystemPath

SystemPath は、シミュレーションモデル中で樹状の階層構造をつくる System オブジェクト群から、特定の System を指定する識別子で、Entity ID を「/」（スラッシュ）で結合した形式で記述します。ただし、ルートシステムの SystemPath は単に / と書きます。例えば System A があり、A がその下層に System B を持つ場合、SystemPath `/A/B` は、System オブジェクト B を指定します。
この SystemPath は３つの部分からなります。

(1) / ：ルートシステム
(2) System A： A はルートシステムの直下に位置します。
(3) System B： B は、System A の直下に位置します。

SystemPath は相対表記することができます（ただし、現在の System が確定していなければなりません）。
以下の場合、SystemPath は相対表記と解釈されます。

(1) SystemPath の表記が「/」（ルートシステム）から始まらない。
(2) SystemPath が「.」（現在の System）または「..」（直上の上位 System）を含む。

例：`/A/B, ../A, ., /Cytoplasm/Mitochondria, ../Chloroplast`

FullID

FullID（完全表記識別子、FULLy qualified IDentifier）はシミュレーションモデル中の特定のEntity オブジェクトを指定します。FullID は、(1) EntityType (2) SystemPath (3) Entity ID の３部分からなり、各部分を「:」でつなぎ合わせて以下のように表記します。

```
EntityType: SystemPath:ID
```

EntityType は以下のいずれかです。

- System
- Process
- Variable

例えば以下の FullID は、System /A/B に含まれる Process P を指します。

```
Process:/A/B:P
```

FullID の表記には、スペース（空白）を入れてはいけません。

FullPN (Fully qualified Property Name)

FullPN （完全表記属性名、FULLy qualified Property Name）はモデル構造の中のある Entity が有する特定の属性（詳細は次節）を指定する識別子です。FullPN は、FullID と属性の名前を「:」でつなぎ合わせて表記します。

```
FullID:属性の名前
```

すなわち、以下のようになります。

```
EntityType: SystemPath:ID: 属性の名前
```

以下のFullPNは、System /A/B に含まれる Process P の Activity 属性を指します。

```
Process:/A/B:P:Activity
```

オブジェクトの属性

Entity および Stepper オブジェクトは属性を持っています。属性はオブジェクトの持つ特性を表しており、その名前は表現する特性に因んでいます（MolarConc 属性が表すのはオブジェクトのモル濃度である等）。属性の値は、シミュレーション中に読み出したり、書き込んだりすることができます。

属性の型

属性の値には 型 があり、以下のいずれかです。

Real（実数）型

例：1.0, 3.33e+10

Integer（整数）型

例：1, 300

String（文字列）型

文字列String型には引用符付きと引用符無しの２つの形式があります。引用符付きのStringは、引用符自身（「'」または「"」）を除くすべての ASCII文字を含むことができます（日本語を含めたマルチバイト文字を含むことはできません）。文字列が正当な識別子（Entity ID、Stepper ID、SystemPath、FullID、FullPN）である場合、引用符を省略できます。三重引用符（ ''' または """ ）で括った Stringには改行記号を含むことができます（複数行の文字列を含むことができます）。

例：`_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.'`

リスト型

リストには、Real型、Integer型、String型の値を含めることができます。また、リストの要素としてリストを含めることもできます（ネストすることができます）。リストは、角カッコ [ ] で囲むことで表記します。要素間は、スペース（空白）によって分割します。最も外側の角カッコが省略される場合があります（EM ファイルなど）。

例：`[A 10 [ 1.0 "a string" 1e+10 234] ]`

属性値の動的な型適用

属性値の型が、シュミュレータ中のオブジェクト（Process、Variableなど）が求める型と異なる場合、システムは自動的に型変換を行います。これによりバックエンドのオブジェクトが受け取った型と与えられた型が異なっていても、システムエラーは起きません。システムは、シミュレータ中のオブジェクトが要求する型にモデルファイル中の値を変換するよう試みます。変換に成功すると、シミュレータ中のオブジェクトは属性の値を取得することができます。以下の節も参照してください。変換は以下のように行われます。

属性値の型適用の方法

数値型（Real または Integer）からの変換

String 型へ

数値は単純に文字列に変換されます。例えば、数値 12.3 は String '12.3' に変換されます。

リスト型へ

数値は、その数値を最初の要素とする長さ１のリストに変換されます。例えば、12.3 は、リスト [12.3] に変換されます。

String 型からの変換

数値型（Real または Integer）へ

文字列の最初の部分が数値に変換されます。数は、10進あるいは16進表記で書くことができます。先 頭の空白は無視されます。「INF」および「NAN」（大文字小文字は無視）は、それぞれ無限大、NaN（非数）に変換されます。文字列の最初の部分が数 値に変換できない場合は、ゼロ（0.0または0）と見なされます。この変換手続きは、C言語の strtol および strtod 関数と同じです。

リスト型へ

文字列は、その文字列を最初の要素とする長さ１のリストに変換されます。例えば、'string' は、リスト ['string'] に変換されます。

リスト型からの変換

数値型あるいは String 型へ

単純に、リストの最初の要素が取り出されます。取り出された要素の型は、必要に応じてさらに変換されます。

属性値変換の際のオーバーフロー、アンダーフロー：Real 型から Integer 型へ、あるいは String 型から数値型への変換の際に、オーバーフロー（桁あふれ）やアンダーフロー（下位桁あふれ）が起こることがあります。バックエンドのオブジェクトが変換を試みた際にこれらが生じた場合、例外が発生します。

E-Cellモデル（EM）ファイルの基礎

これまでに、E-Cell のシミュレーションモデルが数種類のオブジェクトからなり、オブジェクトは属性を持つことを学びました。次に、シミュレーションモデルはどのように構築されているのか、そのモデルの構造を学びましょう。E-Cell モデルファイル

を記述する言語を E-Cell モデル（EM）言語といいます。これ以降、本書に掲載されている例の多くは EM 言語で記述されていますので、次の章に進む前に EM ファイルの文法を学んでおくと、内容の理解を助けてくれるでしょう。

EM とは

E-Cell SEでは、モデルの記述と交換に XML ベースのモデル記述言語 EML (E-Cell Model description Language) を用います。
EML はモデルを記述し、E-Cell と他のソフトウェアで統一的にモデルを扱うためには理想的な記述形式ですが、人間が読み書きするにはあまりにも冗長です。
そこで、人間が読み書きしやすい形式で、EMLと互換性のあるモデル記述言語として、EM (E-Cell Model) が用意されています。EM はプログラム言語に似通った形式でモデルを記述できる言語です。EM と EML は記述されている情報について完全に等価で、相互に変換することができます（ただし、EM 中に記述されたコメントは例外で、EML には変換されません）。EM ファイルの拡張子は「.em」です。

なぜ、そしていつ EM を使うのか

EML を基盤とした、より洗練されたスケーラブルかつインテリジェントなモデル構築のしくみを E-Cell モデリング環境（E-Cell ME、開発中）が提供しますが、EM の文法とセマンティクスを学ぶことで、E-Cell 内部でオブジェクトモデルがどのように構築され、それを用いたシミュレーションがどのように実行されているかを理解することができます。また、プログラム言語様の文法を持つ EM は、他のユーザとコミュニケートする際、シンプルで直観的なツールとして役立ちます。本書でも、E-Cell 上でのモデル構築を説明するために EM を用いています。
EM は、EML を生成するためのスクリプトであるともいえます。

一目で見る EM

EMの細かい文法を学ぶ前に、ごくシンプルな EM ファイルを見てみましょう。今すべて理解する必要はありません。次の例をざっと眺めてみましょう。

例 3-1. 簡単なEMの例

```
Stepper ODE45Stepper( 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 ];
    }
}
```

この例は、質量作用則（law of mass-action）と呼ばれる単純な反応形式の、微分方程式モデルです。このモデルには Stepper オブジェクト ODE_1 が定義されています。Stepper オブジェクトのクラスは ODE45Stepperです。このクラスは一般的な常微分方程式のソルバです。次に、ルートシステム (/) が定義されています。ルートシステム内には、StepperID 属性、ならびに 4 つの Entity オブジェクト（3 つの Variable オブジェクト SIZE、S、P、および Process オブジェクト E）が定義され

ています。 SIZE は、コンパートメントの容積を定義する特殊な Variable オブジェクトです。コンパートメントが3次元空間の場合は、単位はリットル（L）になります。 SIZE の値は、他のオブジェクトの濃度を計算するために用いられます。Entity オブジェクトは、それぞれに異なるさまざまな属性を持っています。ルートシステムの属性 StepperID は、引用符で括られていない文字列 ODE_1 です。Variable S の初期値は、Value 属性で与えられます。ここでは「10000」と記述していますが、自動的にReal型 10000.0 に変換されます。Process E の Name 属性は引用符で括られた文字列 "A mass action from S to P" です。k 属性はReal型 1.0 です。VariableReferenceList は、2つのリストを要素とするリストです（最も外側の角カッコは除外されています）。このリストには、文字列（例：S0）、数値（例：-1）、相対 FullID（例：:.:S）（引用符で括られていない文字列）が含まれています。

EM の一般的な文法

EM（EML も）に記述されてるのは、オブジェクトのインスタンス化の手順です。これまで見てきたように、E-Cell のモデルを構成するオブジェクトは、基本的にたった2種類です（Stepper と Entity）。オブジェクトを作成したら、そのすべての属性を設定しなければなりません。したがって、オブジェクトの作成は以下の2ステップからなります：(1) オブジェクトを新しく作成する (2) 属性を設定する。

オブジェクトのインスタンス化宣言文の一般的な書式

EM中でオブジェクトを定義（インスタンス化）するための一般的な書式は以下の通りです。

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

書式の内容は以下の通りです。

TYPE

オブジェクトの型を書きます。型は、以下のうちのどれかです。

- Stepper
- Variable
- Process
- System

ID

Stepper の場合は StepperID を記述します。

System の場合は SystemPath を記述します。

Variable または Process の場合は Entity ID を記述します。

CLASSNAME

オブジェクトのクラス名。*CLASSNAME* は、*TYPE* によって定義された基底クラスの派生クラスでなければなりません。

例えば、 *TYPE* が Process の場合、 *CLASSNAME* は Process の派生クラスでなければなりません（MassActionFluxProcessなど）。

INFO

オブジェクトの説明。この項目はオプションで、シミュレーションには用いられません。引用符で括られた単一行（`"string"`）あるいは複数行（`"""multi-line string"""`）として記述できます。

PROPERTY（属性）

定義上、オブジェクトは0個以上の属性を持ちます。

属性の記述は、引用符無しの属性の名前を表す文字列と、値を一組として表記され、末尾にセミコロン（;）を付します。

例えば、属性の名前が Concentration で、値が「10.0」の場合、以下のように記述します。

```
Concentration 10.0;
```

値としては、Real型、Integer型、String型、リストを用いることができます。値としてリストを用いる場合、リストの最も外側の角カッコは省略されます。例えば、属性 Foo の値としてリスト `[ 10 "string" [ LIST ] ]` を与えたい場合、以下のように表記します。

```
Foo 10 "string" [ LIST ];
```

角カッコを省略するわけ：すべての属性の値はリストとして扱われています（値が単一のスカラー値である場合でも）。実数型の値 `1.0` は、1つの要素をもつリスト `[1.0]` として扱われています。同様に E-Cell は、角カッコのないリストをリストとして処理

します。
したがって、もし最も外側の角カッコを省略せず、以下のように書いたとすると：

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

以下のように要素にリストをもつ、要素数1のリストとして処理されます。

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

このリストの最初の要素は[10 [LIST]]となります。

マクロとプリプロセッシング

EML への変換に先立って、ecell3-em2eml は EmPy プリプロセッサ（前処理を行うプログラム）を呼び出し、EM ファイルのプリプロセッシング（前処理）を行います。
EmPy を用いることによって、Python 言語による処理をマクロとして EM に埋め込むことができます。
EM ファイル内で「@」に引きつづいて Python の式や文を書くことで、EmPy による処理が行われます。
Python による式を「@(Python の式)」の形（これがマクロです）で EM ファイルに埋め込んでおくと、EmPy によって式が評価され、その評価結果（戻り値）がマクロ部分と置換されます。
式が単純な場合、「()」は省略可能です。
Python の文を埋め込みたい場合は、「@{ python statements }」のように波カッコ{}を用います。
以下に例を示します：

```
@{ AA = 10 }
@( AA * 2 )
@AA
```

これは、以下のように展開されます。

```
20
10
```

もちろん複数行にわたる記述もできます。例えば：

```
@{
def f( str ):
    return str + ' is true.'
}
@f( 'Video Games Boost Visual Skills' )
```

このコードは、以下のように展開されます。

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

EmPyは、他のファイルを取り込むためにも使えます。

以下の1行を表記すると、この行は、EMLへの変換に先だって、ファイル「foo.em」の内容に置き換えられます。

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

ecell3-em2eml コマンド実行時に、-E オプションを用いると、プリプロセッシング（前処理）中に行われた処理を確認することができます。-E オプションを使用すると、プリプロセッシングの結果が標準出力（一般にはコンソール）に出力され、EMLファイルは作成されません。

EmPy を用いることで、柔軟性の高い高度なモデリングが可能となります。EmPy にはさらに多くの特長があります。EM ファイルでよく利用する機能について、Appendix-1 で解説しています。すべての機能を知るには、EmPy の web をご覧ください。

コメント

シャープ（#）はコメント文字です。ある1行の中で、コメント文字以降に書かれた文字はコメントとして扱われ、ecell3-em2eml コマンドの処理対象外となります。引用符で括られた文字列の中に書かれたシャープはコメント文字として処理されません。

EmPy のコメント（@#）は異なった処理をされます。

「@#」は EmPy によって処理されますが、Python のコメントして評価されるので、結果として EML には何も反映されません。

ただし、「#」以降の文字列は、EmPy によるプリプロセッシングの対象となりま

す。したがって、EmPy マクロをコメントアウトしたい場合は、「#@ Python の式」ではなく、「@# Python の式」と書かなければなりません。

現バージョンでは、ecell3-em2eml はマルチバイト文字に未対応であるため、EM ファイルに日本語の 2 バイト文字などASCII文字以外の文字を含めることはできません。一方、EmPyはUTF-8形式に対応しています。そのため、EmPy中のコメントとして「@# コメント」「@''' コメント '''」のような書式でコメントを埋め込めば、日本語などのマルチバイト文字を含めることが可能になります（EMLファイルの文字コードはUTF-8にしてください）。これをecell3-em2emlが処理すると、プリプロセスの段階でEmPyのコメント行が削除され、マルチバイト文字に未対応の ecell3-em2eml 本体での処理にはコメント中のマルチバイト文字が引き渡されないため、正常に変換が行われます。

モデルの構造

最上位の要素

通常、EM は、1 つ以上の Stepper 宣言文と、1 つ以上の System 宣言文を含みます。これらの宣言文は、モデルの階層構造中の最上位の要素です。
一般的な EM ファイルの構造は以下のようになります。

```
STEPPER_0
STEPPER_1
...
STEPPER_n
SYSTEM_0 # the root system ( '/' )
SYSTEM_1
...
SYSTEM_m
```

STEPPER_? は Stepper の宣言文です。SYSTEM_? は System の宣言文です。

System

ルートシステム

モデルには、必ず SystemPath '/' を持つ System オブジェクトが存在します。この System をルートシステム（root system）と呼びます。

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

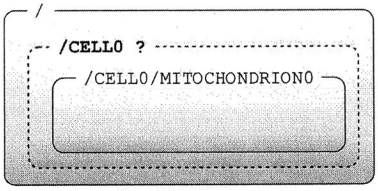
ルートシステムのクラスは System でなければなりません。
E-Cell システムはモデルファイルを読み込む前にルートシステムを作成するため、ユーザが改変した System クラスをルートシステムとすることはできません。
EML ファイルに書かれているルートシステム自体の宣言は、EML ファイル読み込み時には無視されており、属性の設定だけが行われています。したがって、クラス名を指定しても無視されます。
属性をひとつも設定しない場合にも、モデルファイルにはルートシステムを記述する必要があります。

System の階層構造

モデルが2つ以上の System オブジェクトを持つ場合、ルートシステムを頂点とする木構造を持つことになります。
以下の EM は無効です。

```
System System( / )
{
}

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

ここで宣言されている2つの System は互いに結合されていません。



モデルの構造をEMに明確かつ完全に定義しなければなりません。
システムは、曖昧なEMに対して、推定・補完を行いません。

もうひとつの System /CELL0 を付加することで、お互いが結合され、有効な EM になります。

```
System System( / )
{
}

System System( /CELL0 )
{
}

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

ひとつの System は任意の数の下位 System を持つことができます。

```
System System( / )
{
}

System System( /CELL1 ) {}
System System( /CELL2 ) {}
System System( /CELL3 ) {}
# ...
```

モデル合成のサポート（計画中）：将来のバージョンで、複数のモデルファイル（EM または EML）からひとつのモデルを合成する機能をサポートする計画があります。これは、EmPy によるインクルードとは異なる機能です。

System のサイズ（容積）

System の大きさ（サイズ）を定義するには、ID SIZE を持つ Variable を作成します。

System が 3 次元のコンパートメントであれば、SIZE は容積を表し、単位はリットル（L）になります。

以下の例では、ルートシステムの容積は 1e-18 L です。

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

SIZE Variableをもたない System はその上位 System と SIZE Variable を共有します（同じ容積を持つものとして定義されます）。
ルートシステムは常に SIZE Variable を持ちます。
モデルファイルによってルートシステムの SIZE が与えられなかった場合のデフォルト値は 1.0 [L] です。
以下の例には4つの System オブジェクトが含まれます。うち2つ（/ と / COMPARTMENT）は、それぞれの SIZE Variable を持っています。残る2つ（/ SUBSYSTEM とその下位システム /SUBSYSTEM/SUBSUBSYSTEM）はルートシステムと SIZE Variable を共有します。

```
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 system
{
    # no SIZE
}

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

SIZE は正の実数でなければなりません。SIZE にゼロあるいは負の数を設定した場合、SIZE は定義されません。

> **SIZEの単位**：現在、SIZE の単位は $(10\ \mathrm{cm})^d$ です。d は Systemの 次元です。d が3の場合、単位は $(10\ \mathrm{cm})^3$ = リットル です。この仕様には現在も議論があり、将来のバージョンで変更されるかもしれません。

Variable と Process

System 宣言文には、任意の数（ゼロの場合もある）の Variable および Process 宣言文が、それらの属性とともに含まれます。

```
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 ) {}
}
```

System 宣言文を（別の） System 宣言文の中に書くことはできません。
System の階層構造は SystemPath によって定義します。

Stepper と Entity オブジェクトの結合

モデル中のすべての Process オブジェクトは、StepperID を指定することによって、それぞれ 1 つの Stepper と結合されていなければなりません。
Process の StepperID が省略された場合は、Process が属する System の StepperID が設定されます。
System の StepperID は省略できません。
以下の例では、ルートシステムは Stepper STEPPER0 に結合されています。
Process P0 と P1 は、それぞれStepper STEPPER0 と STEPPER1 に結合されています。

```
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;
    }
}
```

Stepper と Variable の結合は自動的に決定され、手動で指定することはできません。

Priority 属性

ある Stepper に結合した Process について、計算の順序を決めたい場合があります。例えば、Process B が Process A の計算結果が反映される Variable X の値に基づいて Variable Y の値を更新する場合などが考えられます。すべての Process は、Priority 属性を持ちます。これを設定して、1 ステップ中の Process の計算順序を指定できます。Priority の値の大きい Process の方が、先に計算されます。Priority のデフォルト値は 0 です。Priority の設定によってシミュレーション結果が変わる場合があるので、Priority 属性を利用する際には、その影響について注意深く検討する必要があります。

Variable と Process の結合

Process オブジェクトは、質量作用則（mass action）などそれぞれに実装された反応モデルにしたがって、Variable の値を変化させます。
個々の Process をプログラムする時点では、それらがモデル中で使われる際に結合する Variable の名前はわかりません。
Process が扱う具体的な Variable の設定は、モデルを構築する際に、モデルファイル上でなされます。
Process と Variable を関連づけるために Process オブジェクトの VariableReferenceList 属性を用います。
VariableReferenceList は、1 つ以上の VariableReference を要素に持つリストです。
VariableReference は、以下の 4 つの要素を持つリストです。

```
[ reference_name FullID coefficient accessor_flag ]
```

最後の 2 つ要素は省略できます：

```
[ reference_name FullID coefficient ]
```

あるいは、

```
[ reference_name FullID ]
```

それぞれの要素は以下の意味を持ちます。

reference_name（参照名）

この要素は、この VariableReference に対する Process 内部での呼び名を与えます。Process によっては、参照名によって VariableReference オブジェクトを特定します。

すべての VariableReference に参照名を指定する必要があります（Process 内部で参照されない VariableReference に対しても）。

参照名に使用できる文字列のルールは、Entity ID と同じです。

FullID

VariableReference が参照する Variable の FullID を指定します。

SystemPath は相対パスで表記することもできます。

EntityType は省略可能です。

例えば、Process が /CELL に属している場合、

```
Variable:/CELL:SO
```

と書く代わりに、以下のような表記が可能です。

```
:.:SO
```

coefficient（係数）（省略可能）

coefficient は、整数型の値です。

coefficient は、Process と、VariableReference が参照する Variable 間の量的な関係を定義します。

係数が非ゼロの整数である場合、Process は、その VariableReference が参照する Variable の値を変更することができることを表します。

係数がゼロである場合、Process は、その VariableReference が参照する Variable の値を変更することはできません。

Process が化学反応を表現している場合、係数の値は、化学量論係数に相当します。

例えば、ある VariableReference の *coefficient* が -2 だったなら、順方向の反応が 1 回起こるたびに、この VariableReference が参照する Variable の値は Process が算出する反応速度の 2 倍の値ずつ減少していきます。

coefficient を省略した場合のデフォルトの値はゼロです。

accessor_flag（isAccessor フラグ）（省略可能）

バイナリのフラグです。1（true）あるいは 0（false）に設定します。

isAccessor フラグが false の場合、Process の振る舞いはその VariableReference が指す Variable の影響を受けません。すなわち、Process が Variable の値を読み込むことはありません。Variable の現在の値にかまわず、Process がこれを書き換えることが起こりえます。

Processによっては、ある VariableReference が指す Variable の値を変更しないことが明らかな場合、isAccessor フラグの値を自動的に設定します。このフラグを手動で設定する場合、このフラグを確認せず Variable の値を読み込む Process が数多くあることに注意してください。

デフォルト値は 1（true）です。この要素は頻繁に省略されます。

> **シミュレーション中のisAccessorフラグの用いられ方**：複数Stepperを用いるシミュレーションにおいて、isAccessorフラグの情報がシステムの実行を効率化する場合があります。例えば、Stepper Aに結合するすべてのProcessが、別のStepper Bに結合するVariableを一切変更しないことを知ることができれば、システムは、Stepper Aがいずれかの Variableの値を変更していないかを常に確認する代わりに、Stepper Bのステップ幅をより大きく設定する機会を得ることができます。このフラグは、主に2つ以上のStepperの存在下で用いられます。

Variable S を基質として消費し、Variable P を生成物として生成するルートシステム内での反応 Process R が、Variable E を反応を触媒する酵素として進行する様子をモデルファイルに記述してみます。

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

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

この Process は、酵素を参照名 ENZYME で参照しているので、上記のような VariableReferenceList を与えることになります。

モデル作成の方式

E-Cellは複数アルゴリズムシミュレータです。離散アルゴリズムと連続アルゴリズムの両方を含む、どんな種類のシミュレーションアルゴリズムでも、任意の組み合わせで用いることができます。本節では、モデル化とシミュレーションの研究プロジェクトに適したオブジェクトクラスの組み合わせを見いだす方法を解説します。本節には、利用可能なオブジェクトクラスの完全なリストやそれらの詳細な利用方法は含まれていません。それらを知るためには7章「標準ダイナミックモジュールライブラリ」の章をご覧ください。

離散か連続か

E-Cellは、離散過程と連続過程の両方をモデル化することができ、シミュレーション中でこれらを混合することもできます。システムは、離散および連続系を2つの明確に異なる型のProcessとStepperオブジェクト：離散Process／Stepperおよび連続Process／Stepper によってモデル化します。

Variableは離散かつ連続：VariableならびにSystemは、離散あるいは連続の属する特別なクラスを持ちません。基底Variableクラスは離散および連続の操作の両方をサポートします。なぜなら、Variableはあらゆる型のProcessやStepperと結合可能だからです。また、Systemオブジェクトは、離散と連続の区別が必要となるいかなる計算も行いません。

離散クラス

1つあるいはそれ以上のVariableオブジェクトの離散的な変化をモデル化するProcessを離散Processと呼び、それらは必ず離散Stepperと結合していなければなりません。離散Processは、Stepperの要求があると、関係するVariableの値を直接変更します。
離散Process／Stepperには離散型（離散時間型）と離散イベント型の2つの型があります。

離散型

離散Processは、これと結合するVariableオブジェクト（VariableReferenceListに含まれるVariable）の値を離散的に変更します。Process基底クラスはデフォルトで離散Processとなっているため、現バージョンにはDiscreteProcessという名前の特別なクラスはありません。Variableの値がどのように変更されるかは、ProcessがアクセスするVariableReferencesの値、属性の値、そして時にはStepperの時刻によって決定されます。次項で説明する

離散イベント型のProcessと異なり、Variableの値の離散的な変更がいつ起こるかを特定する必要はありません。代わりに、値の更新は、離散Stepperによって決定されるタイミングで一方的に実行されます。

すべての離散Processは、離散Stepperと呼ばれるStepperに結合していなければなりません。現バージョンでは、基底Stepperクラスが離散的なので、DiscreteStepperという特別なクラスはありません。

離散イベント型

離散イベントは特殊な離散的な事例のひとつです。システムは離散イベントモデル作成のためにDiscreteEventStepperおよびDiscreteEventProcessを用意しています。基底Processクラスの持つ通常の点火メソッド（fire()メソッド、１ステップの数値計算を実行します）に加え、DiscreteEventProcessには、次のイベント（この離散イベントProcessに関係するVariable値の離散的な更新）がいつ発生するかを、ProcessがアクセスするVariableReferencesの値、属性の値、Stepperの現在時刻から計算するメソッドが定義されています。

DiscreteEventStepperは、このメソッドがもたらす情報を用いて、それぞれの離散イベントProcessを次にいつ点火すべきかを決定します。

DiscreteEventStepperはインスタンス化可能です。DiscreteEventStepperの動作についてのより詳細な記述は7章「標準ダイナミックモジュールライブラリ」の章をご覧ください。

連続クラス

一方、Variable オブジェクトの連続的な変化を計算する Process を連続 Process と呼び、連続 Stepper と組み合わせて用います。離散 Process が Variable の値を直接変更したのに対し、連続 Process オブジェクトは、結合する Variable オブジェクトの変化速度を設定することで現象をシミュレートします。連続 Stepper は、連続 Process が算出した変化速度（Velocity 属性）に基づいて Variable オブジェクトの値（Value 属性）を積分し、Process が次に変化速度を再計算すべき時刻を決定します。連続 Process と Stepper は、典型的には、微分方程式と微分方程式ソルバの実装として利用され、微分方程式系のシミュレーションを実現します。

利用可能な離散クラス（抜粋）

以下にE-Cellで利用可能ないくつかの離散クラスを挙げます。

DiscreteEventStepper および GillespieProcess (Gillespie-Gibson ペア)

E-Cellによる離散イベントシミュレーション手法の一例として、Gillespieの確率論アルゴリズムの変法であるNext Reaction Method（Gillespie-Gibsonアルゴリズム）が挙げられます。DiscreteEventStepperはこのアルゴリズムを実装しています。DiscreteEventProcessの派生クラスでGillespieの反応確率方程式を用いて次の反応が起こる時刻を計算するGillespieProcessを、このStepperに結合して用いることで、素化学反応のGillespie-Gibsonの確率論シミュレーションを実行することができます。これらのオブジェクトの使い方は簡単で、GillespieProcessオブジェクトに StepperID、VariableReferenceList および速度反応定数 k を設定するだけです。

DiscreteTimeStepper（離散時間 Stepper）

離散Stepperのひとつとしてシステムが用意しているものにDiscreteTimeStepperがあります。このStepperクラスがインスタンス化されると、ユーザが設定したステップ間隔（StepInterval）で離散 Process オブジェクトを呼びだします。例えば、あるモデル中のDiscreteTimeStepperのStepInterval属性が 0.001（秒）に設定されていると、これに結合しているすべてのProcessオブジェクトは１ミリ秒毎に点火されます。DiscreteTimeStepperは、ステップとステップの中間の時刻を持たないため、離散時間Stepperであり、Processオブジェクトに影響する可能性のあるシステムの状態（Variableオブジェクトの値）の変化を知らせる他のStepperによる割り込みが発生しても、それらを無視します。それらの変化は次のステップで反映されます。

PassiveStepper

PassiveStepper は、もうひとつの離散 Stepper のクラスです。これは、StepInterval が無限大の DiscreteTimeStepper のように振る舞う部分もありますが違いもあります。DiscreteTimeStepperと異なり、Stepperに結合するProcessオブジェクトに影響をあたえるかもしれないシステム状態の変化を知らせるStepperの割り込みを無視しません。

このStepperのProcessオブジェクトがアクセスするVariableの少なくともひとつの変化速度の値が他のStepperオブジェクトによって変更されたときにだけ、Processオブジェクトが点火されればよいような特殊な手順を組み込むためにこのStepperを用います。

PythonProcess

PythonProcessを用いると、ユーザはPython構文によってProcessオブジェクトをスクリプトで操作することができます。
Process の持つ `initialize()` および `fire()` メソッドを、それぞれ、InitializeMethod および FireMethod 属性でスクリプト化できます。
PythonProcessは離散的にも連続的にも用いることができ、この「動作モード」はIsContinuous 属性で設定します。デフォルト値は false（0）で離散的です。連続モードに切り替えるには、この属性をに 1 に設定します。

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

通常のPython構文に加えて、以下のオブジェクト、メソッド、属性を、InitializeMethod および FireMethod の双方のメソッド属性で利用することができます：

属性

PythonProcess では、任意の名前の属性を用いることができます。例えば、以下のコードでは 2 つの新規属性を作成しています。

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

これらの属性をPythonメソッド中で用いることができます：

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

    InitializeMethod "print NewProperty";
    FireMethod '''
KK += 1.0
print KK
''';
}
```

上の例のようにFireMethod属性などに三重引用符を用いて複数行のPython構文を書くときには、行頭の空白がPython言語のインデントとして解釈されることに注意してください。

新しい属性は、Pythonメソッド中で作成することもできます。

```
InitializeMethod "A = 3.0"; @# Aを作成
FireMethod "print A * 2";   @# Aを使用
```

これらの属性はグローバル変数として扱われます。

オブジェクト

`self`

Processオブジェクト自身です。以下の属性を持ちます：

`Activity`

このProcessのActivity属性の値。

`addValue( value )`

各VariableReferenceに、valueにVariableReferenceのcoefficientを乗じた値を加えます。

Processが離散的な場合に限り、このメソッドを利用することができます。IsContinuous属性がfalseであることを確認してください。

`getSuperSystem()`

このProcessが属するSystemを返します。Systemオブジェクトの属性については後述します。

`Priority`

このProcessの Priority 属性の値。

`setFlux( value )`

各VariableReferenceのVelocityに、*value*にVariableReferenceのcoefficientを乗じた値を加えます。

Processが連続的な場合に限り、このメソッドを利用することができます。IsContinuous属性がtrueであることを確認してください。

`StepperID`

このProcessのStepperID。

`VariableReference`

VariableReferenceインスタンスは、このProcessのVariableReferenceList

属性によって与えられ、Pythonメソッド中で用いることができます。各インスタンスは以下の属性を持ちます：

`addValue( value )`

VariableのValue属性に *value* を加える。

`Coefficient`

VariableReferenceのCoefficient属性の値。

`getSuperSystem()`

Variableが属するSystemオブジェクトを返します。

`MolarConc`

Variableの濃度をモル濃度 [M] で返します。

`Name`

VariableReferenceの名前。

`NumberConc`

Variableの濃度を個数濃度 [個数/Variableが属するSystemの容積] で返します。

`IsFixed`

VariableのFixed属性が 0 のとき falseを、それ以外の場合は非ゼロの整数を返します。

`IsAccessor`

VariableReferenceのIsAccessorフラグが 0 のとき falseを、それ以外の場合は非ゼロの整数を返します。

`Value`

Variableの値（Value属性）。

`Velocity`

現在計算中のStepperによる暫定的な反応速度（Velocity）を返します。通常使用しません。

`System`

Systemオブジェクトは以下の属性を持ちます。

`getSuperSystem()`

このSystemの上位SystemのSystemオブジェクトを返します。

`Size`

Systemの容積。

`SizeN_A`

Size × N_A の計算結果を返します。N_Aはアボガドロ数です。

`StepperID`

SystemのStepperID。

以下に PythonProcess の使用例を示します。

例 3-2. PythonProcessの使用例

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

PythonEventProcess

ユーザは、このクラスを用いて時間イベントをスクリプトで操作できます。このクラスでは、`initialize()` と `fire()` に加え、`updateStepInterval()` メソッドをスクリプトすることができます。これを設定するためには UpdateStepIntervalMethod 属性を用います。

PythonProcess で利用可能なものに加えて、PythonEventProcess の self オブジェクトは以下の属性を持ちます：

`StepInterval`

`updateStepInterval()` メソッドが計算した最新の StepInterval。

`DependentProcessList`

このProcessに依存するProcessの ID のタプル。

このクラスのオブジェクトは、DiscreteEventStepper とともに用いなければなりません。

ExpressionAssignmentProcess

ExpressionAssignmentProcess は離散的なVariableの更新を簡単かつ効率的に表現できるよう設計されています。

このクラスの Expression 属性には、平文テキストによって式を与えます。式は、更新後のVariableの値を [個数] で与えなければなりません（[濃度]ではないので注意してください）。以下に ExpressionAssignmentProcess の使用例を示します：

```
Process ExpressionAssignmentProcess( P1 )
{
    k 0.1;
    Expression "S.Value + k";

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

PythonProcess に比べ、このProcessは高速に動作します（代償として表現の柔軟性を犠牲にしています）。

Expression 属性で利用できる要素を以下に示します。利用可能な算術演算子および数学関数は、制御構造を除き SBML level 2 に準拠しています。

定数

数値（例：10, 10.33, 1.33e-5）, true, false（ゼロと等価）, pi（π）, NaN（非数）, INF（無限大）, N_A（アボガドロ数）, exp（ネイピア数 *e* を底とする指数関数）

算術演算子

+, -, *, /, ^（冪（べき）；pow(x, y) と等価）

数学関数

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（アスタリスク「*」を付した関数はWindows 版では利用できません）

pow以外のすべての関数は引数を 1 つとります。powの引数は 2 つです。

比較／論理関数

比較／論理演算子の代わりに以下の関数を利用できます。

eq(*lhs, rhs*)

lhs = *rhs* なら 1 を、それ以外なら 0 を返します。

neq(*lhs, rhs*)

lhs ≠ *rhs* なら 1 を、それ以外なら 0 を返します。

gt(*lhs*, *rhs*)

lhs > *rhs* なら 1 を、それ以外なら 0 を返します。

lt(*lhs*, *rhs*)

lhs < *rhs* なら 1 を、それ以外なら 0 を返します。

geq(*lhs*, *rhs*)

lhs ≧ *rhs* なら 1 を、それ以外なら 0 を返します。

leq(*lhs*, *rhs*)

lhs ≦ *rhs* なら 1 を、それ以外なら 0 を返します。

and(*lhs*, *rhs*)

lhs と *rhs* がともに true なら 1 を、それ以外なら 0 を返します。

or(*lhs*, *rhs*)

lhs と *rhs* の両方あるいはいずれかが true なら 1 を、両方とも false なら 0 を返します。

xor(*lhs*, *rhs*)

lhs と *rhs* のいずれか一方だけが true なら 1 を、それ以外なら 0 を返します。

not(*b*)

b が false なら 1 を、true なら 0 を返します。

例：Variable A、B があり、A ≧ 0 のとき B に属性 k の値を加え、A < 0 のとき何もしない Process P のモデルを示します。

```
Process ExpressionAssignmentProcess( P )
{
    k 0.1;
    Expression "B.Value + geq( A.Value, 0.0 ) *k";

    VariableReferenceList [ A :.:A 0 ] [ B :.:B 1 ];
}
```

属性

PythonProcess同様、ExpressionAssignmentProcess も任意の名前の属性をモデル中で使うことができます。ただし、PythonProcess と異なり、このクラスでは、それらの属性の型は Real に限られます。

オブジェクト

self

このProcessオブジェクト自身です。PythonProcessのサブセットにあたる以下の属性を持ちます：

getSuperSystem()

VariableReference

このProcessのVariableReferenceListによって与えられるVariableReferenceインスタンスを式の中で用いることができます。各インスタンスは、PythonProcessのサブセットにあたる以下の属性を持ちます：

Value
MolarConc
NumberConc
TotalVelocity
Velocity

System

Systemオブジェクトは以下の2つの属性を持ちます。

Size
SizeN_A

複雑な酵素反応速度式をモデルファイルに記述する場合など、Expression 属性にセットする式が非常に長くなることがあります。Expression 属性に書けるのは1つの式ですが、EmPy を用いれば、式を複数行に分割して書くことができ、読みやすさを改善できます。EmPy は、改行の直前に @ を見つけると、改行コードを削除します。これを利用して、1つの式を複数行にわたって書くことができます。
例えば次の記述（改行記号 ⏎ を明示します）は：

```
Expression "@⏎
  (Vmax * Glc.MolarConc / @ ⏎
  ( Km + Glc.MolarConc ) * ((ATP.MolarConc /
ADP.MolarConc ) /@ ⏎
        ( 1.0 / 0.44 + ATP.MolarConc / ADP.MolarConc))) @
⏎
    * self.getSuperSystem().SizeN_A @ ⏎
";
```

は、以下の記述（改行記号を含んでおらず3行で表示されているものが1パラグラフです）と等価です。

```
Expression "(Vmax * Glc.MolarConc / ( Km + Glc.MolarConc )
* ((ATP.MolarConc / ADP.MolarConc ) / ( 1.0 / 0.44 +
ATP.MolarConc / ADP.MolarConc))) * self.getSuperSystem
().SizeN_A ";
```

なお、Python プログラムでは、バックスラッシュ（\）によって行の継続を表現できますが、Expression 属性では \ を使うことはできません。

その他の離散クラス

tau leap法を実行するためのStepperも用意されています。そのためには、TauLeapStepperをGillespieProcessと組み合わせて使います。tau leap法は、ポアソン近似に基づいて重要なイベントが発生しない区間を推定するアルゴリズムで、こうした区間の計算を省略することによって計算コストを軽減し、シミュレーションを加速します。TauLeapStepper の epsilon 属性を用いて、許容する最大誤差を制御できます（大きな値を設定すると近似が緩くなり、計算を高速化できる反面、精度が低下します）。

生化学的反応経路の動的-静的ハイブリッドシミュレーションのための流束分布法は、以下のクラスによって利用できます：FluxDistributionStepper, FluxDistributionProcess, QuasiDynamicFluxProcess。これらのProcessを用いることによって、反応速度式と速度論パラメータを用いず、連立質量保存式（mass balance equation）の解として流束分布を得ることができます。この方法によって、未知の要素を含む系のシミュレーションと解析を行うことができます。この理論とこれらのProcessの使い方に関しては、以下の学術論文で詳しく議論しています：

Yugi K, Nakayama Y, Kinoshita A, Tomita M., Hybrid dynamic/static method for large-scale simulation of metabolism. *Theor Biol Med Model*. 2005 Oct 4;2:42.

利用可能な連続クラス（抜粋）

E-Cellは常微分方程式（ODE）と微分代数系（DAE）の双方をサポートしており、それぞれのためのStepperを用意しています。システムにはいくつかの連続Processクラスも含まれています。たとえば、MassActionFluxProcess は質量作用則（law of mass action）に基づく反応速度を計算します。ExpressionFluxProcessでは、ユーザが任意の速度式をモデルファイルに書くことができます。PythonProcess と PythonFluxProcess は、Process オブジェクトを Python スクリプトで操作するた

めに用います。特定の酵素反応速度式に特化した Process もいくつか用意されています。

一般的な常微分 Stepper

モデルが常微分方程式系であるなら、現バージョンの E-Cell では ODEStepper の利用を推奨します。この Stepper は、系の状態に応じて数値積分アルゴリズムを切り替えます（陽的な Dormand-Prince 法と陰的方法の Radau IIA 法）。
この他の ODE Stepper として、より低次（2次）の積分機構を搭載した ODE23Stepper、もっともシンプルなオイラー法を実装し、適応的なステップ幅調節機構を持たない FixedODE1Stepper があります。
FixedODE1Stepper を除き、これらの ODE Stepper には、ユーザが設定できるいくつかの共通の属性があります。以下はその一部です：

`Tolerance`

局所的な打ち切り誤差に対するエラー耐性。この値を小さくすると、Stepper のステップ幅が小さくなり、シミュレーションが遅くなります。値を大きくすると、精度を犠牲にして計算速度が速くなります。標準的な値は 1e-6 です。

`MinStepInterval`

ステップ幅の最小値です。この値は、Tolerance より優先されます。

`MaxStepInterval`

この属性はすでにサポートされていませんので、設定してもシステムに影響を与えません。

MassActionFluxProcess

MassActionFluxProcess は、単純な質量作用則（law of mass action）のための Processクラスです。このクラスは非可逆の質量作用則に従って反応速度を計算します。速度定数を設定するには、属性 k を用います。

ExpressionFluxProcess

ExpressionFluxProcess は以下の 2 点を除き、ExpressionAssignmentProcess と同じです。(1) Expression 属性には、反応速度式を書きます。反応速度は [個数／秒] で与えなければなりません（[濃度／秒]ではないので注意してください）。(2) Expression 属性の値（式の計算結果）は、暗黙に setFlux() メソッドに渡されたものとして解釈され、Variable の Velocity に各VariableReferenceの Coefficientを乗じて受け渡されます。

ExpressionFluxProcess は、PythonProcess、PythonFluxProcess より高速に動作します。

以下に ExpressionFluxProcess の使用例を示します：

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

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

基本的なMichaelis-Menten反応をExpressionFluxProcessでプログラムした例を以下に示します。

サンプルコード 3-3. ExpressionFluxProcess によるMichaelis-Menten反応

```
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 ];
}
```

定義済みの反応速度クラス

特定の酵素反応速度式を実装したProcessについては、7章「標準ダイナミックモジュールライブラリ」をご覧ください。

PythonFluxProcess

PythonFluxProcess は以下の 2 点を除き、PythonProcess と同じです。(1) Expression 属性には、1 行のPython式を書きます。(2) ExpressionFluxProcess のように、暗黙に setFlux() メソッドに渡されたものとして解釈されます。

一般的な微分代数系 Stepper

微分代数系（DAE）モデルには、DAEStepper を用います。モデルは index 1[1]の DAE でなければなりません。DAEStepperは、１つあるいはそれ以上の離散Processオブジェクトを見つけると、それらを代数Processオブジェクトとみなします。したがって、DAEStepperと結合するすべての離散Processは代数を表現していなければなりません。代数Processの定義については以下を読んでください。

> **DAEStepperを常微分系（ODE系）に用いることができます**：なぜなら ODE は代数方程式を持たない DAE 問題の特殊型としてみることができるからです。DAEStepperは ODE モデルを実行することができます。ただし、モデルが ODE の場合には、積分手法や実装において ODE 問題に特化している ODE Stepper を用いた方が優れたパフォーマンスを発揮します。

ODE Stepper クラスの属性（Tolerance属性など）は、DAEStepper にも備わっています。

代数 Process（Algebraic Process）

これは離散Processのひとつですが、連続Stepperである DAEStepper とともに用いられるのでここで記述します。

原則として、連続Processオブジェクトはつねに連続Stepperインスタンスに結合されていなければならず、離散Stepperは離散Processオブジェクトだけを結合すると考えられます。ただし、例外があり、そのひとつが代数Processです。奇妙ではありますが、DAE のシミュレーションでは、離散的な代数方程式はその他の微分方程式と連動して連続的に解かれます。

E-Cellにおける代数方程式の記述形式は以下の通りです：

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

ただし、t は時刻、x は VariableReference のベクトルです。

E-CellのDAEソルバは、代数関数 $g(t,x)$ の値は、ProcessオブジェクトのActivity属性に格納されているものとして計算するよう設計されています。代数Processは Variableの値を陽的に変更することはありません。DAEStepperは、式 g() が表現されている場所を見いだす役割を果たしています。

モデルを作る際には、代数ProcessのVariableReferencesの係数（Coefficient）

---

[1] index とは、大ざっぱにいって、ある DAE をそれと等価な ODE に変形するために必要な最小の微分回数です。ODE は、index 0 の DAE といえます。index 1 の系とは、DAE 中の代数方程式の解を微分方程式に代入することで ODE になるような系です。

に気を配ってください。多くの場合、単純に 1 と設定します。例えば、A の係数がゼロだったとすると、式の計算に A の値を利用することはできますが、式の解によって A の指すVariableの値が変更されることはありません。

代数方程式を記述する方法として、ExpressionAlgebraicProcess を利用できます。式の評価結果が代数関数 $g()$ の値として解釈される点を除いて、使い方は ExpressionAssignmentProcess、ExpressionFluxProcess と同じです。
下のサンプルコード3-4は、以下の式を記述しています。

$$aA + B = 10, \quad a = 1.5$$

サンプルコード 3-4. ExpressionAlgebraicProcess>を用いた簡単な代数式

```
Stepper DAEStepper( DAE1 ) {}

Process ExpressionFluxProcess( P )
{
    StepperID DAE1;

    a 1.5;

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

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

C++ あるいは PythonProcess を代数式に用いるには、式の値を Activity属性にセットするために `setActivity()` メソッドを呼びだします。PythonProcess による例を以下に示します：

サンプルコード 3-5. PythonProcessを用いた簡単な代数式

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

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

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

Power-law（べき乗則）の正規形微分方程式[2]（S-System、GMA）

ESSYNSStepperは、ESSYNSアルゴリズムを用いたS-SystemおよびGMAのシミュレーションをサポートします。ESSYNSStepperは、単一のSSystemProcessあるいはGMAProcessと結合していなければなりません（Processの参照するVariableReferenceと、StepperがProcessを介して参照するVariableReferenceが一致していなければなりません）。SSystemMatrix属性あるいはGMAMatrix属性を用いてパラメータを設定します。
これらのProcessはどちらも、power-law（べき乗則）の正規形微分方程式によってモデル化されたさまざまな細胞現象をシミュレートすることができます。S-SystemとGMA（Generalized Mass Action）は、それぞれ、SSystemMatrix属性、GMAMatrix属性として、S-Systemあるいは GMA座標の行列を持ちます。サンプルモデルssystemがこのアルゴリズムを用いた例になっています（Appendix-2を参照してください）。
これらのモジュールは現在開発中です。

モデル化に際しての変換

単位

E-Cell SEでは以下の単位を用いています。この標準単位はシミュレータ内部での表現を意味するだけで、モデル化の過程ではどんな単位を用いることもできます。ただし、標準と異なる単位を用いた場合には、シミュレータに読み込む前に標準単位に変換しておかなければ成りません。

時間

s（秒）

容積

L（リットル）

濃度

モル濃度（M、mol/L（リットル）、VariableオブジェクトのMolarConc属性に相当）あるいは、個数濃度（個数/L（リットル）、VariableのNumberConc属性がこの単位を持つ）。

[2] 正規型の微分方程式とは $dy/dx = f(x, y)$ と書けるものをいいます。

モデル作成のチュートリアル

4

本章は、E-Cellを用いた簡単なモデル作成のチュートリアルです。

モデルの実行

本章のすべてのサンプルコードは、以下の手順で実行できます。

1. エディタでモデルファイルを作成し、保存します（simple-gillespie.emなどとして）。
   emacsなど、UNIX系のエディタの利用をお勧めします。Macintosh、Windowsなどのエディタを用いた場合、改行コードをLFとして保存してください（UNIXのファイル形式）。

2. ecell3-em2eml コマンドで、EMファイルをEMLファイルに変換します。

```
$ ecell3-em2eml simple-gillespie.em
```

3. ecell-session-monitor コマンドを用いて GUIモードでモデルを読み込み、シミュレーションを実行します。

```
$ ecell-session-monitor -f simple-gillespie.eml
```

   あるいは、ecell3-session コマンドを用いてスクリプトモードで実行します（5章をご覧ください）。

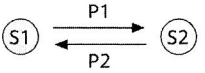
```
$ ecell3-session -f simple-gillespie.eml
```

Gillespie アルゴリズムを使う

E-Cell SE は Gillespie の確率論アルゴリズムを用いたシミュレーションのためのクラスを備えています。

ちいさな反応系

まず最初に、2つのVariable（ここでは、各分子種の分子の個数）と2つの素反応Processからなる、もっとも単純で安定な素反応系からはじめましょう。素反応は不可逆なので、反応系が安定になるためには少なくとも2つの反応インスタンスが必要です。反応系は以下のようになります：



S1とS2は分子種、そしてP1とP2は反応過程です。2つの反応の速度定数は等しく、1.0 [s^{-1}]です。S1とS2の初期値は、それぞれ1000、0です。速度定数が等しいので、この系はS1 = S2 = 500で定常となります。

Next Reaction methodの設定

DiscreteEventStepperクラスは、GillespieアルゴリズムのGibsonによる効率的な変法であるNext Reaction (NR) methodを実装しています。
シミュレーションモデル中でDiscreteEventStepperを用いるには、EMファイルに以下のように記述します：

```
Stepper DiscreteEventStepper( DS1 )
{
    # no property
}
```

この例では、DiscreteEventStepperはStepperID「DS1」を与えられています。今のところ、このオブジェクトの属性を指定する必要はありません。

コンパートメント（区画）の定義

次に、コンパートメント（区画）を定義し、それらをStepper DS1に結合します。このモデルは1つのコンパートメントしか持たないので、ルートシステム（/）を使うことにします。すべての反応が一次反応であるためこのモデルはコンパートメントの容積の影響を受けませんが、明示的にSIZEを定義しておくことは、定義せずにデフォルト値1.0のままにしておくよりもよい考えです。
ここでは、1e-15 [L]に設定しておくことにします。

```
System System( / )
{
    StepperID DS1;

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

    # ...
}
```

変数（Variable）の定義

次に、Variable S1とS2を定義します。オブジェクトの初期値を設定するためには、「Value」属性を用います。

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

    Variable Variable( S1 )
    {
        Value 1000;
    }

    Variable Variable( S2 )
    {
        Value 0;
    }

    # ...
}
```

反応過程（Process）の定義

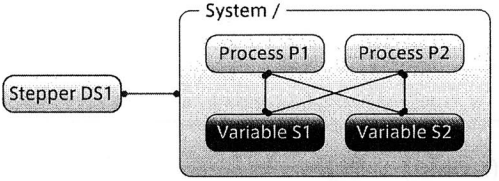
最後に、反応過程のインスタンスであるProcess P1 と P2 を作成します。GillespieProcess クラスを DiscreteEventStepper と組み合わせることで、素反応をシミュレートします。

２つの異なる型の属性（k および VariableReferenceList）をそれぞれのGillespieProcessオブジェクトについて設定しなければなりません。k は速度定数パラーメータで、単位は、一次反応の場合 [s^{-1}]、二次反応では [$s^{-1}M^{-1}$] です（二次反応が存在する場合、SIZE Variableの定義をお忘れなく）。VariableReferenceList属性には、P1はS1を消費してS2を産生し、P2はS2を用いてS1を産生するように設定しましょう。

```
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;
    }
}
```

つなぎ合わせる

このモデルに含まれるオブジェクトの関係図と、実行可能な完全なEMファイルを以下に示します。



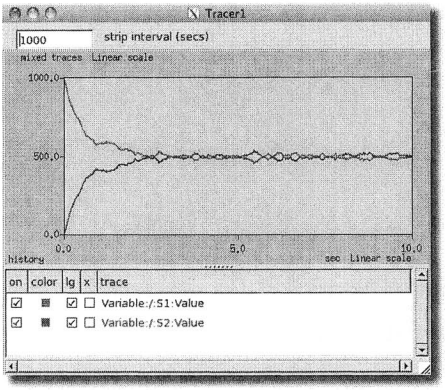
特に指定しない場合、Process の Stepper は Process が結合する System の Stepper に設定されます。

サンプルコード 4-1. もっとも簡単なGillespie-Gibsonモデル

```
Stepper DiscreteEventStepper( DS1 )
{
    # no property
}

System System( / )
{
    StepperID DS1;

    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;
    }
}
```

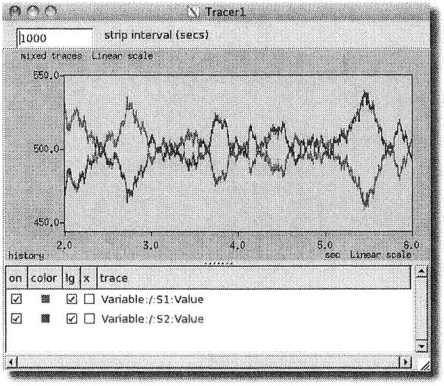
このモデルを**セッションモニタ**（ecell3-session-monitor）で実行し、S1とS2の軌跡をプロットするためにトレーサーウィンドウを開いてみてください。2つのVariableが速やかに定常状態である500.0付近に到達することがわかるでしょう（セッションモニタの使用法は Appendix-4 をご覧ください。）。

以下に、セッションモニタで観察した例を示します。



Variable S1と、S2を
トレーサーに登録し、10秒の
シミュレーションを実行した
状態。
ほぼ定常状態に達しています。

トレースを拡大してみると、確率的ゆらぎを見ることができます。



2〜6 秒のトレースを拡大し
てみると、確率的ゆらぎのた
め、ぴったり 500 で一定には
ならない様子がわかります。

決定論的微分方程式を使う

前節では確率論的Gillespieアルゴリズムを用いて実行するモデルの作り方を示しました。E-Cellは複数アルゴリズムシミュレータなので、別のアルゴリズムをこのモデルに用いてシミュレーションを行うこともできます。本節では、シンプルな質量作用則（mass-action）反応系のシミュレーションに決定論的微分方程式ソルバを用いる方法を解説します。

Stepper と Process クラスの選択

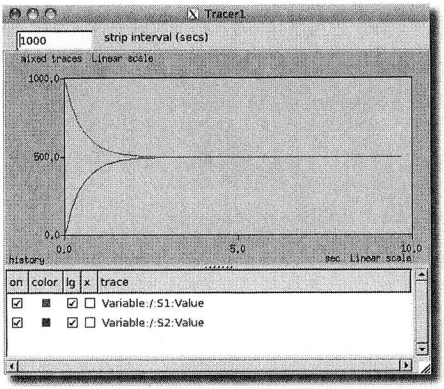
現バージョンでは、微分方程式系をシミュレートするための汎用 Stepper として、ODEStepper の利用を推奨します。
MassActionFluxProcess は、連続な微分方程式 Process で、離散イベントにおけ

る GillespieProcess に相当します。GillespieProcess と異なり、MassActionFluxProcess では反応機構の次数に制限はありません。例えば、次のような反応も取り扱うことができます：S0 + S1 + 2 S2 → P0 + P1

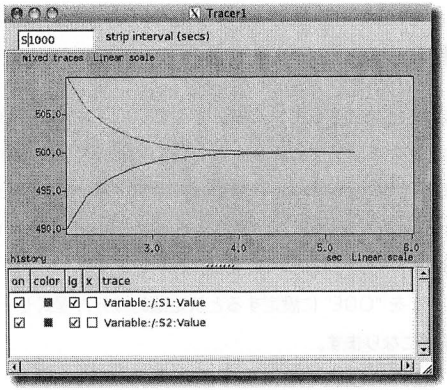
モデルの変換

Gillespie アルゴリズムのちいさな反応系モデルを、微分方程式モデルに変換する方法はとてもシンプルで、DiscreteEventStepper を ODEStepper に置き換え、Processのクラス名を GillespieProcess から MassActionFluxProcess に書き換えるだけです。

微分 ODEStepper で実行されるちいさなモデルを以下に示します。前節の確率論モデルと似通ったシミュレーション結果が得られます。ただし、Variableの変動を拡大してみると、もはや確率的ゆらぎが見られないのがわかります。確率論モデルから決定論モデルに変更されたからです。



確率論モデル同様、S1と、S2をトレーサーに登録し、10秒のシミュレーションを実行した状態です。
定常状態に達しています。



やはり同様に、2〜6 秒のトレースを拡大してみると、確率的ゆらぎは見られず、S1 = S2 = 500 の定常状態に収束しています。

トレースの右端が途切れたように描画されているのは、現バージョンの仕様です。

サンプルコード 4-2. 簡単な決定論mass-actionモデル

```
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;
    }
}
```

複数のアルゴリズムを切り替えられるモデルを作る

モデルが素反応だけから構成されている場合などでは、DiscreteEventStepper と GillespieProcess のペア、ODEStepper と MassActionFluxProcess のペアのクラス名を切り替えるだけで、確率論と決定論のアルゴリズムを切り替えることができます。どちらのProcessクラスも、同じ属性 k を、同じ単位で持っています。

EmPy マクロを用いて、モデルを汎用化することができます。以下の例では、Pythonの変数 TYPE を "ODE" に設定すると決定論的微分方程式モードに、"NR" に設定すると確率論的になります。

サンプルコード 4-3. simple-switchable.em

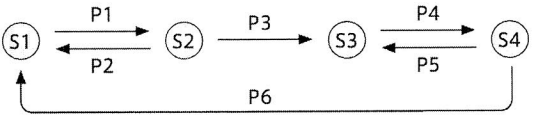
```
@{ALGORITHM='ODE'}
@{
if ALGORITHM == 'ODE':
    STEPPER='ODEStepper'
    PROCESS='MassActionFluxProcess'
elif ALGORITHM == 'NR':
    STEPPER='DiscreteEventStepper'
    PROCESS='GillespieProcess'
else:
    raise 'unknown algorithm: %s' % ALGORITHM
}

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;
    }
}
```

簡単な決定論／確率論 連成シミュレーション

E-Cell は複数の Stepperオブジェクトを持つモデルを実行できます。それぞれの Stepper に、異なるシミュレーションアルゴリズム、異なる時間スケールを実装することができます。この複数アルゴリズム、複数時間スケールシミュレーションのフレームワークは、どんな数のどんなに異なる下位モデルの組み合わせも許す、真に一般的なモデル化とシミュレーションを実現します。本節ではその簡単な例として、ODE と Gillespie 反応を組み合わせた（連成した）モデルを示します。

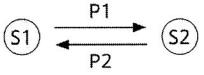
ちいさな複数タイムスケール反応モデル

下に示す４つのVariableと６つの反応Processからなるちいさなモデルを考えます：

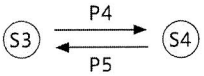


一見複雑そうですが、この系は、前出の２つのちいさなモデルを下位モデルとしてつなぎ合わせただけです：

下位モデル１

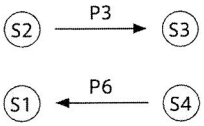


下位モデル２



これら２つの下位モデルは、反応Process P3 と P6 で結合されています。P3 と P6 の時間スケールは、それぞれ S2 と S4 によって決まるので、P3 は下位モデル１に、P6 は下位モデル２に属しているといえます。

下位モデル１：



：下位モデル２

主反応 P1、P2、P4、P5 の速度定数は前出のモデルと同じ 1.0 [sec^{-1}] とします。一方で、下位モデルをつなぐ「ブリッジ」反応は主反応よりも遅いものとし、P3 は 1e-1、P6 は 1e-3 とします。その結果、下位モデル１と２の定常状態の物質濃度には、おおよそ 1e-1 / 1e-3 = 1e-2 倍の差が生じます。ブリッジ反応の速度差による時間スケールの違いが原因であると考えられます。

モデルファイルを書く

以下のコードは、複数時間スケールを実装しています。最初の2行では、モデルの2つの部分で用いるアルゴリズムを指定しています。変数 ALGORITHM1 は、下位モデル1に、ALGORITHM2 は、下位モデル2に用いるアルゴリズムです。どちらの変数も、"NR" または "ODE" の値をとります。
たとえば、純粋な確率論的シミュレーションを試行したいなら以下のように変数を設定します：

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

ALGORITHM1 を "NR" に、ALGORITHM2 を "ODE" にセットするのが理想的な設定です。確率論だけに設定した場合より、はるかに高速にシミュレーションを実行します。

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

純粋に決定論的なシミュレーションも試してみましょう。

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

このモデルに関していえば、この設定は非常に高速です。これは、この系が非常に早く定常状態に到達し、モデルの stiffness も低いためです。ただし、このことは必ずしも、純粋な ODE がつねに最速であることを意味しません。条件によっては、NR と ODE の連成シミュレーションが、確率論のみ、あるいは決定論のみのどちらよりも高速になります。

サンプルコード 4-4. composite.em

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

# a function to give appropriate class names.
@{
def getClassNamesByAlgorithm( anAlgorithm ):
    if anAlgorithm == 'ODE':
        return 'ODEStepper', 'MassActionFluxProcess'
    elif anAlgorithm == 'NR':
```

```
        return 'DiscreteEventStepper', '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;
    }
}
```

方程式のカスタマイズ

複雑な反応速度式

独自の反応速度式を表現するもっとも簡単な方法は、ExpressionFluxProcess を用いることです。以下に、ショウジョウバエのサンプルモデル（Drosophila、Appendix-2 参照）から一例を示します。この例では、Processの上位SystemのSizeN_A（Size × N_A）を用いて、式の単位を [個数／秒] に保っています。

サンプルコード 4-5. ショウジョウバエモデル内の微分方程式の一例

```
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 ];
}
```

代数方程式

代数方程式を記述するためのもっとも簡単な方法は、ExpressionAlgebraicProcess を使うことです。VariableReference の係数（Coefficient属性）に注意してください（通常は 1 に設定します）。

スクリプトによるセッションの操作

5

本章は以下の項目について書かれています。

- E-Cellセッションスクリプト（ESS）とは
- スクリプトモードでESSを実行する方法
- GUIモードでESSを実行する方法
- ESSファイルを書くことでシミュレーションを自動化する方法
- PythonでE-Cellのフロントエンドソフトウェアを書く方法

セッションをスクリプトで操作する

E-Cell セッションスクリプト（ESS）は、E-Cell Session オブジェクトによって読み込まれる Python スクリプトです。Session インスタンスは、1回のシミュレーションセッションの実行を担っています。
ESS は、1回のセッションの実行（およびその前後の処理）を自動化するために用います。シンプルかつ典型的なセッションには、以下の5つの段階が含まれています。

1. **モデルファイルの読み込み（load）**
   通常、EMLファイルを読み込みます。
2. **シミュレーションに先立つシミュレータの設定**
   シミュレータとモデルパラメータ（Variableオブジェクトの初期値、Processオブジェクトの属性値など）を、設定あるいは変更します。データLogger（ロガー、記録器）もこの段階で作成します。
3. **シミュレーションの実行**
   シミュレーションが、ある時間実行されます。
4. **シミュレーション後のデータ処理**
   この段階では、シミュレーション後のモデルの状態や、Logger オブジェクトによって記録されたデータを検査します。シミュレーション結果を数値処理する場合もあります。必要に応じて、前の段階に戻り、さらにシミュレーションを実行します。

5. **データの保存**

最後に、処理済みのあるいは生のシミュレーション結果のデータをファイルに保存します。

ESS ファイルの拡張子は通常「.py」です。

E-Cell セッションスクリプトの実行

ESS を実行するには３つのやり方があります；

- スクリプトをコマンドライン（シェルプロンプト）から実行する方法。
- ecell3-session-monitor のようなフロントエンドソフトウェアからスクリプトを読み込む方法。
- シミュレーションセッションそのものの実行を自動化するためにセッションマネージャ（ecell3-session-manager）を使う方法。この方法は通常、パラメータ最適化など、複数のシミュレータの実行を含む数理解析スクリプトを書く場合に用いられます。

コマンドラインでのESSの実行

ecell3-session コマンドを用いて、ESS を、バッチモードあるいはインタラクティヴモードで実行することができます。

バッチモード

ユーザによる操作を求めずに ESS を実行するには、シェルプロンプトで以下のコマンドを入力します：

```
$ ecell3-session [-f model.eml ] [-e]ess.py
```

ecell3-sessionコマンドは、シミュレーション Session オブジェクトを作成し、ESS ファイル *ess.py* をそのオブジェクト上で実行します。オプション [-e] は省略できます。[-f *model.eml*] オプションが指定されている場合には、ESS の実行に先立って、直ちに EML ファイル *model.eml* が読み込まれます。

インタラクティヴモード

ecell3-session をインタラクティヴモードで実行するには、ESS ファイルを指定せずにコマンドを実行します。

```
$ ecell3-session [-f model.eml]
ecell3-session [ E-Cell SE Version 3.1.106, on Python Version 2.4.3 ]
Copyright (C) 1996-2008 Keio University.
Copyright (C) 2005-2008 The Molecular Sciences Institute.
More info: http://www.e-cell.org/software
ecell3-session>>>
```

表示されるバナーとプロンプトは、E-Cell のバージョンによって大きく異なります。オプション [-f] *model.eml* が指定されている場合には、プロンプトの表示に先立って、直ちに EML ファイル *model.eml* が読み込まれます。

スクリプトへのパラメータの受け渡し

セッションパラメータをスクリプトに引き渡すこともできます。引き渡したセッションパラメータは、グローバル変数として ESS スクリプトからアクセスできます（以下の節をご覧ください）。

ecell3-session コマンドから ESS パラメータを引き渡すには、-D オプションあるいは --parameters= オプションを用います。

```
$ ecell3-session -DNAME1=VALUE1 -DNAME2=VALUE2...
$ ecell3-session --parameters="{'NAME1':VALUE1,'NAME2':VALUE2,..}"
```

-D オプションと --parameters= オプションを混在させても問題ありません。

ecell3-session-monitor での ESS の読み込み

ESS ファイルを GUI から手動で読み込むには、メニューから File → loadScript を選びます。

ecell3-session-monitor コマンドは、ecell3-session コマンド同様、-e および -f オプションを受け付けます。

セッションマネージャを使う

E-Cell セッションマネージャを用いると、E-Cell の SessionManager クラスを利用して複数のシミュレーションセッションをスクリプトすることができます。SessionManager クラスは、単一プロセッサのワークステーション、複数プロセッサの共有メモリ（SMP）マシン、ワークステーションクラスタ、グリッドコンピューティング環境などでジョブを実行します。

ecell3-session-manager を用いて、複数のシミュレーションのセットを実行することができます。ecell3-session-manager コマンドは、インタラクティヴ

モードあるいは、-e オプションで .ems スクリプトファイルを指定することで実行できます。

単一プロセッサワークステーション、複数プロセッサワークステーション、クラスターまたはグリッド上のコンピュータにジョブを割り当てる SessionManger の機能にアクセスするには、--environment および --concurrency オプションを用います。--environment オプションは、ローカル、SGE、Globus2 のいずれかの値をとることができ、デフォルト値はローカルです。--concurrency オプションは正の整数をとり、デフォルト値は 1 です。

E-Cell セッションマネージャを実行するには、(1) モデルファイル、(2) セッションスクリプトファイル、(3) セッションマネージャスクリプトファイル の 3 種類のファイルが必要です。典型的には、EMS 中で、ジョブは、実行に必要なセッションスクリプトの引数、オプションのパラメータ、スクリプトの入力ファイルとともに registerEcellSession() メソッドを用いて登録され、実行されます。

以下の例では、1 つの ESS ファイルに、100 通りの異なる Variable:/:S の値を渡して条件の異なる複数のジョブ構成し、計算環境に投入する EMS を示します。

```
MODEL_FILE='model.eml'
ESS_FILE=runsession.py

# Register Jobs
aJobIDList = []
for S_VALUE in range(0,100):
  aParamDict = {'MODEL_FILE':MODEL_FILE, 'VALUE_OF_S':S_VALUE}

  # registerEcellSession(theScript, parameters,
  #                                        files that the ESS uses)
  aJobID = registerEcellSession(ESS_FILE, aParamDict, [MODEL_FILE])
  aJobIDList.append(aJobID)

# Run the registered jobs
run()

#Examine the results
for aJobID in aJobIDList:
  print getStdout(aJobID)
```

以下は、これに対応する ESS ファイルです：

```
loadModel( MODEL_FILE )
S = createEntityStub('Variable:/:S')
S['Value'] = VALUE_OF_S
run(200)
message(S['Value'])
```

E-Cellセッションスクリプトを書く

ESS の文法は Python 言語そのものです。Python の機能をフル活用できる上、いくつかの便利な機能が加わっています。

Session メソッドを使う

一般的なルール

ESS では、1 つの Session インスタンスが与えられており、このクラスの持つメソッドのうち、グローバル名前空間に定義されているものをすべて使うことができます。
例えば、10 秒間のシミュレーションを実行するには、Session オブジェクトに備わっている run() メソッドを用います。

```
self.run( 10 )
```

ここで、self はシステムによって与えられている現在の Session オブジェクトです。self の代わりに、theSession を用いることもできます。

```
theSession.run( 10 )
```

通常の Python スクリプトと異なり、現在の Session に対してメソッドを呼びだす際には、オブジェクトを省略することができます。

```
run( 10 )
```

モデルの読み込み

ecell3-session コマンドのインタラクティヴモードで loadModel() メソッドを実行する（プロンプトに入力する）と、EML ファイルが読み込まれます。

```
ecell3-session>>> loadModel( 'simple.eml' )
```

EML ファイルが読み込まれると、プロンプトが「ecell3-session>>>」から「[モデル名], t=[現在時刻]>>>」に変化します。

```
simple.eml, t=0>>>
```

シミュレーションの実行

シミュレーションを実行して時刻を進めるには、step() および run() メソッドを用います。

```
simple.eml, t=0>>> step()
simple.eml, t=0>>> step()
simple.eml, t=7.67306e-07>>> run( 10 )
simple.eml, t=10.0032>>>
```

step(*n*) によって *n* ステップのシミュレーションが実行されます。 *n* のデフォルト値は 1 です。

> ノート：上の例で、step() を最初に呼びだした際に、時刻が変化しないことにお気づきかもしれません。シミュレータは、ステップの開始時点で時刻を更新し、その後、一時的なステップ幅を計算します。ステップ幅の初期値はゼロです。そのため、時刻を進めるためには step() を 2 度呼びだす必要があります。シミュレーション機構の詳細については 6 章をご覧ください。

シミュレーションをある時間分進めるには、run() メソッドを呼びだして、実行時間を秒単位で引き渡してください。例えば、run(10) では 10 秒分のシミュレーションを実行します。run() メソッドはステップを繰り返し実行し、与えられた秒数が過ぎるまでシミュレーションを進行させます。換言すれば、run(10) はシミュレーションを少なくとも 10 秒間進めます。このメソッドは、つねに指定された実行時間をやや超過しますが、超過する長さは、step() メソッドのステップ幅よりも小さくなります。

引数を渡さずに run() を実行することもできます。この場合、イベントチェッカ、イベントハンドラが設定されていないと停止することなくシミュレーションを実行しつづけ、継続に支障を来すと例外を発生します。Session クラスのメソッドリストにある setEventChecker() をご覧ください。

現時刻の取得

シミュレータの現在時刻を取得するには、getCurrentTime() メソッドを用います。

```
simple.eml, t=10.0032>>> getCurrentTime()
10.003221620379463
```

> **モデル中で時刻を使うには**：E-Cell では、Process に記述された生命現象モデルがシミュレータの現在時刻を取得するためのメソッドを用意していません。これは、分子な

ど細胞内の要素は、直接時間を計測することはできないという事実に基づく仕様です。時間を計測しているのは、細胞ではなく計測者であり、計測者は、絶対的な時刻を知っているのではなく、計測者の用意した機器で、細胞の観察と平行して時刻を計測しています。モデル中で時刻を用いるには、時刻を計測するための「時計モデル」を作成しておくといった方法があります（下に例を挙げます）。また、C++で独自の Process を書けば、getCurrentTime() で得られる値と同じ現在時刻を取得するプログラムは書くこともできます。

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

Process ExpressionFluxProcess( stopwatch )
    {
        VariableReferenceList [ time :.:time 1 ];
        expression "1.0";
    }
```

この例では、単純な微分方程式 dx/dt = 1.0 [sec^{-1}] を記述した ExpressionFluxProcess によって時刻を表現する Variable time を積分することで、time に時刻を保持させています。シミュレータの現在時刻はセッション毎にゼロにリセットされますが、モデル中にこうした仕組みを持たせることで、簡単に、複数のセッションにわたってひとつの時間軸を持たせることもできます。

メッセージの表示

ESS 中でメッセージを出力したい場面があるかもしれません。その際には、message (*message*) メソッドを利用します。引数 *message* は出力される文字列です。デフォルトの設定では、message() は Python の print 文と同様に取り扱われ、標準出力に新しい行を出力します。setMessageMethod() メソッドを用いて、この振る舞いを変更することができます。

Session メソッドの例

以下に、Session メソッドを用いてモデルを読み込み、100 秒間のシミュレーションを実行し、短いメッセージを出力する簡単な例を示します。

サンプルコード 5-1. 簡単なESSの例

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

Session パラメータの取得

Session パラメータはグローバル変数として ESS に引き渡されます。ですから、Session パラメータの使い方はとても簡単です。例えば、Session パラメータ *MODELFILE* が与えられていると想定され、これを ESS 中で変数として使うには以下のように書きます：

```
loadModel( MODELFILE )
```

どういったパラメータが ESS に引き渡されているかを確認するには、組み込み関数 dir() あるいは globals() を用います。ある特定の Session パラメータまたはグローバル変数が引き渡されているかを確認するには、以下のような if 文を書きます：

```
if 'MODELFILE' in globals():
    # MODELFILE is given

else:
    # not given
```

ノート：現在、Session パラメータとグローバル変数を区別する方法はありません。

ObjectStub によるモデルの観察と操作

ObjectStub とは

ObjectStub はシステムのフロントエンドにおけるシミュレータの内部オブジェクトの「身代わり」オブジェクトです。シミュレータの内部オブジェクトに対するいかなる操作も、ObjectStub を通じて行います。
ObjectStub の型は、以下の 3 種類です。

- EntityStub
- StepperStub
- LoggerStub

これらはそれぞれ、シミュレータ中の Entity、Stepper、Logger クラスに相当します。

ObjectStub はなぜ必要か

ObjectStub クラスの実体は E-Cell Python ライブラリの ecell.emc.Simulator クラスに対する薄いラッパーです。これは、Simulator クラスのフラットな手続き型

API にオブジェクト指向の使い勝手を提供します。Session クラスの theSimulator 属性から Simulator オブジェクトに直接アクセスすることはできますが、ObjectStub の利用を強く推奨します。

このバックエンドとフロントエンドの分離が必要なのは、バックエンドオブジェクトの寿命が、フロントエンドオブジェクトの寿命と同じとは限らず、また状態遷移も必ずしも同期されていないからです。フロントエンドが直にシミュレータの内部オブジェクトを操作すると、オブジェクトの寿命や状態の一貫性が容易に破られます。これは起こってはならないことです。

ID から ObjectStub を作る

ObjectStub オブジェクトを取得するには、Session クラスメソッドの `createEntityStub()`、`createStepperStub()`、`createLoggerStub()` を用います。例えば、EntityStub を取得するには `createEntityStub()` メソッドに FullID 文字列を引き渡します：

```
anADPStub = createEntityStub( 'Variable:/CELL/MT1:ADP' )
```

同様に、StepperStub オブジェクト、LoggerStub オブジェクトも、それぞれ、StepperID、FullPN から取得することができます。

```
aStepperStub = createStepperStub( 'STEPPER_01' )
aLoggerStub = createLoggerStub( 'Variable:/CELL/
MT1:GLUCOSE:Concentration' )
```

ObjectStub の作成とバックエンドオブジェクトの確認

バックエンドに対応するオブジェクトが存在しなくても、ObjectStub は作成されます。つまり、ObjectStub の作成は純粋にフロントエンドの操作です。ObjectStub を作ったら、これに対応するバックエンドオブジェクトが存在するかを確認する、あるいはバックエンドオブジェクトを作成する必要があるかもしれません。

ObjectStub に対応するバックエンドオブジェクトの存在を確認するには、`exists()` メソッドを用います。例えば、以下の if 文は、STEPPER_01 という ID を持つ Stepper の存在を確認するものです。

```
aStepperStub = createStepperStub( 'STEPPER_01' )
if aStepperStub.exists():
    # it already exists
else:
    # it is not created yet
```

バックエンドオブジェクトを作成するには、create() メソッドを呼ぶだけです。

```
aStepperStub.create()
```

ObjectStub からの 名前、クラス名の取得

ObjectStub の名前（または ID）を取得するには、getName() メソッドを使います。

EntityStub あるいは StepperStub のクラス名を取得するには、getClassname() メソッドを呼びます。この操作は LoggerStub に対して行うことはできません。

属性の設定と取得

これまでも述べてきたように、Entity、Stepper オブジェクトは属性を持ちます。本節では ObjectStub を通してオブジェクトの属性にアクセスする方法を説明します。本節の内容は、LoggerStubs に対しては適用できません。EntityStub または StepperStub を用いてバックエンドオブジェクトから値を取得するには、getProperty() メソッドを実行するか、属性の名前を使ってオブジェクト属性にアクセスします。

```
aValue = aStub.getProperty( 'Activity' )
```

あるいは、

```
aValue = aStub[ 'Activity' ]
```

Entity または Stepper の属性に新たな値を設定するには、setProperty() メソッドを呼びだすか、属性の名前と新たな値でオブジェクト属性を変更します。

```
aStub.getProperty( 'Activity', aNewValue )
```

あるいは、

```
aStub[ 'Activity' ] = aNewValue
```

すべての属性のリストを取得するには、getPropertyList() メソッドを用います。これは、属性の名前のリストを、文字列を要素とする Python タプルで返します。

```
aStub.getPropertyList()
```

ある属性が読み込み可能（getable、accessible）または書き込み可能（settable、mutable）かを知るには、getPropertyAttributes() メソッドに属性の名前を引き渡します。これは、Python タプルを返します。属性が書き込み可能なら、タプルの最初の要素は true です。読み込み可能なら、2番目の要素が true です。読み込み不可の属性を取得しようとしたり、書き込み不可の属性を書き換えようとすると、例外が発生します。

```
aStub.getPropertyAttribute( 'Activity' )[0]
aStub.getPropertyAttribute( 'Activity' )[1]
```

Logger データの取得

LoggerStub から Logger に記録されたデータを取得するには、getData() メソッドを用います。getData() メソッドには、要求するデータ取得区間と時間解像度によって3つの書式があります：

getData()

全データを取得します。

getData(*starttime* [, *endtime*])

starttime から *endtime* までの区間のデータを取得します。 *endtime* が省略された場合、*starttime* から末尾までのデータが返されます。

getData(*starttime*, *endtime*, *interval*)

starttime から *endtime* までの区間のデータを取得します。保存されたデータポイントの間隔が *interval* より短ければ、間のデータは間引かれます。科学的なデータ分析には不適当ですが、高速です。

getData() メソッドは、Numeric Python モジュールの rank-2（行列）[3]の配列オブジェクトを返します。この配列は以下のいずれかの形式です：

```
[ [ time value average min max ]
  [ time value average min max ]
... ]
```

または、

```
[ [ time value ]
  [ time value ]
... ]
```

最初の例の 5-タプルのデータ形式では、1つのデータポイントが以下の5つの値を持っています：

time
: 時刻。

value
: 値。

average
: 直前のデータポイントからこのデータポイントまでの区間での値の平均（時間で加重した平均値）。

min
: 直前のデータポイントからこのデータポイントまでの区間での最小の値。

max
: 直前のデータポイントからこのデータポイントまでの区間での最大の値。

2-タプルのデータは time と value だけを持ちます。
データ取得に先立って、記録されたデータの開始時刻、終了時刻、サイズを知るには、LoggerStub の getStartTime()、getEndTime()、getSize() メソッドを用います。getSize() は、Logger に保存されているデータポイントの数を返します。

[3] rank は行列の階数のことです。階数とは行列の列ベクトルの一次独立な要素の最大個数を指します。ここでは、時刻と値の2つが一次独立なので rank は2となります。5-タプルのデータに含まれる average、min、max は time と value から求めたものなので、一次独立とはいえません。したがって、5-タプルですが、rank は2となります。

Logger による記録間隔の取得と変更

Logger の記録間隔を確認、変更するには、LoggerStub の `getMinimumInterval()`、`setMinimumInterval( interval )` メソッドを用います。*interval* は、秒を単位とするゼロまたは正の数値でなければなりません。*interval* が非ゼロの正の数の場合、Logger は、直前の記録時点から *interval* 秒が経過するまでデータを記録しません。*interval* がゼロの場合、Logger はすべてのシミュレーションステップでデータを記録します。

EntityStub の使用例

以下の例は、EML ファイルを読み込み、System /CELL 中の Variable ATP の値を 10 秒おきに出力します。値が 1000 以下になると、シミュレーションを停止します。

サンプルコード 5-2. ATP濃度を10秒毎に確認するESS

```
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() )
```

データファイルの操作

ECD ファイルについて

E-Cell SE は、シミュレーション結果の保存に ECD（E-Cell Data）ファイル形式を用いています。ECD は平文テキストファイルで、ユーザおよびサードパーティの書いたデータ処理ならびに視覚化ソフトウェア（gnuplot など）で容易に取り扱うことができます。

ECD ファイルには、浮動小数点数の行列を保存することができます。ecell.ECDDataFile クラスを使って、ECD ファイルを保存したり読み込んだりすることができます。ECDDataFile オブジェクトは、Numeric Python の rank-2 の配列を保持し、返します。rank-2 の配列とは、Numeric.rank(ARRAY) および len (Numeric.shape(ARRAY)) の戻り値が '2' であるような行列を意味します。

ECDDataFile クラスのインポート

ECDDataFile クラスをインポートするには、ecell モジュール全体をインポートするか、

```
import ecell
```

ecell.ECDDataFile モジュールを選択的にインポートします。

```
import ecell.ECDDataFile
```

データの保存と読み込み

データを（たとえば datafile.ecd という）ECD ファイルに保存するには、ECDDataFile オブジェクトをインスタンス化し、save() メソッドを使います。

```
import ecell
aDataFile = ecell.ECDDataFile( DATA )
aDataFile.save( 'datafile.ecd' )
```

ここで、DATA は Numeric Python の rank-2 配列もしくはそれと等価のオブジェクトです。インスタンス化の後、setData() メソッドでデータを書き込むこともできます。すでにデータが存在していたときには、上書きします。

```
aDataFile.setData( DATA )
```

ECD ファイルの読み込みも簡単です。

```
aDataFile = ecell.ECDDataFile()
aDataFile.load( 'datafile.ecd' )
DATA = aDataFile.getData()
```

getData() メソッドは、ECDDataFile オブジェクトから配列形式でデータを取り出します。

ECD のヘッダに含まれる情報

データそのものに加え、ECD ファイルはいくつかの情報をヘッダに保持しています。

DATA：データ名

データの名前。FullPN を書き込んでおくとよいかもしれません。setDataName(*name*)、getDataName() メソッドを用いてこのフィールドを読み書きできます。

LABEL：ラベル

データの軸の名前です。setLabel(*labels*)、getLabel() メソッドで読み書きできます。これらのメソッドの引数、戻り値は Python タプルで、ECD ファイル中ではスペース区切りのリストとして保存されています。デフォルト値は：('t', 'value', 'avg', 'min', 'max') です。

NOTE：ノート

自由書式のフィールドです。複数行または 1 行の文字列です。setNote(*note*)、getNote() メソッドで読み書きできます。

ヘッダ情報は以下のように保存されています。

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

ヘッダの各行はシャープ（#）で始まっています。「#SIZE:」で始まる行は、データのサイズを示しており、ファイルが保存される際に自動的に作成され、読み込まれる際には無視されます。ヘッダは、「#----...」の行で終わります。

E-Cell SE の外部で ECD を利用する

Numeric Python は、多くの科学的データ処理に必要な機能を備えています。一方で、外部ソフトウェアを用いることで利便性が向上する場合もあります。

スペース区切りテキストをサポートし、行頭のシャープ（#）をコメントとして扱うことができるソフトウェアであれば、ECD ファイルを取り扱うことができます。GNU gnuplot は、洗練されたインタラクティヴなコマンドシステムを備えた科学的プレゼンテーションに十分な品質のグラフ描画ソフトウェアです。gnuplot を用いて ECD ファイルをプロットするには、plot コマンドを使うだけです。例えば、以下は、時間-値の 2 次元グラフを描画するコマンドの例です：

```
gnuplot> plot 'datafile.ecd' with lines
```

どの列をプロットするかを指定するにはモディファイア（modifier）を利用します。以下の例は、時間-平均値の 2 次元プロットを作成します。

```
gnuplot> plot 'datafile.ecd' using 1:3 with lines
```

データ処理に役立つオープンソースのソフトウェアとして、この他に GNU Octave があります。Octave から ECD ファイルを読み込むのも、とても簡単です。

```
octave:1> load datafile.ecd
```

この操作で、データは、ファイル名から拡張子を除いた名前（ここでは datafile）を持つ行列変数に格納されました。

```
octave:2> mean(datafile)
ans =
      5.0663 51.7158 51.7158 51.2396 52.2386
```

バイナリ形式

現在、E-Cell はバイナリ形式での読み込み、保存はサポートしていません。ただし、Numeric Python には、プラットフォーム依存の効率的な配列データの書き出し（export）、取り込み（import）手段が備わっています。

モデルファイルの操作

ここでは、E-Cell Python ライブラリの EML モジュールを用いて EML ファイルの作成、変更、読み込みをおこなう方法を説明します。

EML モジュールのインポート

ecell モジュールを読み込むだけで、EML モジュールが読み込まれます。

```
import ecell
```

これで ecell.Eml クラスを利用できるようになります。

その他のメソッド

バージョン番号の取得

ecell.ecs モジュールの getLibECSVersion() メソッドを用いて、C++バックエンドライブラリ（libecs）のバージョンを文字列で取得できます。このモジュールの getLibECSVersionInfo() メソッドは Python タプルを返します。タプルは（メジャーバージョン、マイナーバージョン、マイクロバージョン）の順で 3 つの数値を持っています。

```
ecell3-session>>> import ecell
ecell3-session>>> ecell.ecs.getLibECSVersion()
'3.1.106'
ecell3-session>>> ecell.ecs.getLibECSVersionInfo()
(3, 1, 106)
```

ダイナミックモジュール（DM）読み込みに関連するメソッド

setDMSearchPath()、getDMSearchPath() メソッドで DM ファイルの探索パスを設定あるいは取得することができます。これらのメソッドは、ディレクトリ名のリストをコロン（:）区切りで取得したり、返したりします。探索パスは ECELL3_DM_PATH 環境変数を用いて指定することもできます。DM 探索パスに関しては、2 章の「DM 探索パスと環境変数 ECELL3_DM_PATH」で詳しく述べています。

```
ecell3-session>>> import ecell
ecell3-session>>> ecell.ecs.setDMSearchPath( '~/dm:~/test/dm' )
ecell3-session>>> ecell.ecs.getDMSearchPath()
'~/dm:~/test/dm'
```

ecell.emc.Simulator クラスの getDMInfo() メソッドを用いて、組み込み済みおよびすでに読み込まれた DM クラスのリストを取得できます。Simulator インスタンスは、Session の theSimulator 変数として利用できます。このメソッドは、((TYPE1, CLASSNAME1, PATH1), (TYPE2, CLASSNAME2, PATH2), ...) という形式のネストされた Python タプルを返します。TYPE は、'Process'、'Variable'、'System'、'Stepper' のいずれかです。CLASSNAME は DM のクラス名です。PATH は DM が読み込まれたディレクトリです。組み込みクラスの場合、PATH は空の文字列 ('') です。

```
ecell3-session>>> theSimulator.getDMInfo()
(('Process', 'GillespieProcess', '/usr/lib/ecell/3.2/
GillespieProcess.so'),
('Stepper', 'DiscreteTimeStepper', ''),
('Stepper', 'NRStepper', '/usr/lib/ecell/3.2/NRStepper.so'), ... )
```

上級者向けの話題

ecell3-session の実行機構

ecell3-session コマンドは、ecell3-python インタプリタコマンド上で実行されます。ecell3-python コマンドは、Python インタプリタへの薄いラッパーです。ecell3-python コマンドは、コンパイルされた際に指定された Python インタプリタを呼びだします。ecell3-python は、Python が必要な E-Cell の Python 機能拡張や標準 DM ライブラリを見つけ出せるように、Python を実行する前にいくつかの環境変数を設定します。コマンドラインオプションを処理した後で、ecell3-session コマンドは ecell.ecs.Simulator オブジェクトを作成します。そして、シミュレータオブジェクトが利用する ecell.Session オブジェクトをインスタンス化します。

基本的に ecell3-python は単なる Python インタプリタであり、E-Cell SE のフロントエンド要素はこのコマンドを実行しています。ecell3-python コマンドから E-Cell Python ライブラリを用いるには、以下のように記述します：

```
import ecell
```

プロンプト画面は以下のようになります：

```
$ ecell3-python
Python 2.4.2 (#1, Feb 12 2006, 19:13:11)
[GCC 4.1.0 20060210 (Red Hat Linux 4.1.0-0.2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ecell
>>>
```

あるいは、UNIX 系のシステムでは以下の記述で始まるファイルを実行します：

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

実行環境に関する情報の取得

ecell3-python コマンドの現在の設定内容を取得するには、-h オプションとともに ecell3-python コマンドを実行します。コマンドの使い方とともに、現在設定されている変数のいくつかが出力されます。

```
$ ecell3-python -h
[...]
    Configurations:
    PACKAGE = ecell
    VERSION = 3.1.105
    PYTHON = /usr/local/bin/python
    PYTHONPATH = /usr/local/lib/python2.4/site-packages:
    DEBUGGER = gdb
    LD_LIBRARY_PATH = /usr/lib:/url/local/lib:
    prefix = /usr
    pythondir = /usr/local/lib/python2.4/site-packages
    ECELL3_DM_PATH =
[...]
```

「PYTHON =」の行は、ecell3-python が使っている Python インタプリタのパスです。

デバッグ

ecell3-python コマンドをデバッグモードで実行するには、環境変数 ECELL_DEBUG を設定します。ECELL_DEBUG が true（1）に設定されていると、ecell3-python コマンドは GNU gdb デバッガソフトウェア上で実行されます。ECELL_DEBUG は、ecell3-session や ecell3-session-monitor を含

む ecell3-python 上で実行されるあらゆるコマンドで利用されます。以下は、シェルプロンプト上で、ecell3-session をデバッグモードで実行した例です：

```
$ ECELL_DEBUG=1 ecell3-session -f foo.eml
gdb --command=/tmp/ecell3.OmlQyE /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) 1996-2003 Keio University.
Send feedback to Koichi Takahashi <shafi@e-cell.org>
<foo.eml, t=0>>> Ctrl+C
Program received signal SIGINT, Interrupt.
[Switching to Thread 1074178112 (LWP 7327)]
0xffffe002 in ?? ()
(gdb)
```

プログラムとそのコマンドオプションは、gdb の '--command=' コマンドオプションとして自動的に実行されます。そして、プログラムがクラッシュするか、ユーザが Ctrl+C を入力してプログラムの実行を中止すると、gdb のプロンプトが表れます。ECELL_DEBUG は、C++コードのレベルで機能する gdb を実行します。Python レイヤーのスクリプトをデバッグするには、Python デバッガを用います。Python ライブラリのリファレンスマニュアルなどをご覧ください。

プロファイル作成

OS が GNU sprof コマンドを備えていて、C ライブラリが LD_PROFILE 環境変数をサポートしていれば、ecell3-python コマンドをプロファイリングモードで実行することができます。現在サポートしているのは共有オブジェクトプロファイルのみです（GNU C ライブラリのリファレンスマニュアルなどを参照してください）。プロファイリングモードで ecell3-python を実行するには、共有オブジェクトの SONAME を環境変数 ECELL_PROFILE に設定します。共有オブジェクトファイルの SONAME は、objdump コマンドを -p オプションなどとともに実行することで取得できます。

```
$ ECELL_PROFILE=libecs.so.2 ecell3-session [...]
```

実行が終了すると、SONAME.profile という名前のプロファイルデータファイルがカレントディレクトリに作成されます。この例では、libecs.so.2.profile です。sprof コマンドを使って、バイナリのプロファイルデータをテキスト形式に変換することができます。以下はその例です：

```
$ sprof -p libecs.so.2 libecs.so.2.profile
```

E-Cell Python ライブラリ API

本節ではE-CellにおいてPythonから利用できるAPI(Application Program Interface)について説明します。

Session クラス API

Session クラスのメソッドは、以下の５つのグループに分けられます。

- Session メソッド
- Simulation メソッド
- Stepper メソッド
- Entity メソッド
- Logger メソッド

Session メソッド

`loadModel(` *`file`* `)`

戻り値：なし

EML ファイルを読み込みます。 *`file`* はファイル名またはファイルオブジェクトである必要があります。

`loadScprit(` *`filename`* `)`

戻り値：なし

ESS ファイルを読み込みます。通常このコマンドは ESS 内では使用しません。

`message(` *`massage`* `)`

戻り値：なし

引数 *`massage`* を出力します。デフォルトでは *`massage`* を標準出力に表示しま

す。 *massage* の処理方法は setMessageMethod( ) メソッドを用いて変更できます。

saveModel(*file*)

戻り値：なし

モデルの現在の状態を EML ファイルに保存します。 *file* は、ファイル名あるいはファイルオブジェクトです。

setMessageMethod(*method*)

戻り値：なし

message( ) メソッドが呼ばれたときの処理方法を変更します。 *method* は Python メソッドでなければなりません。

関連項目：message

getCurrentTime( )

戻り値：float

シミュレータの現在時刻を返します。

getNextEvent( )

戻り値：Python 2-タプル (float, string)

次にスケジュールされているイベントを、スケジュールされている時刻、StepperID の順で 2 つの要素を含む Python タプルで返します。このイベントは、次に run( ) か step( ) が呼ばれたときに処理されます。通常、getCurrentTime( ) が返す時刻とは異なる値です。このメソッドは、次のイベントがスケジュールされている時刻を返します。一方、getCurrentTime( ) は直近のイベントが起こった時刻を返します。2 つ以上のイベントが同じ時刻にスケジュールされている場合、これらのメソッドは同じ値を返す場合があります。

run([*sec*])

戻り値：なし

sec 秒のシミュレーションを実行します。このメソッドが呼び出されると、指定した sec 秒が経過するまで、すなわち、現時刻 > 開始時刻 + *sec* が満たされるまで、 step( ) を繰り返し呼びだします。そして、シミュレータ内の時刻が指定された時点を超えると直ちにシミュレーションを停止します。モデルによって、Stepper のステップ幅が非常に大きい場合、指定した時間と実行したシミュレーション時間に誤差が生じます。

イベントチェッカやイベントハンドラオブジェクトが設定されている場合、 *sec*

を省略することができます。

関連項目：setEventCheker、 setEventHandler

setEventChecker(*eventchecker*)

戻り値：なし

イベントチェッカやイベントハンドラが正しく設定されていて、run() メソッドが実行時間引数とともに、あるいは引数を省略して呼びだされている状況下で、シミュレータは n ステップ毎に、イベントハンドラが true を返しているかを確認します。ここで、n はsetEventCheckInterval() によって宣言された正の整数です（初期値は n = 20)。このメソッドが呼び出されると、シミュレータはただちにイベントハンドラを呼びだします。イベントハンドラが Session の stop() メソッドを呼びだすと、シミュレータは、次のステップが計算される前に停止します。引数を与えずに run() を呼びだした場合、この方法によってのみシミュレーションのループを止めることができます。

この機構は、主に GUI フロントエンドの要素を実装するために用いられますが、用途がそこに限定されるわけではありません。

setEventcheckInterval(*n*)

戻り値：なし

関係項目：setEventChecker

このメソッドは未実装です。

setEventHandler(*eventhandler*)

戻り値：なし

setEventChecker の項をご覧ください。

step([*numsteps*])

戻り値：なし

1 ステップのシミュレーションを実行します。任意の整数 *numsteps* が与えられた場合、その数だけシミュレータをステップします。引数が省略された場合、1 回だけステップします。

stop()

戻り値:なし

シミュレーションを停止します。このメソッドは多くの場合、イベントハンドラか、イベントハンドラが呼びだすメソッドによって呼ばれます。

関連項目:setEventChecker、setEventHandler

`initialize()`

戻り値:なし

シミュレーション実行の準備をします。`step()` や `run()` を実行するのに先だって自動的に呼びだされるので、通常はこのメソッドを呼びだす必要はありません。

Stepper メソッド

`getStepperList()`

戻り値 : ID文字列のタプル

シミュレータ中の Stepper オブジェクトの ID を含む Python タプルを返します。

`createStepperStub( id )`

戻り値 : 新規 StepperStub オブジェクト

引数 *id* に一致する ID を持ち、この Session オブジェクトと結合した StepperStub オブジェクトを返します。

Entity メソッド

`getEntityList( entitytype, systempath )`

戻り値: FullID 文字列のタプル

引数 *systempath* で指定された System 中に存在するすべての *entitytype* 型の Entity オブジェクトの FullID を含む Python タプルを返します。

引数 *entitytype* は、`ecell.ECS` モジュールで定義されている `'VARIABLE'`、`'PROCESS'`、`'SYSTEM'` のいずれかでなければなりません。引数 *systempath* は有効な SystemPath 文字列でなければなりません。

`createEntityStub( fullid )`

戻り値 : 新規 EntityStub オブジェクト

引数 *fullid* に一致する FullID を持ち、この Session と結合した EntityStub オブジェクトを返します。

Logger メソッド

`getLoggerList()`

戻り値 : FullPN 文字列の Python タプル

シミュレータ中のすべての Logger オブジェクトの FullPN 文字列を含む Python タプルを返します。

`createLoggerStub( fullpn )`

戻り値: 新規 LoggerStub オブジェクト

引数 *`fullpn`* で与えられた FullPN を持ち、この Session と結合した LoggerStub オブジェクトを返します。

引数 *`fullpn`* は、有効な FullPN 文字列でなければなりません。

`saveLoggerData( fullpn, aSaveDirectory, aStartTime, anEndTime, anInterval )`

戻り値 : なし

`fullpn` で指定される Logger が保持している前データを *`aSaveDirectory`* に保存します。*`fullpn`* を指定しない場合、すべての Logger のデータを保存します。*`aSaveDirectory`* はデータファイルを保存するディレクトリで、指定がない場合のデフォルトは `./Data` です。 *`aStartTime`*、*`anEndTime`*、*`anInterval`* は、それぞれ、保存する最初の時刻、最後の時刻、データポイントの間隔です。

LoggerStub のメソッドなども参照してください。

サンプルモデル Drosophila 内にあるスクリプト `sample_osogo_script.py` に使用例があります。

Session クラスの属性

`theSimulator`

型 : `Simulator`

Session API および ObjectStub クラスを用いることで、シミュレーション中の作業の大部分を実行することができるので、ESS ユーザは、通常 Simulator クラスの詳細に踏み込む必要はありません。

Simulator クラスの詳細を知るためには、E-Cell C++ライブラリリファレンスマニュアル（準備中）および、システムのソースコード、中でも `ecell3/ecell/libemc/Simulator.hpp` をご覧ください。

`theMainWindow`

型 : `MainWindow`

theMainWindow 変数は、Session が `ecell3-session-monitor` で実行される場合に存在している可能性があります。この変数は状況によっては存在しないので、利用する際には、先だってその存在を確認する必要があります。

ObjectStub クラス API

ObjectStub には以下のサブクラスがあります。

- EntityStub
- StepperStub
- LoggerStub

いくつかのメソッドは、複数のサブクラスに共通しています。

すべての ObjectStub に共通するメソッド

`create()`

戻り値：なし

オブジェクトを作ります。例えば StepperStub の場合、この StepperStub を作成した際に指定した名前を持つ（バックエンド）Stepper オブジェクトの作成を試みます。

`exists()`

戻り値：`boolean`

ObjectStub が指す（バックエンド）オブジェクトが存在すれば true、そうでなければ false を返します。

`getName()`

戻り値：`string`

ObjectStub が指すオブジェクトの名前を返します。通常、名前はシミュレータ中で用いられている文字列の識別子です。EntityStub は FullID を、StepperStub は StepperID を、LoggerStub は FullPN をそれぞれ返します。

EntityStub と StepperStub に共通するメソッド

`getClassName()`

戻り値：`string`

Entity または Stepper のクラス名を得るために使用されます。

`getProperty( propertyname )`

戻り値：属性の型により、int、float、string、Python タプルのいずれか

propertyname の Entity または Stepper オブジェクトの属性を返します。戻り値は、int、float、string あるいはこれらの型の混在するタプルのいずれかで、タプルはネストされる場合があります。特殊なメソッドである `__getitem__` を使ってこれらの値を取得することもでき：

```
value = stub.getProperty( propertyname )
```

と

```
value = stub[ propertyname ]
```

は同じ意味を持ちます。

`getPropertyAttributes( propertyname )`

戻り値: タプル

propertyname を名前に持つ属性の特性を Python タプルで返します。タプルの要素は true または false で、(settable, getable) の順に返します。例えば、(false, true) であれば、読み出し専用です。読み出し専用の属性に書き込もうとするか、書き込み専用の属性を読み出そうとすると例外が発生します。

`getPropertyList()`

戻り値 : Entity または Stepper の属性の名前の Python タプル

Entity または Stepper オブジェクトのすべての属性の名前を含む Python タプルを返します。

`setProperty( propertyname, value )`

戻り値:なし

propertyname で設定された Entity または Stepper オブジェクトの属性に、*value* を書き込みます。

value として、int、float、string あるいは、それらの混在する Python タプル、Python リストオブジェクトを与えることができます。

特殊なメソッドである __setitem__ を通してこのメソッドを使うこともでき :

```
stub.setProperty( propertyname, value )
```

と、

```
stub[ propertyname ] = value
```

は同じ意味を持ちます。

LoggerStub だけが持つメソッド

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

戻り値 : 数値の array

Logger のデータを取得するために用います。戻り値は rank-2 の配列で、各列は 5-タプルまたは 2-タプルです。5-タプルの場合、(time, value, average, min, max)、2-タプルの場合、(time, value)です。time はデータが記録された時刻、value はその時刻での値、min と max は指定した *interval* 区間内

での最小値と最大値、average は時間加重平均値を返します。 *interval* で指定されるデータ間隔の区間は、データポイント集積である場合と、単一のデータポイントである場合があります（データポイント集積については setMinimumInterval() の項をご覧ください）。データポイント集積でない場合、value、average、min、maxの値はすべて同じになります。

引数をとらずに、書式 getData() で呼びだされた場合、Logger に記録されたすべてのデータを返します。

書式 getData(*starttime*) で呼びだされた場合、 *starttime* 以降のすべてのデータを、書式 getData(*starttime*, *endtime*) で呼びだされた場合、 *starttime* から *endtime* の区間内のすべてのデータを返します。

書式 getData(*starttime*, *endtime*, *interval*) で呼びだされた場合は、 *starttime* と *endtime* の間のデータを *interval* の間隔で返します。例えば、データポイント n におけるデータ d(n) について、¦d(n - 1) - d(n + 1)¦ > *interval* の場合、d(n) は、メソッドが返すデータから取り除かれます。そのため、 *interval* が同じでも、 *starttime* が異なれば記録されるデータが変ることがあり、注意が必要です。したがって、科学的なデータ分析には適しませんが、リアルタイムの GUI フロントエンドでは役に立ちます。

getStartTime()

戻り値: float

Logger が作成された時刻、あるいは最初にデータを記録した時刻のうち、遅い方を返します。多くの場合、これらは同じ時刻になります。

getEndTime()

戻り値: float

データが最後に Logger に記録された時刻を返します。

getSize()

戻り値: float

Logger に記録されたデータポイントの数を返します。

getMinimumInterval()

戻り値: float

Logger オブジェクトの現在の最小記録間隔を返します。

関連項目 : setMinimumInterval

`setMinimumInterval( interval )`

戻り値：なし

interval は、ゼロまたは正の実数でなければなりません。

ゼロに設定されている場合、Logger はすべてのステップでデータを記録します。この場合、データポイント集積は行われません。

正の数が設定されていると、指定した *interval* よりも短いステップ間隔でシミュレーションが進行する場合に、Logger がデータポイント集積を行うことがあります。

`getLoggerPolicy()`

戻り値：Python タプル

この Logger の Logger Policy（データ記録の方式）を、4 つの値を持つ 4-タプルで返します。1 番目の要素は最小ステップ数です。2 番目は最小時間間隔です。3 番目は、保存のために割り当てられたディスク容量を使い切った場合の対処方法を表し、0 は例外を発生し、1 は古いデータを上書きします。4 番目は、この Logger に割り当てる最大記憶容量で、キロバイト単位で設定します。

現在のバージョンでは、最小ステップ数と最小時間間隔の両方を設定するとデータが記録されません。どちらか一方を設定してください。

`setLoggerPolicy( aLoggingPolicy )`

戻り値：なし

Logger Policy を設定します。引数 *aLoggingPolicy* は、getLoggerPolicy の項で述べた 4-タプル です。

ECDDataFile クラス API

ECDDataFile メソッド

`ECDDataFile( data = None )`

戻り値：なし

コンストラクタです。引数 *data* は、Numeric Python の rank-2 の配列あるいはそれと等価のオブジェクトです。data が引き渡されない場合、空の行列（`[[]]`）が用いられます。

`getData()`

戻り値：`array`

Numeric Python の rank-2 の配列を返します。

`getDataName()`

戻り値：string

データの名前を返します。デフォルト値は空の文字列（''）です。

`getFileName()`

戻り値：string

ファイルの読み込みあるいは保存に成功した場合に、ファイル名を返します。それ以外の場合には、空の文字列（''）を返します。

`getLabel()`

戻り値：タプル

データの軸の名前（軸ラベル）を文字列のタプルとして返します。デフォルト値は（'t', 'value', 'avg', 'min', 'max'）です。

`getNote()`

戻り値：string

ECDDataFile のノート（NOTE フィールドの内容）を返します。戻り値は 1 行または複数行の文字列です。

`load( filename )`

戻り値：なし

filename というファイルからデータを読み込みます。

`save( filename )`

戻り値：なし

filename というファイルにデータを書き込みます。

`setData( data )`

戻り値：なし

ECDDataFile オブジェクト内のデータを、引数 data で上書きします。 *data* は Numeric Python の rank-2 の配列あるいはこれと等価のオブジェクトでなければなりません。

`setDataName( name )`

戻り値：なし

ECDDataFile オブジェクトの名前を設定します。 *name* は改行を含まない文字列でなければなりません。

`setLabel( labels )`

戻り値：なし

ECDDataFile オブジェクトのデータ軸の名前（軸ラベル）を設定します。

labels は軸ラベルの文字列を含む Python シーケンスでなければなりません。

`setNote( note )`

戻り値：なし

ECDDataFile オブジェクトのノートを設定します。 *note* は `string` オブジェクトでなければなりません。文字列は 1 行でも複数行でも構いません。

新規オブジェクトクラスの作成

6

本章には、シミュレーションで利用するための独自のクラスを定義する方法が書かれています。

ダイナミックモジュールについて

ダイナミックモジュール（DM）は、アプリケーションによって読み込まれインスタンス化されるオブジェクトクラス（特にC++クラス）を記述したファイルです。E-Cell SEは、ユーザがシステム全体を再コンパイルすることなく、シミュレーションモデルで用いる新規クラスを定義、追加するための方法としてこの仕組みを用意しています。クラスの定義はネイティヴのコード様式によって行われるので、容量や速度の観点からは、新しいコードやオブジェクトクラスを追加するための最も効率的な方法です。

E-Cell SEでは、Process、Variable、System、Stepperの派生クラスをシステムから動的に読み込むことができます。

E-Cell SEとともに配布されている標準ダイナミックモジュールに加え、`ecell3-dmc`コマンドでC++ソースコードファイル（`'.cpp'`ファイル）をコンパイルすることで、ユーザが定義するDMファイルを作成することができます。コンパイルされたファイルは、通常、共有オブジェクトファイル（UNIXでは`'.so'`、Mac OS Xでは`'.dylib'`、Windowsでは`'.dll'`）の形式をとります。

新規クラスの定義

新規オブジェクトは、いくつかの特別なC++マクロを利用してC++ソースコードを書くことで定義できます。

以下にDMファイルの常用テンプレートを示します。C++の経験があればわかりやすい記述でしょう。DMTYPE、CLASSNAME、BASECLASSは適宜特定の値に書き換えます。

サンプルコード6-1. CLASSNAME.cpp ; 常用 DM テンプレート

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

USE_LIBECS;

LIBECS_DM_CLASS( CLASSNAME, BASECLASS )
{
public:

  LIBECS_DM_OBJECT( CLASSNAME, DMTYPE )
  {
    // ( Property definition of this class comes here. )
  }


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

LIBECS_DM_INIT( CLASSNAME, DMTYPE );
```

DMTYPE、CLASSNAME、BASECLASS

最初に、定義しようとするクラスの基本的な特性を決めなければなりません。具体的には、DM の型（Process、Variable、System、Stepper のいずれか）、クラス名、基底クラスです。

DMTYPE

DMTYPE は E-Cell SE が定義する DM 基底クラスで、Process、Stepper、Variable、System のいずれかです。

CLASSNAME

CLASSNAME はオブジェクトクラスの名前です。

有効な C++のクラス名でなければならず、末尾は *DMTYPE* 名とすべきです。例えば、新しい Process クラスを定義し、「Foo」と名付けるなら、クラス名は「FooProcess」とするのが適切です。

BASECLASS

作成するクラスが継承するクラスです。

BASECLASS は *DMTYPE* と異なる場合があります。作成するクラスが DM 基底クラスの直接の派生クラスであるかどうかに依存します。

ファイル名

ソースファイルの名前は、*CLASSNAME* に '.cpp' 拡張子を付したものでなければなりません。例えば、*CLASSNAME* が FooProcess なら、ファイル名は FooProcess.cpp である必要があります。ソースコードは、ヘッダファイルとソースファイルに分割することもできます（FooProcess.hpp と FooProcess.cpp など）。その場合にも、最低限 LIBECS_DM_INIT マクロはソースファイル（FooProcess.cpp）に記述する必要があります。

インクルードするファイル

最低限 libecs ヘッダファイル（libecs/libecs.hpp）と基底クラスのヘッダファイル（libecs/BASECLASS.hpp など）は、ファイルのヘッダ部分でインクルードされている必要があります。

DM マクロ

テンプレートがいくつかの特別なマクロ：USE_LIBECS、LIBECS_DM_CLASS、LIBECS_DM_OBJECT、LIBECS_DM_INIT を利用していることに注意しましょう。USE_LIBECS は、E-Cell SE のコアライブラリである libecs ライブラリの使用を宣言するマクロで、ファイル中の次の行に宣言文を挿入します。

```
LIBECS_DM_CLASS
```

LIBECS_DM_OBJECT(*DMTYPE*, *CLASSNAME*) は、クラス定義の記述の先頭、すなわちクラスの '{' の直後に配置します。このマクロは、クラスが DM クラスであることを宣言し、クラスが動的にインスタンス化できるように設定し、自動的に getClassName() メソッドを定義します。このマクロは、アクセス指定子 public を挿入します。従って、このマクロ以降の記述は public に位置することになります。このマクロの直後に、つねに明示的に「public:」と書くようにするとよいでしょう。

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

LIBECS_DM_INIT(*DMTYPE*, *CLASSNAME*) は、クラス *CLASSNAME* を *DMTYPE* 型の DM クラスとして書き出します。このマクロは、LIBECS_DM_OBJECT によって書き出されるクラス定義（宣言だけではなく）の後に配置される必要があります。

コンストラクタとデストラクタ

DM オブジェクトは、つねに引数なしのコンストラクタによってインスタンス化されます。デストラクタは、仮想関数として基底クラスに定義されています。

型と宣言

基本的な型

コード中でヘッダファイル libecs/libecs.hpp をインクルードし、USE_LIBECS マクロを呼びだすと、以下の４つの基本的な型を利用できるようになります。

`Real`

実数。通常、倍精度浮動小数点数として実装されています。
Linux/IA32/gcc プラットホームなどの 64-bit 環境では、64-bit 浮動小数点数です。

`Integer`

符号つき整数。64-bit 環境では 64-bit long int。

`UnsignedInteger`

符号なし整数。64-bit 環境では 64-bit 符号なし long int。

`String`

文字列。C++標準ライブラリの std::string クラスと等価。

`Polymorph`

Polymorph は一種のユニバーサル（汎用性の高い）なオブジェクト型（実際にはクラス）です。Polymorph オブジェクトは、Real、Integer、String のいずれとして振る舞うこともでき、またいずれからも作成することができます。また、これらの型が混在するリストである PolymorphVector として利用／作成することもできます。詳細は次節をご覧ください。

これらの型を、C++の標準オブジェクト型である double、int、char* の代わりに用いることを推奨します。

ポインタ型と参照型

各オブジェクト型について、以下の typedef による型定義を利用できます。

`TYPEPtr`

ポインタ型（`TYPE*` と等価）

`TYPECptr`

定数ポインタ型（`const TYPE*` と等価）

`TYPERef`

参照型（`TYPE&` と等価）

`TYPECref`

定数参照型（`const TYPE&` と等価）

例えば、`RealCref` は、`const Real&` と書くのと同じ意味をもちます。これらの typedef の使用を推奨します。

新規の型を宣言するには、`DECLARE_TYPE` マクロを用います。例えば、

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

というマクロをシステム中で呼ぶことで、`RealCref` を `const double&` として利用できるようになります。

```
DECLARE_CLASS( Process );
```

このマクロによって、`ProcessCref`、`ProcessPtr` などの表記が利用可能になります。libecs で定義されているクラスの多くは、これらの typedef を持っています。

型の制限とその他の属性

これらの数値型の制限や制度を知るには、C++標準ライブラリの std::numeric_limits<> テンプレートクラスを用います。例えば、Real 型で表現可能な最大値を取得するためには、以下のようにテンプレートクラスを使います：

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

より詳しくは、C++標準ライブラリのリファレンスマニュアルをご覧ください。

Polymorph クラス

Polymorph オブジェクトは、Real、Integer、String 型および PolymorphVector クラスから作ることができ、またいずれにも変換することができます。

Polymorph オブジェクトのコンストラクト

Polymorph オブジェクトを作成する手順は、値とともにコンストラクタを呼びだすだけです：

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

Polymorph オブジェクトは、別の Polymorph オブジェクトから作成（あるいは複製）することもできます：

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

Polymorph の値の取得

Polymorph オブジェクトの値は、as<>() テンプレートを用いて、いずれの型でも取得することができます。

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

> ノート：非常に大きな Real を Integer に変換しようとして桁あふれが起こると、ValueError 例外が投げられます。

Polymorph の型の確認と変更

Polymorph の型を取得するには `getType()` を、型を変更するには `changeType ( Type aType )` を用います。

PolymorphVector

PolymorphVector は、Polymorph オブジェクトのリストです。

その他の C++構文

唯一の制約は、クラスを DM クラスとして書き出す DM_INIT マクロを、単一の共有ライブラリを構成する一連の記述（ヘッダファイルとソースファイルなど）の中で 1 回だけ用いるという点です。この点を除けば、C++コンパイラが解釈できる限り、記述に制限はありません。クラス定義の内部あるいは外部で、他のクラス定義、ネストされたクラス、typedef、静的関数、名前空間、テンプレートを含む、あらゆる C++構文を記述することができます。
ただし、名前空間の衝突には注意してください。DM クラスの外でクラスや関数を宣言する必要がある場合、プライベートな C++名前空間や静的空間を利用したい場合があるかもしれません。

PropertySlot

PropertySlot とは

PropertySlot（属性スロット）は、オブジェクトの属性に対する読み出し（get）、書き込み（set）メソッドのペアで、属性の名前に関連する名前を与えられています。オブジェクト属性の値は、オブジェクトのメンバ変数に格納されていることもあれば、メソッドが呼ばれた際に動的に作成されることもあります。Process、Variable、System、Stepper の 4 つの DM 基底クラスはすべて、PropertySlot のセットやオブジェクト属性を持つことができます。換言すると、これらのクラスは、共通基底クラス PropertyInterface を継承しています。

PropertySlot は何のためにあるのか

シミュレーションモデル中の各オブジェクト（Entity、Stepper オブジェクトなど）にパラメータ値を与えるために、モデルファイル（EM ファイルなど）中で、PropertySlot を用いることができます。また、シミュレーション中に動的にオブジェクト間の連絡を行うための方法としても使えます。

PropertySlot の型

PropertySlot の型は、以下の 4 つのいずれかです。

- Real
- Integer
- String
- Polymorph

PropertySlot の定義

オブジェクトクラス中で PropertySlot を定義する手順は以下の通りです：

set、get メソッドの両方あるいは片方を定義します。

必要に応じて、属性値を保存するためのメンバ変数を定義します。

作成したメソッドを PropertySlot として登録します。

set メソッドと get メソッド

PropertySlot は、オブジェクトの属性に対する読み出し（get）、書き込み（set）メソッドのペアで、属性の名前に関連する名前を与えられています。どちらかのメソッドを省略できる場合があります。PropertySlot に set メソッドがある場合、PropertySlot は書き込み可能（settable）であるといいます。get メソッドがある場合、読み出し可能（gettable）であるといいます。
システムによって認識されるために、set メソッドは以下の形式でなければなりません。

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

また、get メソッドは以下の形式でなければなりません：

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

ここで、T は属性の型です。CLASS は、PropertySlot が属するオブジェクトクラスです。
これらのプロトタイプを記憶しておく必要はありません。以下の４つのマクロを用いて、特定の型と属性名に対する set、get メソッドを宣言、定義することができます。

`SET_METHOD( TYPE, NAME )`

展開

```
void setNAME( const TYPE&value )
```

使い方

SET_METHOD マクロは、クラス定義内で、属性に値を書き込む set メソッドを宣言、定義するために用いられます。対象となる属性の型は *TYPE*、名前は

NAME です。書き込まれた属性の値は、Variable の Value として取得できます。

例

下のコード：

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

  Real theFlux;
};
```

は、以下の C++プログラムに展開されます。

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

  Real theFlux;
};
```

この例では、引き渡された属性値は、メンバ変数 theFlux に保存されます。

GET_METHOD(*TYPE*, *NAME*)

展開

const *TYPE* get*NAME*() const

使い方

GET_METHOD マクロは、クラス定義内で、属性に値を読み出す get メソッドを宣言、定義するために用いられます。対象となる属性の型は *TYPE*、名前は *NAME* です。メソッドは、属性の値を *TYPE* オブジェクトとして返すように定義されます。

例

下のコード：

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

  Real theFlux;
};
```

は、以下の C++プログラムに展開されます。

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

  Real theFlux;
};
```

SET_METHOD_DEF(*TYPE*, *NAME*, *CLASSNAME*)

展開

void *CLASSNAME*::set*NAME*(const *TYPE*&value)

使い方

SET_METHOD_DEF マクロは、クラスのスコープ外で属性の set メソッドを定義するために使います。

例

SET_METHOD_DEF マクロは通常、SET_METHOD マクロと組み合わせて使います。例えば、以下のコードでは、SET_METHOD マクロを用いてクラス定義内で仮想関数として set メソッドを宣言し、クラス定義の後で、SET_METHOD_DEF を用いてメソッドの実体を定義しています。

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

  Real theFlux;
};

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

定義部分は、以下の C++プログラムに展開されます。

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

GET_METHOD_DEF(*TYPE*, *NAME*, *CLASSNAME*)

展開

const *TYPE CLASSNAME*::get*NAME*() const

使い方

GET_METHOD_DEF マクロは、クラスのスコープ外で属性の get メソッドを定義するために使います。

例

上記の SET_METHOD_DEF の例をご覧ください。

属性が書き込み可能かつ読み出し可能で、単純にメンバ変数に保存されている場合、以下のマクロを用いることができます。

```
SIMPLE_SET_GET_METHOD( NAME, TYPE )
```

このマクロは、属性名（NAME）と同じ名前を持つ Variable の存在を仮定して、以下のコードと等価のコードに展開します：

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

  GET_METHOD( NAME, TYPE )
{
  return NAME;
}
```

PropertySlot の登録

PropertySlot をクラスに登録するには、対象クラスの LIBECS_DM_OBJECT マクロに含まれる以下のマクロのうち 1 つを用います：

```
PROPERTYSLOT_SET_GET( NAME, TYPE )
```

属性が書き込み可能かつ読み出し可能であるときに用います。クラスは、set、get メソッドの両方を定義します。
例えば、FooProcess クラスに属する Real 型の Flux 属性を定義するには、クラス定義の public 領域に以下のように書きます：

```
public:
LIBECS_DM_OBJECT( FooProcess, Process )
{
  PROPERTYSLOT_SET_GET( Flux, Real );
}
```

このマクロは、FooProcess の Flux 属性の set、get メソッドとして、以下の 2 つのメソッドをそれぞれ登録します：

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

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

メソッドの特性は、既に定義済みのプロトタイプと一致していなければなりません。LIBECS_DM_OBJECT は属性をいくつでも持つことができます。また、1 つも持たなくても構いません。

PROPERTYSLOT_SET(*NAME*, *TYPE*)

このマクロは、get メソッドを登録しない点を除き、PROPERTYSLOT_SET_GET と同じです。set メソッドだけを利用可能にする際に用います。

PROPERTYSLOT_GET(*NAME*, *TYPE*)

このマクロは、set メソッドを登録しない点を除き、PROPERTYSLOT_SET_GET と同じです。get メソッドだけを利用可能にする際に用います。

PROPERTYSLOT(*NAME*, *TYPE*, *SET_METHOD*, *GET_METHOD*)

set、get メソッドの一方あるいは両方の名前がデフォルトの形式（setNAME()、getNAME()）と異なる場合に、このマクロを用いて、明示的にメソッドへのポインタを指定します。

以下の例では、FooProcess クラスの Flux 属性に対して、setFlux2() および anotherGetMethod() メソッドを登録しています：

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

ひとつのオブジェクトに同名の PropertySlot が複数作られた場合には、最後に作られたものが利用されます。

load / save メソッド

set、get メソッドに加えて、load、save メソッドを定義することができます。load メソッドは、モデルファイルからモデルを読み込む際に呼びだされます。同様に、save メソッドは、シミュレータの saveModel() メソッドによってモデルの状態をファイルに保存する際に呼ばれます。

特に指定しなくても、load、save メソッドは、set、get メソッドを作成する際にデフォルトで定義されます。以下に挙げるいくつかのマクロを用いて、デフォルトの定義を変更することができます。

PROPERTYSLOT_LOAD_SAVE(*NAME*, *TYPE*, *SET_METHOD*, *GET_METHOD*, *LOAD_METHOD*, *SAVE_METHOD*)

このマクロは、属性メソッドを設定する最も一般的な方法です。set、get、load、save メソッドを独立に指定できます。*LOAD_METHOD* が NOMETHOD の場合、属性はモデルファイルからの読み込みができません。また、*SAVE_METHOD* が NOMETHOD の場合、属性の値をファイルに保存することはできません。

PROPERTYSLOT_NO_LOAD_SAVE(*NAME*, *TYPE*, *SET_METHOD*, *GET_METHOD*)

このマクロは、*LOAD_METHOD* と *SAVE_METHOD* を NOMETHOD に設定する点を除き、前節で説明した PROPERTYSLOT と同じです。

すなわち、このマクロは、以下の記述と等価です：

```
PROPERTYSLOT_LOAD_SAVE( NAME , TYPE , SET_METHOD,
GET_METHOD, NOMETHOD, NOMETHOD )
PROPERTYSLOT_SET_GET_NO_LOAD_SAVE( NAME, TYPE, SET_METHOD,
GET_METHOD )
PROPERTYSLOT_SET_NO_LOAD_SAVE( NAME, TYPE, SET_METHOD )
PROPERTYSLOT_GET_NO_LOAD_SAVE( NAME, TYPE, GET_METHOD )
```

これらのマクロは、set、get メソッド load、save メソッドを設定しない点を除いて、PROPERTYSLOT_SET_GET、PROPERTYSLOT_SET、PROPERTYSLOT_GET と同じです。

基底クラスの属性の継承

基底クラスの属性を使いたい場合がしばしばあります。基底クラスの属性を継承するには、INHERIT_PROPERTIES(*PROPERTY_BASECLASS*) マクロを用います。このマクロは通常、属性定義マクロ（PROPERTY_SET_GET() など）の前に配置します。

```
LIBECS_DM_OBJECT( CLASSNAME, DMTYPE )
{
  INHERIT_PROPERTIES( PROPERTY_BASECLASS );

  PROPERTYSLOT_SET_GET( NAME, TYPE );
}
```

通常、*PROPERTY_BASECLASS* と *BASECLASS* は同一です。例外は、*BASECLASS* が、LIBECS_DM_OBJECT() マクロを利用していない場合です。その場合、LIBECS_DM_OBJECT() を使っている基底クラスのうち、クラスの継承関係上もっとも近いものを *PROPERTY_BASECLASS* に選びます。

シミュレーションでの PropertySlot の利用

シミュレーション中にオブジェクトの属性にアクセスするには、以下の３種類の経路があります：

(1) ネイティヴのC++メソッドを用いた静的で直接のアクセス。この方法は PropertySlot を介しません。

(2) PropertySlot オブジェクトを介した動的に結合されたアクセス。
(3) PropertyInterface を介した動的に結合されたアクセス。

新規 Process クラスの定義

新規の Process クラスを定義するには、最小限、以下の 2 つのメソッドを定義する必要があります。

- `initialize()`
- `fire()`

`initialize()` は、シミュレーションの状態をリセットする必要がある時に呼びだされます。リセットはセッション開始時に限らず、状態の再積分が必要な場合など、いかなる時にも起こりえます。

`fire()` は、モデル中の反応の計算を進める際に呼びだされます。

VariableReference に従って、Process が参照する Variable を更新する方法を記述します。

Process の持つ VariableReference は、coefficient の値でソートされてメンバ変数 `theVariableReferenceVector` に格納されています。ソートの順序は、負の coefficient を持つ VariableReference、 coefficient がゼロのもの、正の値を持つものの順です。ゼロまたは正の値を coefficient に持つ VariableReference のオフセットを取得するために、`getZeroVariableReferenceOffset()` および `getPositiveVariableReferenceOffset()` メソッドを利用できます。特定の名前の VariableReference を探すには、`getVariableReference()` メソッドを用います。

コード 6-2. SimpleProcess.cpp

```
#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()
    {
      setFlux( k * S0.getValue() );
    }

  protected:

    Real k;
    VariableReference S0;
};

LIBECS_DM_INIT( SimpleProcess, Process );
```

標準ダイナミックモジュール ライブラリ

7

本章では、E-Cell SE とともに配布されている標準ダイナミックモジュールライブラリを概観します。標準ダイナミックモジュールライブラリに含まれるクラスの一部をリストし、使い方を説明します。システムが正常にインストールされると、ライブラリの提供するクラスを特別な手順を必要とせずに用いることができます。
本章は完全なリファレンスではありません。ライブラリで定義されているクラスについてより詳しく知るにはE-Cell3 標準ダイナミックモジュールライブラリリファレンスマニュアル（準備中）をご覧ください。

Stepper クラス

Stepper には、以下の４つの直接の下位クラスがあります：

- DifferentialStepper
- DiscreteEventStepper
- DiscreteTimeStepper
- PassiveStepper

DifferentialStepper（微分 Stepper）

汎用の微分 Stepper

以下の Stepper クラスは、汎用の常微分方程式ソルバを実装しています。これらのクラスは連続 Process クラスを結合して機能します。

ODEStepper

ODEStepper は、各時点における方程式系の stiffness に応じて、適応的に Dormand-Prince 法と Radau IIA 法を切り替えます。
多くの場合、この Stepper が ODE モデルに対する最適の汎用ソルバです。

ODE45Stepper

ODE 系のシミュレーションのために、Dormand-Prince 5(4)7M アルゴリズムを実装しています。

ODE23Stepper

ODE 系のシミュレーションのために、Fehlberg 2(3) アルゴリズムを実装しています。

モデルの一部が小さなタイムスケールを有する場合に、この Stepper を試してみてください。この Stepper は、中程度に stiff な微分方程式系に用いることができます。

FixedODE1Stepper

適応的なステップ幅調節機構を持たない DifferentialStepper です。この Stepper の微分方程式の解法は 1 次です。

この Stepper は、1 ステップにつき、各 Process の `fire()` メソッドを 1 回だけ呼びだします。この Stepper は、平坦な連続的微分方程式系を高精度に解くのに適しているとはいえませんが、アルゴリズムの単純さが役立つ場合もあります。

S-System および GMA Stepper

S-SystemおよびGMAのモデルで用いるStepperです。3章の「Power-law（べき乗則）の正規形微分方程式」をご覧ください。また、サンプルモデル SSystem、branchGも参照してください（Appendix 2）。

DiscreteEventStepper（離散イベント Stepper）

DiscreteEventStepper

離散イベントシミュレーションを実行する際に用います。この Stepper は、DiscreteEventProcess の下位クラスと組み合わせて用いなければなりません。

この Stepper は、Process オブジェクトをイベント生成器として用います。この Stepper の `initialize()` メソッドは、以下のような手順です：

1. Stepper が結合するすべての DiscreteEventProcess オブジェクトの `updateStepInterval()` メソッドを実行します。
2. 最も小さいスケジュール時刻を持つ Process（トップ Process）を見つけだします。スケジュール時刻は、（現在時刻）+（Process の StepInterval 属性値）で計算します。
3. Stepper のスケジュールを、トップ Process のスケジュール時刻に設定します。

この Stepper の `step()` メソッドは以下のような手順です：

1. 現時点のトップProcess の `fire()` メソッドを呼びます。

2. トップ Process および、トップ Process の影響を受けるすべての Process の `updateStepInterval()` メソッドを呼びだし、次のトップ Process を見いだすためにスケジュール時刻を更新します。

3. 最後に、Stepper のスケジュールを、新しいトップ Process のスケジュール時刻に設定します。

このクラスの `interrupt()` メソッドの手順は `initialize()` メソッドと同じです。

DiscreteTimeStepper（離散時間 Stepper）

DiscreteTimeStepper

この Stepper は固定された間隔でステップします。例えば、Stepper の StepInterval 属性が 0.1 に設定されている場合、この Stepper は 0.1 秒毎にステップします。

この Stepper がステップすると、結合するすべての Process インスタンスの `fire()` メソッドを呼びだします。この振る舞いを変更するには、下位クラスを作成してください。

この Stepper は、他の Stepper による割り込みを無視します。

PassiveStepper（受動 Stepper）

PassiveStepper

この Stepper は自発的にステップしません（ステップ間隔は無限大です）。代わりに、この Stepper は、割り込みによってステップします。つまり、この Stepper に影響を与える別の Stepper がステップすると、その直後にこの Stepper もステップします。

この Stepper がステップすると、結合するすべての Process インスタンスの `fire()` メソッドを呼びだします。この振る舞いを変更するには、下位クラスを作成してください。

Process クラス

連続 Process クラス

微分方程式に基づく Process クラス

以下の Process クラスは、微分方程式をそのまま実装しており、ODEStepper、ODE45Stepper、ODE23Stepper、FixedODE1Stepper などの汎用の DifferentialSteppers と組み合わせて用いることができます。
現在のバージョンでは、多くのクラスが特定の反応速度式を表現するものになっています。
もちろん、これらの使途は、化学的、生化学的シミュレーションに限定されるものではありません。

DecayFluxProcess

DecayFluxProcess は FluxProcess の一種で、質量作用則（law of mass-action）による減衰過程を、半減期 T に基づいて計算します。T 属性および VariableReference を設定して用います。

MassActionProcess

MassActionProcess は標準的な質量作用則（law of mass-action）を実装しています。速度定数 k 属性と VariableReference を設定して用います。

MichaelisUniUniFluxProcess

この Process は、Michaelis の Uni-Uni 反応モデル（1 基質、1 生成物反応モデル）を実装しており、以下の反応速度式によって Velocity を算出します。
[C]、[S]、[P]はそれぞれ、酵素、基質、生成物の濃度です。モデル中の定数、K_{mS}、K_{mP}、K_{cF}、K_{cR} はそれぞれ KmS、KmP、KcF、KcR 属性としてモデルファイルに記述します。

$$V = \frac{[\mathrm{C}]\left(K_{cF}K_{mP}[\mathrm{S}] - K_{cR}K_{mS}[\mathrm{P}]\right)}{K_{mS}[\mathrm{P}] + K_{mP}[\mathrm{S}] + K_{mS}K_{mP}}$$

その他の連続 Process クラス

PythonFluxProcess

PythonFluxProcess は、Expression 属性に 1 つの Python 式をとり、この属性の式の評価結果を `setFlux()` に引き渡します。

SSystemProcess

SSystemProcess は VariableReferenceList、rank-n の行列（n は Variable の数）である SSystemMatrix、Order 属性をとります。

```
Process SSystemProcess( SSystem )
{
    Name "SSystemPProcess";
    Order 3;
    SSystemMatrix
        [ 0.5 0   0   0   0   0   0.6 0   0.2 0   0   0   ]
        [ 0.7 0.1 0   0   0   0   0.2 0   0   0.2 0.2 0   ]
        [ 0.5 0   0.1 0   0.1 0   0.4 0   0   0   0   0.2 ]
        [ 0.2 0   0.1 0   0   0   0.9 0   0   0.2 0   0   ]
        [ 0.1 0   0.7 0   0   0   0.5 0   0   0   0   0   ];

    VariableReferenceList
        [ P0 Variable:/CELL/CYTOPLASM:A 1 ]
        [ P1 Variable:/CELL/CYTOPLASM:B 1 ]
        [ P2 Variable:/CELL/CYTOPLASM:C 1 ]
        [ P3 Variable:/CELL/CYTOPLASM:D 1 ]
        [ P4 Variable:/CELL/CYTOPLASM:E 1 ];
}
```

離散 Process クラス

GillespieProcess

この Process は DiscreteProcessStepper とともに用いなければなりません。

この Process は、1つの属性 k を持ち、Gillespie の確率過程によって結合する Variable の変化をシミュレートします。

その他の Process クラス

PythonProcess

3章に詳細な説明があります。

Variable クラス

Variable

状態変数を表現するための標準クラスです。

E-Cell のシミュレーション機構

8

本章では、E-Cell SEがシミュレーションを実行する際に中核をなしているメタアルゴリズムを概説し、E-Cell SEがメタアルゴリズムを実行する仕組みについて解説します。

メタアルゴリズム

E-Cell SEは、**メタアルゴリズム**（**meta-algorithm**）というフレームワークを用いて、様々なシミュレーションアルゴリズムを一斉に実行します。E-Cell システムのコア部分の多くは、メタアルゴリズムの実装そのものです。

離散事象システム

メタアルゴリズムは、**離散事象**（**離散イベント**）シミュレーションと呼ばれる手法のひとつです。時間駆動型のシミュレーションアルゴリズムはすべて、変数の値を更新する手法によって大きく3種類に分類できます。ひとつは微分方程式で、変数の変化速度を計算することで、連続的に変数の値を更新します。2つめは離散時間方程式で、変数の値を離散的、即時的に更新します。3つめが離散事象方程式で、これは、モデル内で生じる他の事象によって起こる変数の変化を記述する方程式からなります。

1976年、Zeiglerは、**離散事象システム仕様**（**DEVS**, Discrete Event System Specification）と名付けた表現形式で、これら3つのすべてを包含したモデル化と解析ができると提案しました[4]。DEVSでは、モデル内の変数は離散時間に発生するイベント（事象）によって更新されます。そして、更新された系の状態によって次のイベントが発生する時刻が決まります。個々のイベントが離散的であれ連続的であれ、結果として起こるのは変数の更新です。変数を更新すると、次のイベント発生まで時間が進み、速度が与えられている変数は積分されます。DEVSシミュレーションは、イベント発生毎に系の状態（＝モデル中のすべての変数の値）を計算し、系の時間発

[4] Zeigler B. Theory of Modeling and Simulation (1st ed.). Wiley Interscience, New York. 1976

展のようすを算出します。理論的には、DEVSを用いて汎用シミュレータを構築することが可能です。

メタアルゴリズムの概要

メタアルゴリズムは、DEVSのアイディアを基盤に、高橋によって考案されたアルゴリズムで、DEVSの枠組みで、複数のアルゴリズムを用いてひとつのモデルのシミュレーションを実行するための詳細な方法を提案しています。このアルゴリズムが「メタ」と名付けられているのは、アルゴリズムが提示しているのはシミュレーションを実行するためのフレームワークであり、具体的なモデル、具体的なアルゴリズムを適用することによってはじめてシミュレーションが実体化するものだからです。

メタアルゴリズムではまず、複数アルゴリズムモデルのシミュレーションに用いるデータ構造を規定しています。もっとも基本的なデータ構造は、**Model**と呼ばれるオブジェクトで、これは、Variableオブジェクトおよびstepperオブジェクトの集合として定義されます。**Variable**はユニークな名前を持つ実数で、任意の時点のModelの状態は、Variableの状態の集合として完全に記述することができます。

Stepper

Stepperは、Model内で起こるさまざまな相互作用を表現しており、個々のStepperオブジェクトは、**Process**オブジェクトの集合、割り込みメソッド、ローカルStepper時刻、ステップ幅（E-Cell 3.2以降ではNextTime）から構成されます。**Process**は個々のアルゴリズムを内包するオブジェクトです。メタアルゴリズムにおけるイベント（事象）は、あるひとつのStepperの「**ステップ**」に相当します。Stepperが「ステップ」すると、Stepperは自身が持つProcessによってModelを更新し、他のStepperに更新の発生を通知し、自身が次にステップするスケジュールを更新します。これらがメタアルゴリズムにおけるひとつのイベントを構成しています。

Process

Processは、自身と関係づけられたModel内のVariableの値を更新する役割を担う計算ユニットです。Processは、現在の状態に基づき、未来の状態を計算します。Processに関係づけられるVariableには、未来の状態を計算するために読みだされるVariableと、計算結果の書きこみによって変更されるVariableの２種類があり、中には読みだしと書きこみの両方が行われるVariableもあります。メタアルゴリズムではProcessを**連続型**と**離散型**に区別し、ひとつひとつのStepperはどちらかのタイプ

のProcessしか持つことができません。Stepperには、**連続Stepper**、**離散時間Stepper**、**離散イベントStepper**の３種類があります。

必須の情報

メタアルゴリズムでシミュレーション実行するために必須の情報が少なくとも２つあります。ひとつは**グローバル時刻**で、すべてのStepperのローカルStepper時刻の最小値と等しくなります。もうひとつは、Stepper間の二項関係で、**Stepper従属関係**と呼びます。Stepper従属関係は次のように定義されます。異なる２つのStepper S_1とS_2の対があり、Stepper S_1はProcess P_iを、S_2はProcess P_jを内包するものとします。P_iによって変更されるVariableとP_jが未来の値を計算するために参照するVariableに重複があったとき、２つのStepperには従属関係があるといいます。S_1によってModelが変更されると、S_2がその影響を受けるため再計算を行わなければならないということです。

メタアルゴリズムの実行

DEVSのシミュレーションは一連の離散イベントによって時間が進行していきます。メタアルゴリズムで表現されたModelの場合、それぞれのStepperがステップすることによるProcessの**点火**（**firing**）によってイベントが発生し、時間が進みます。

メタアルゴリズムの１回の離散イベントを実行するには、まず、次にステップすべきStepperを選び、続いて、選ばれたStepperをステップしModelを更新します。これがメタアルゴリズムの１ラウンドになり、これを繰り返すことで系の時間発展をシミュレートします。

それぞれのStepperは次にステップすべき時刻を**ローカルStepper時刻**として保持しているので、次にステップすべきStepperを選ぶのは簡単です（最も小さいローカルStepper時刻を持つStepperが、次にステップすべきStepperです）。

Stepperを選んだら、そのStepperで次のイベントが発生する時刻まで、時間を進めます。それぞれのイベントは離散時刻に発生するので、一般に、連続するイベントの時間間隔はゼロではありません。また、イベントが終了した時点で、すべてのVariableの変化速度がゼロになっているとは限りません。そこで、前回のイベントから今回のイベントまでに起こる変数の変化を、前回のイベント終了時点の変数の値と変化速度に基づいて外挿[5]します。

[5] 既知のデータを参考に、既知のデータの範囲外の値を推定すること。ここでは、現在までのデータに基づいて未来の値を近似しようとしています。

次に、Stepperがステップします。グローバル時刻を書き換え、Model内のVariableの値を更新し、自身の次のステップの準備をし、他のStepperにModelの更新を通知します。
ステップする際には、まずStepperの**step関数**が呼ばれ、step関数は、Stepperに属するひとつあるいは複数のProcessを呼びだします。Processは、それぞれに関連づけられたVariableの値、あるいは変化速度を更新します。step関数は、Stepper自身の時刻変数も変更します。まず、現在のローカルStepper時刻にステップ幅が加えられ、更新されます。また、ステップ後のModelの状態に基づき、次のステップ時刻までのステップ幅を設定します（E-Cell 3.2以降では、ステップ幅ではなく、次にステップする時刻NextTimeを設定します）。
step関数による処理が終了すると、このStepperと従属関係のあるすべてのStepperにModelが更新されたことを通知します。通知を受けたStepperは、Modelの更新によって次のイベントの発生時刻を変更する必要がないか、再計算します。

以上がメタアルゴリズムの概要です。メタアルゴリズムの数学的な詳細については、高橋らによる以下の論文をご覧ください。

Takahashi K, Kaizu K, Hu B, Tomita M. A multi-algorithm, multi-timescale method for cell simulation. *Bioinformatics*. 2004;**20**(4):538-46.

E-Cell SE カーネル

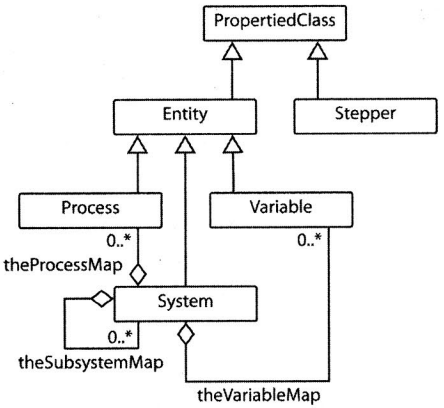
Libecs

E-Cell SEのシミュレータカーネルである**Libecs**は、C++言語で記述されています。Libecsには、メタアルゴリズムが実装されています。それに加え、モデルオブジェクトの生成やデータの記録といった機能と、それら機能へのAPIを提供します。
Libecsは、モデルの状態を表現するデータ構造、モデル内で生じる相互作用を表現するデータ構造をそれぞれ定義し、これら２つを操作してシミュレーションを実行し、時間を進行させます。

４つの基本的なオブジェクトクラス

Libecsのデータ構造の基盤は、４つの基本的なオブジェクトクラスです。
Variable、**Process**、**Stepper**の３つのクラスは、メタアルゴリズムにおける同名のオブジェクトに相当します。４番目のクラスである**System**は、Variableの集合と関連づけられており、変数間の関係などモデルの構造を表現する役割を担っています。

基本的なオブジェクトクラスの関係を以下に図示します。それぞれのオブジェクトの意味、役割については3章もご覧ください。



E-Cellの基本クラス構造の概要

属　性

汎用性を高め、さまざまなモデル構築を容易にするために、各オブジェクトには、予め定義されている**属性**（**Property**）があります。例えば、Variableオブジェクトは，実数はまたは整数を保持するValue属性を持ちます。Valueは、多くの場合、Variableが表現する対象の数を表します。Systemオブジェクトは、Size属性を持ちます。これは、コンパートメントの容積を表します。また、すべてのオブジェクトは、文字列型のName属性を持ちます。こうしたオブジェクトの属性を読み書きするためのAPIも提供されており、オブジェクトの種類に関わらず共通になっています。こうした設計により、どんなモデルを構築しても共通の方法でモデルを取り扱うことができます。

こうした共通の属性インターフェイスを実現するために、カーネルの4つのモデルオブジェクトは基底クラス**PropertiedClass** からの派生クラスとして設計されています。PropertiedClass で、属性の取り扱いが規定されているため、そこから派生したオブジェクトクラスでは共通の方法で属性の読み書きができます。

属性の読み書きには**PropertySlot**を用います。PropertySlot は **PropertyName**（属性の名前、文字列型）と **PropertyValue**（属性の値）を対にしたもので、PropertyValue は、実数型、整数型、文字列型、リスト型のいずれかを取りうる多態型（polymorph）です。PropertiedClassオブジェクトには、すべてのPropertySlot の静的なマップが備わっているため、E-Cellのモデルでは、あらゆる

モデルオブジェクトから、任意のオブジェクトの任意の属性に容易に到達することができます。

2種類のProcessと、4種類のStepper

Libecs が提供する Process には、**連続型**（**Continuous**）と**離散型**（**Discrete**）の2つのタイプがあります。**連続Process**は、連続的に変化する値を記述する微分方程式を表現します。**離散Process**は、時刻毎に離散的に変化する値を記述する方程式を表現します。これに基づき、Libecsは4種類のStepperを提供しています。微分Stepper（DifferentialStepper）、離散時間Stepper（DiscreteTimeStepper）、離散イベントStepper（DiscreteEventStepper）そして PassiveStepper です。

微分Stepperは連続Processを内包し、微分方程式系を解く計算ユニットになります。個々のProcessは、モデル中のひとつの方程式に相当し、微分Stepperは、方程式のセット（連立微分方程式）を解くソルバとして機能します。一部の連続Stepperは、シミュレーションを高速に実行するため、精度を維持できる範囲でできるだけ計算回数を減らそうと（ステップ幅を延ばそうと）します。また、微分方程式を解く際の問題として微分方程式系の硬さ（stiffness）があります。陽的な数値解法を用いた際に、ステップ幅を極端に小さくしない限り精度が保てないような系を硬い（stiffである）といいます。こうした場合、現在に加え過去の状態に関する情報を用いて式を解く陰的解法が有効です（系が硬くない状況では陽的解法の方が高速です）。Libecsが提供する微分Stepperには、系の硬さを判別して適応的に陽的Dormand-Princeアルゴリズム（ステップ幅を適応的に可変する4次Runge-Kutta法）と陰的Radau IIAアルゴリズム（現在最高の陰的Runge-Kutta法）を切り替える**ODEStepper**も含まれています。

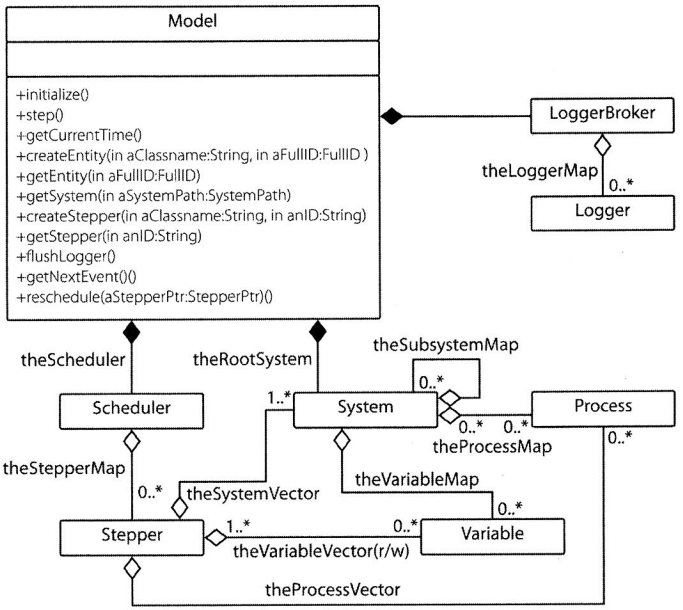
離散モデル構築のために、E-Cellは、**離散時間Stepper**、**離散イベントStepper**、**PassiveStepper**の3種類のStepperを提供しています。離散時間Stepperは、系の状態が離散的に変化し、「ステップ」すべき時刻が系の状態によって決まるような状況を表現するアルゴリズムに用います。Gillespieアルゴリズムはその一例です。離散イベントStepperは、モデルの状態とは無関係に点火する離散Processに用います。PassiveStepperは、自発的にステップすることはありません。従属関係にあるStepperによる割り込みがあると、これに応じて受動的にステップします。

LoggerBroker

もうひとつ、Libecsの重要な要素に**LoggerBroker**があります。LoggerBrokerは、データ記録に関するインターフェイス全般を担っています。LoggerBrokerを用

いることで、シミュレーション中にModel内のあらゆるPropertySlotの値を選択し、記録することができます。LoggerBrokerは、**Logger**オブジェクトを生成、管理します。シミュレーション中の各イベントの終了後にLoggerBrokerが**log()メソッド**を実行すると、すべてのLoggerオブジェクトが設定された属性の値を記録します。

以下にE-Cell SEカーネルのクラス構造の概要を図示します。



E-Cell SEカーネルのクラス構造の概要

シミュレーションの実行

モデルのインスタンス化

次に、E-Cell SE上に数理モデルがインスタンス化され、シミュレーションが実行される仕組みについて述べます。カーネルが初期化されると、**Model**という名称のオブジェクトが生成されます。Modelオブジェクトにはシミュレーション実行に必要な各種要素とE-Cell SEカーネルへのインターフェイスが含まれていますが、中でも重要なのが、root System オブジェクト、Schedulerオブジェクト、LoggerBrokerオブジェクトの3つです。**root System オブジェクト**は、モデルを構成するすべてのVariableと root System以外のすべてのSystemを内包しています。**Scheduler**

オブジェクトは、モデル内のすべてのStepperを内包し、Stepperイベントの発生を司っています。すべてのStepperがSchedulerに属し、すべてのProcessが、それぞれ特定のひとつのStepperに属しています。**LoggerBrokerオブジェクト**は、メタアルゴリズムの１ラウンドが終了する度にオブジェクトの属性を記録します。

Modelクラスがインスタンス化されると、Variable、Process、Stepperといったオブジェクトが、それぞれ、System、Stepper、Scheduler内に生成されます。これらの生成が終了すると、Modelクラスのメンバ関数 **initialize()** が呼ばれ、その他にModelオブジェクトが必要とするデータ構造を用意します。グローバル時刻とStepper従属関係の設定もここで行われます。この処理によって、シミュレーションを開始する準備が整います。

メタアルゴリズムの実行

Modelオブジェクトの**step()メソッド**により、メタアルゴリズムの反復処理が実行されます。１回のメタアルゴリズムの実行は、次回イベントの発生時刻の決定、それに応じたグローバル時刻の更新、新しいグローバル時刻までのモデル状態の積分、次のイベントでステップするStepperのstep()メソッドの呼び出しからなります。Stepperのstep()メソッドは、関連づけられたProcessを点火し、その結果生じた変化を記録し、Stepper従属関係によって従属関係にあるStepperにイベントの発生を通知し、Stepper自身の次回のイベント発生のタイミングを計算します。

次回イベントの発生時刻

次のイベントの発生時刻は、将来のイベント発生時刻を時刻順にリストしたイベントキューの最上位を参照することで取得されます。

次回イベントまでのVariableの積分

つづいて、次回イベントでステップするStepperが参照するすべてのVariableの積分を行います。Steppeが内包するすべてのProcessについて、それぞれが参照するすべてのVariableを読み込み、Stepperの参照変数リストを作成します。そして、参照変数リスト中のすべてのVariableオブジェクトの**integrate()メソッド**を呼びだします。integrate()メソッドは、記録された補間値を用いて指定された未来の時刻の値を外挿します。Variableは、自身の変化速度を記録しており、これを積分に利用します。連続的に変化するVariableは、１つあるいは複数の連続Stepperオブジェクトによる更新を受けます（そうでなければ連続的に更新されません）。連続Stepperは、それぞれ**Interpolant（補間）クラス**を持っていて、連続Stepperが初期化される際、変更対象のVariable毎に、Interpolantインスタンスが作成されます。連続

ProcessによってVariableの変化速度が更新されると、Interpolantインスタンスを介して、その変化が補間値にも反映されます。Variableは、補間値の差分を計算し、自らの値を次回イベント時刻まで近似積分します。この補間係数を用いることにより、Variableはシミュレーション中のあらゆる時刻において自身の値を算出できることが保証されています。

Stepperのステップ

Variableの積分が終了すると、Schedulerは次回イベントでステップするStepperのstep()メソッドを呼びます。step()は、Stepperに関連づけられたProcessを点火します。点火は、すべてのProcessが共通して持っている**fire()メソッド**によって実行されますが、fire()は仮想関数として実装されており、点火の際に起こるモデルの変化は、Stepperの種類によって異なります。例えば、微分Stepperは微分方程式に対応するProcess群を持っていますが、Processによって更新されるVariableの変化速度をInterpolantクラスを介して計算し、Stepper自身が次にステップするまでのステップ幅も算出して対応する自身の属性を更新します。一方、離散時間Stepperは、関連づけられたProcessの点火によって、Variableの値などの変数を単純に書き換えるだけです。

データ記録

ステップが終了すると、Stepperは**log()メソッド**を実行してデータを記録します。log()メソッドは、Stepperに関連するVariableの値のPropertySlotのロガー（記録器）に指定した時刻の値を保存します。ロガーは、PropertySlotProxyを介してPropertySlotの値を取得し、PhysicalLoggerオブジェクトに書き込むことでデータを記録します。

イベントキューの更新

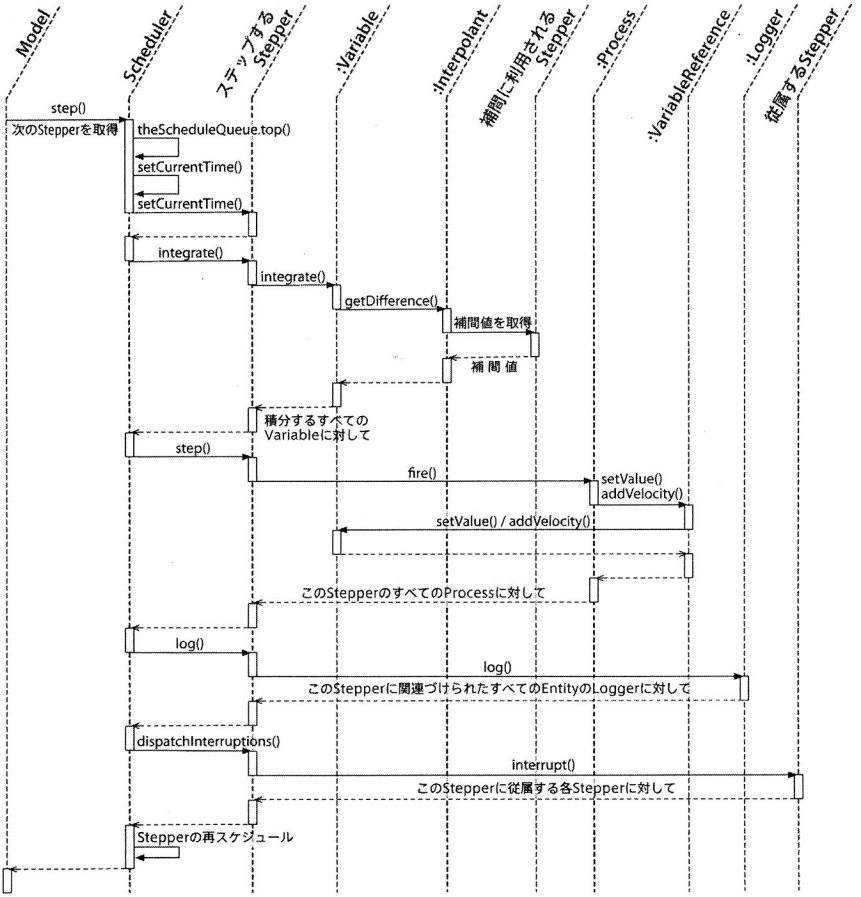
最後に、step()メソッドで計算した次回ステップの発生時刻によってSchedulerのイベントキューを更新します。

他のStepperへの割り込み

ここまでの一連の処理によって、モデル内に発生した状態変化が反映され、時間も更新されました。最後に、今回のイベントでステップしたStepperと従属関係にあるStepperに割り込んで、それぞれのステップ幅（次回イベントまでの時間）を再計算し、Schedulerのイベントキューを更新します。その結果、次回のイベント発生時刻が変更される場合があります。PassiveStepperは、内包するProcessを点火します

（PassiveStepperは、他のStepperからの割り込みがあったときだけ、Processを点火します）。

ここまでに述べたE-Cell SEカーネルの時間進行プロセスを以下に図示します。



E-Cell SEカーネルの時間進行プロセス

アーキテクチャの優位性

このアーキテクチャに従って実装すれば、多様なアルゴリズムをE-Cellプラグインモジュールとして Libecsカーネルの実行中に動的に読み込むことができます。例えば、Processクラスの場合は、initialize()メソッドとfire()メソッドを定義するだけで、新規アルゴリズムモジュールとしてE-Cell SEで利用することができます（6章）。

カーネルへのインターフェイス

PythonインターフェイスAPI

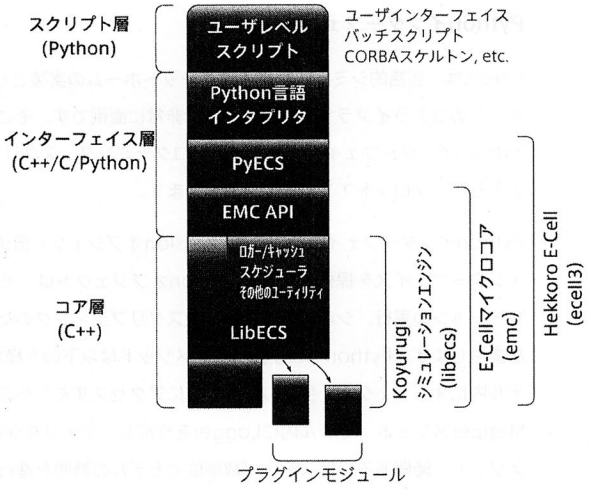
Libecsは、包括的シミュレーションプラットホームの実装としては完結していますが、このコアライブラリを直接扱うのは非常に面倒です。そこで、カーネルに対するPythonインターフェイスを用意し、プログラミング、スクリプティングを支援するとともに、フロントエンドも提供しています。

Pythonインターフェイス層APIは、Sessionオブジェクト周辺で、カーネルへの薄いインターフェイスを提供します。Sessionオブジェクトは、モデルの設定、シミュレーションの実行、シミュレーションのスクリプティングのためのインターフェイスを提供します。Python APIが提供するメソッドは以下の５種類に大別されます。モデル内にオブジェクトを生成し、これらにアクセスするためのEntityメソッド、Stepperメソッド；モデル内にLoggerを作成し、データを保存するためのLoggerメソッド；時間あるいはステップ数単位でモデルの時間を進行させるためのSimulatorメソッド；EMLファイルの入出力など、E-Cell SEの高レベルでの制御を行うためのSessionメソッドの５つです。Python APIの詳細については５章をご覧ください。

libemcマイクロコア

Pythonインターフェイス層はカーネルを直接ラップしておらず、libemcと呼ばれるC++で記述されたマイクロコア層がカーネルをラップしています。libemcは、Pythonインターフェイスが持つ多くの機能を持っており、Pythonインターフェイスによってラップされ、PyEcellと呼ばれるPythonコードと結びつけられてE-Cellへのフロントエンドを実現しています。

E-Cell SEのこのアーキテクチャを以下に図示します。



E-Cell SEのアーキテクチャ

フロントエンド

Python API上に、3つのフロントエンド（ecell3-session-monitor、ecell3-session、ecell3-session-manager）が提供されています。ecell3-session-monitor（セッションモニタ、Appendix 4参照）は、インタラクティヴなモデルの変更とシミュレーションの実行に適したグラフィカルユーザインターフェイスです。モデル内のすべての要素にアクセスし、可視化することができるので、研究者が、最初にモデルの振る舞いを確認、解析する際に有用です。ecell3-sessionはコマンドラインインターフェイスで、大規模なモデルのスクリプティングや自動実行に適しています。ecell3-sessionは、Pythonシェルの拡張となっており、Python Session APIを直接利用できます。ecell3-session-managerは、グリッドまたはクラスターで複数のSessionを並列に実行できるように設計されています。ecell3-session-managerは、SessionManager, SessionProxy、SystemProxyの3つのクラスを提供します。これらを用いて、1台のPCからグリッド／クラスターといった様々なコンピュータ環境で、大量のシミュレーションを容易に実行することができます。

本章は以下の文献の一部を翻訳し、改変、加筆したものです。より詳しくは、文献を参照してください。

Addy N, Takahashi K. Foundations of E-Cell Simulation Environment Architecture, *E-Cell System: Basic Concepts and Applications*, Arjunan SNV, Dhar PK, Tomita M (Eds); Austin: Landes Bioscience, 2010
http://www.landesbioscience.com/curie/chapter/3559/

E-Cell SE について

9

E-Cell SEは、髙橋恒一（Koichi Takahashi <shafi@e-cell.org>）によって開発されました。E-Cell SEに関するより多くの情報を得るには、E-Cellプロジェクトのweb（http://www.e-cell.org/）をご覧ください。

アプリケーションあるいはマニュアルに関するバグ報告、ご意見は、web に記載する方法でお寄せください。

EmPy モジュール†

Appendix 1

EmPy はテキストファイル中に Python による記述を埋め込み、展開するシステムです。
EmPy には多彩な機能がありますが、以下に、E-Cell でのモデル作成の際に利用機会の多いものを抜粋します。

基　礎

EmPy は、Python コードを、Python で処理しないテキストの中に埋め込むために用います。ソースファイルを処理し、出力ファイルに書き出します。通常のテキストはそのまま書き出され、EmPy 書式でマークアップされた部分が処理、展開されて書き出されます。EmPy によるソース中のマークアップの取り扱いと処理を「展開」といいます。
コードは、Python インタプリタに読み込まれた場合と同じように処理され、式に対しては eval() メソッド、文に対しては exec 文で処理したのと同じ結果を返します。
埋め込みの開始を表す記号は「@」です。これは、有効な Python コードと平常のテキストのどちらにも出現しない文字です。「@」をそのまま埋め込みたい場合は、「@@」と表記します。
EmPy における改行コードは LF（ラインフィード）です。OS によってデフォルトの改行文字は異なるので注意してください。改行コードは、UNIX では LF、Windows では CR（キャリッジリターン）＋ LF です。Mac OS は、CR を改行コードとしていましたが、Mac OS X で UNIX ベースになったため、システムファイルなどは LF を改行コードとしており、一部のアプリケーションの改行コードも LF です。改行コードを指定できるエディタで、明示的に改行コードを指定することをお勧めします。

† 本節は Erik Max Francis による EmPy ドキュメントをベースに、翻訳・改変しています。

展　開

@# コメント 改行

コメント行。@# から行末（改行文字）までをコメントとして除外します。コメント中に @ ではじまる部分があっても、EmPy はこれを展開しません。

EmPy は UTF-8 をサポートしているので、UTF-8 形式であれば日本語などを書くこともできます。

例：

```
@# この行はコメントです。
@# この @x は展開されません。
```

@ （空白文字）

@ の次に空白文字（スペース、タブ、改行）がある場合、何にも展開されません。

`MassActionFlux@ Process` は、 `MassActionFluxProcess` と展開されます。

行末に @ がある場合、直後の改行文字が EmPy によって取り除かれるため、次の行に継続することになります。ExpressionFluxProcess の Expression 属性など、１つの長い式をモデルファイルに書く際、行末の @ によって複数行に分割して読みやすくすることができます。

@\ エスケープコード

EmPy のエスケープコードは、C に似通っています。ただし、すべて接頭文字ではじまります。

有効なエスケープコードは以下の通りです。

`@\0`	NUL, nullFoo
`@\a`	BEL, bellFoo
`@\b`	BS, backspaceFoo
`@\dDDD`	3桁の10進コード DDDFoo
`@\e`	ESC, escapeFoo
`@\f`	FF, form feedFoo
`@\h`	DEL, deleteFoo
`@\n`	LF, linefeed character, newlineFoo
`@\oOOO`	3桁の8進コード OOOFoo

`@\qQQQQ`	4桁の4進コード QQQQFoo
`@\r`	CR, carriage returnFoo
`@\s`	SP, spaceFoo
`@\t`	HT, horizontal tabFoo
`@\v`	VT, vertical tabFoo
`@\xHH`	2桁の16進コード HHFoo
`@\z`	EOT, end of transmissionFoo
`@^X`	制御文字 ^X

C形式のエスケープコードと異なり、EmPy のエスケープコードには直後にいくつかの数字をとるものがあります。曖昧さを回避するため数字の数は決められています。解釈できないエスケープコードに対してはエラーを発生します。そうしないと微細な誤りを見過ごす可能性があるためです。また、8進数を表記するのに `@\o` を使用する点も C と異なります。

`@@`

アットマーク「@」に展開されます。

`@), @], @}`

それぞれ、丸カッコ、角カッコ、波カッコの右カッコに展開されます。

`@"...", @"""..."""`, etc.

これらの文字列リテラルは文字列そのものに展開されます。例えば、「`@"test"`」は、「`test`」に展開されます。

`@(` *EXPRESSION* `)`

式 *EXPRESSION* を評価し、その結果を文字列として展開します。評価結果が文字列でなければ、`str()` メソッドで変換します。カッコの直後のスペースは無視されます。`@(` *EXPRESSION* `)` は `@(`*EXPRESSION*`)` と等価です。

例：

```
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*`] [$` *CATCH*`] )`

式 *TEST* を評価し、これが true であれば、式 *THEN* を評価してその結果を展開します。*TEST* が false なら、式 *ELSE* の評価結果を展開します。*ELSE* はオプショ

ンです。指定がない場合、None として展開します。

CATCH が指定されていると、例外が発生した場合に *CATCH* の評価結果を展開します。

@SIMPLE_EXPRESSION

式が簡単な場合、@(...) のカッコを省略することができます。

簡単な式 *SIMPLE_EXPRESSION* の例を以下に示します：

変数やオブジェクトの名前：`@value`, `@os.environ`

単純なメソッドの呼びだし（関数名と左カッコの間にスペースを入れないこと）：`@min(2, 3)`, `@time.ctime()`

配列（配列名と左カッコの間にスペースを入れないこと）：`@array[8][index]`、`@os.environ[9][name]`

上記の組み合わせ：`@function(args).attr[10][sub].other[11][i](foo)`

文字列の直後の . は展開されません（EmPy で処理されません）。式の中にスペースを入れることができますが、メソッド名などの識別子とカッコの間にスペースを入れることはできません（Python 構文としては許されていても）。埋め込まれた EmPy とテキストの境界が曖昧にならないようにしてください。曖昧になるようなら @(...) 書式を使ってください。

例：

```
Variable Variable( S1 )
{
    Value @S1Value;
}
```

@' EXPRESSION '

式 *EXPRESSION* を評価し、結果を `repr()` メソッドで評価した値で展開します。デバッグでの用途が主になります。`@'...'` は、`@(repr(...))` と等価です。

@: EXPRESSION : DUMMY :

式 *EXPRESSION* を評価し、評価結果を *DUMMY* に置き換えた文字列を返します。*DUMMY* は無視されます。自己評価して同形式の EmPy の記述を出力に挿入する書式です。テキストを複数回 EmPy で処理するような場合に利用機会があります。

例：

```
@:2 + 2:this will get replaced with 4:
```

この例の展開結果は以下の通りです：

```
@:2 + 2:4:
```

@{ *STATEMENTS* }

Python の文（複数も可）を実行します。文は値を返さないので、何にも展開されません。複数の文を複数行にわたって、あるいはセミコロン（;）で区切って記述することができます。通常の Python 同様にインデントも有効です。print 文は、標準出力ではなく、展開結果に出力されます。文が 1 つの場合、文の前後の空白文字は無視され、インデントとして解釈されません。

例：

```
@{S1Value = 500}

...

    Variable Variable( S1 )
    {
        Value @S1Value;
    }
```

@% *KEY* [空白文字 *VALUE*] 改行

シグニフィケータを宣言します。シグニフィケータの宣言は 1 行を占め、改行文字で終わります。*KEY* 文字列には空白文字を含んではいけません。オプションの *VALUE* は先頭に空白文字を付すようにします。シグニフィケータは、大きなプロジェクトで繰り返し同じ表現を用い、それを統一したい場合などに用いることができます。EmPy の記述内で、文字列 '__KEY__' を見つけると、シグニフィケータは、'__KEY__' を *VALUE* に置換します。*VALUE* は Python の式です。

制　御

EmPyでは、制御タグによって条件付きや繰り返しの展開を行えます。制御タグは、Python の `if`、`for`、`while` といった制御文に似ています。制御タグは @[...] の書式で表記します。

制御タグは対応するPython制御文と同じ動作をするよう設計されています。モデルファイルに多量のEmPy表記が挿入されている際などには、制御タグを利用した方が便利です。

制御タグには、制御構造の先頭に配置される一次制御タグ（if、for、whileなど）と、それらに引きつづいて記述される二次制御タグ（elif、else、continue、breakなど）があります。

Pythonと違って、EmPyでは、制御構造を決めるのにインデントを使うことができないため、終了タグによって構造を明示します。

```
@[PRIMARY ...]...@[end PRIMARY]
```

*PRIMARY*は一次制御タグの名前（キーワード）です。終了タグには、文字列endと、制御タグ名をスペースで区切って書きます。以下の例は、サンプルコード4-3のアルゴリズム切り替えを制御タグで記述し直したものです。

```
@{ALGORITHM='ODE'}
@[if ALGORITHM=='ODE']@{
STEPPER='ODEStepper'
PROCESS='MassActionFluxProcess'
}@[elif (ALGORITHM == 'NR')]@{
STEPPER='DiscreteEventStepper'
PROCESS='GillespieProcess'
}@[else]@{
raise 'unknown algorithm: %s' % ALGORITHM
}@[end if]

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

...
```

すべての一次制御タグはこの形式で終了する必要があります。すべての一次タグがそれに対応する終了タグとペアになっていない場合、EmPyによる展開はエラーで終了します。開始タグ@[*PRIMARY* ...]と終了タグ@[end *PRIMARY*]の間に別のEmPy表記（制御構造を含む）を挿入できます。つまり、制御構造はネストすることができます。

```
@[for elem in elements]@[if elem]@elem@\n@[end if]@[end for]
```

一次制御タグには、大きく次の３種類があります：(1) 条件分岐（if、try など）(2) ループ制御（for、while など）　(3) 定義（def など）。

条件分岐制御タグは、与えられた条件によって展開内容を決めます。ループ制御タグは、繰り返しその内容を展開します。定義タグは、新しいグローバルなオブジェクトを定義します。

ループ制御構造は、二次制御タグとして @[continue] と @[break] を備えています。Python と同様、@[continue] は次の繰り返し処理への継続を、@[break] はループからの脱出を意味します。これらの二次制御タグはネストされた制御構造の中でも利用できます。

これらのタグは、Python の制御構造をできるだけ忠実に再現するように設計されています。利用できる制御タグは以下の通りです：

```
@[if CONDITION1]...@[elif CONDITION2]...@[else]...@[end if]
@[try]...@[except ...]...@[except ...]...@[end try]
@[try]...@[finally]...@[end try]
@[for VARIABLE in SEQUENCE]...@[else]...@[end for]
@[while CONDITION]...@[else]...@[end while]
@[def SIGNATURE]...@[end def]
```

すべての書式が対応する Python 文と同様に動作します。if タグは複数の elif を持つことができ、else はオプションです。

except 節をともなう try タグは、複数の例外を扱うことができます。

try タグが finally 節を伴う場合、伴うことのできる finally 節は１つだけで、同時に except 節を伴うことはできません。

for、while タグは、continue、break 節を伴うことができます。

定義タグ（def など）はグローバル属性の新しいオブジェクトを作成します。定義したオブジェクトを呼びだすと、オブジェクト内の EmPy 表記を展開して返します。

例えば、以下の記述では f という関数がグローバルに定義されます。

```
@[def f(x, y, z=2, *args, **keywords)]...@[end def]
```

Empy の定義タグの書式：

```
@[def SIGNATURE]CODE@[end def]
```

は、以下の Python コードと等価です：

```
def SIGNATURE:
    r"""CODE""" # so it is a doc string
    empy.expand(r"""CODE""", locals())
```

これは、同じ名前と引数を持つ Python の関数を定義しています。docstring には、展開される EmPy 表記を格納します。この関数を呼び出すと、*CODE* を展開して返します。

注意すべき点

原則として、EmPy を三重引用符（「“““」「'''」など）の中で使うべきではありません。EmPy のバグによって展開結果に予期せぬ文字列が混入し、期待する結果を得ることができません。厳密には、三重引用符内の文字列が改行文字を含まなければ正常に展開されますが、三重引用符を用いる場合にはその中の文字列が改行を含む場合が多いので、使用を避けた方が賢明です。

PythonProcess の FireMethod 属性などの複数行にわたる文字列中で EmPy を使ったプリプロセシングを行いたい場合、三重引用符を含めた文字列全体を EmPy で埋め込むことで問題を回避することができます。以下にその例を示します。

以下の記述は正常に展開されません。

```
@{ k = 2.3 }
Process PythonProcess( PY1 )
{
    FireMethod """
S2S3 = @k * S2.Value * S3.Value
S1.Value = ( 1.0 - S2S3 ) / ( 1.0 + S2S3 )
""";
    VariableReferenceList [(S1)] [(S2)] [(S3)];
}
```

以下のように書けば正常に展開されます。

```
@{
k = 2.3
PY1FireMethod = "S2S3 = " + str( k )
PY1FireMethod += """ * S2.Value * S3.Value
S1.Value = ( 1.0 - S2S3 ) / ( 1.0 + S2S3 )"""
PY1FireMethod = "\"\"\"" + PY1FireMethod + "\"\"\""
}
Process PythonProcess( PY1 )
{
    FireMethod @ PY1FireMethod;
    VariableReferenceList [(S1)] [(S2)] [(S3)];
}
```

サンプルモデル

Appendix 2

E-Cell SE に同梱されているサンプルモデルについて簡単に記述します。
バージョン3.1.106 では、サンプルモデルは、/usr/share/doc/ecell3-3.1.106/samples/ に格納されています。
以下の説明の項目名は、サンプルの納まっているディレクトリ名です。例えば、以下の「初心者向けのモデル」の「simple」は、/usr/share/doc/ecell3-3.1.106/samples/**simple**/ に納められています。各サンプルのより詳しい記述は、ディレクトリ内の README ファイルに記載されています。

> **上級者向け**：厳密には、モデルファイルは、{*datadir*}/doc/ecell/samples/ にインストールされます。ここで、{*datadir*} とは、configure スクリプトの --datadir オプションで引き渡したパス、あるいは、{ *prefix* }/share です。{ *prefix* } は、アプリケーションがインストールされたディレクトリです。アプリケーションを /usr/ にインストールした場合、--datadir オプションを省略すると、上記のディレクトリにサンプルモデルがインストールされることになります。

初心者向けのモデル

simple

E-Cell3 初学者向けのとても簡単なモデル。３つの分子種と１つの反応（Michaelis Menten 反応）からなるモデルです。

決定論モデル

Drosophila、Drosophila-cpp

ショウジョウバエの概日リズムのモデルで、period（PER）タンパク質の振動をモデル化しています。Drosophila と Drosophila-cpp に納められているモデルの数理モデルは同じもので、以下の学術論文で発表されたものです。

Goldbeter A., A model for circadian oscillations in the Drosophila period protein (PER). *Proc Biol Sci*. 1995 Sep 22;261(1362):319-24.

Drosophila には、PythonProcess を用いて作成したモデルファイルと、ExpressionFluxProcess を用いて作成したモデルファイルが納められています。Drosophila-cpp には、独自の共有オブジェクトを用いて作成したモデルファイルが、共有オブジェクトのソースファイルとともに納められています。

Heinrich

ヒト赤血球の解糖系のモデル。ATP の合成と消費を考慮したモデルです。以下の学術論文に基づいています。

Rapoport TA, Heinrich R., Mathematical analysis of multienzyme systems. I. Modelling of the glycolysis of human erythrocytes. *Biosystems*. 1975 Jul;7(1):120-9.

Heinrich R, Rapoport TA., Mathematical analysis of multienzyme systems. II. Steady state and transient control. *Biosystems*. 1975 Jul;7(1):130-6.

CoupledOscillator

シンプルな結合振動子系のモデル。複数タイムスケールシミュレーションのサンプルです。

branchG

一般質量作用則（GMA, Generalized Mass Action law）のサンプルモデルです。

LTD

小脳プルキンエ細胞の長期抑制（LTD）のモデル。

SSystem

SSystemProcess を用いた S-System のモデル。

Pendulum

単振り子のモデル。index-1 の微分代数系のサンプルモデルです。２つの微分方程式と３つの代数方程式からなります。

確率論モデル

tauleap

tau-leap 法によるモデルのサンプル。TauLeapProcess を用いています。
tau-leap 法に関する参考文献を以下に挙げます。

Gillespie DT, Petzold LR, Improved Leap-Size Selection for Accelerated Stochastic Simulation. *J. Chem. Phys.* 2003;119 (24), 12784-12794.

Gillespie DT, Approximate accelerated stochastic simulation of chemically reacting systems. *J. Chem. Phys.* 2001;115,1716-1733.

確率論-決定論 連成モデル

heatshock

大腸菌の熱ショック応答の簡単なモデル。確率論モデル（Gillespie）と決定論モデル（ODE）を連成したサンプルです。

Toy_Hybrid

動的-静的ハイブリッド法を用いたちいさなモデルです。
FluxDistributionStepper、QuasiDynamicFluxProcess を利用しています。

セッションマネージャの利用例

sessionmanager

`ecell3-session-manager` のごく簡単な利用例。

ga

`ecell3-session-manager` を用いて遺伝的アルゴリズムを実装したサンプル。

システムのファイル構成

Appendix 3

E-Cell SE のインストールによって作成されるファイルは、LINUX OS の場合、おおよそ以下の通りです（デフォルトのインストールパスを用いた場合、斜字はバージョン番号なので、環境や E-Cell のバージョンによって異なる場合があります）。

/usr/lib/python2.4/site-packages/ecell

E-Cell が用いる Python パッケージが格納されています。

/usr/lib/ecell-3.1

/usr/lib/ecell-3.1/**dms** 以下に、共有ライブラリのバイナリファイルを格納しています。

バージョン3.1.106 に含まれる共有ライブラリは以下の通りです。

Stepper

```
DAEStepper
ESSYNSStepper
FixedDAE1Stepper
FixedODE1Stepper
FluxDistributionStepper
ODE23Stepper
ODE45Stepper
ODEStepper
TauLeapStepper
```

Process

```
ConstantFluxProcess
DecayFluxProcess
ExpressionAlgebraicProcess
ExpressionAssignmentProcess
ExpressionFluxProcess
GMAProcess
GillespieProcess
MassActionFluxProcess
MichaelisUniUniFluxProcess
PythonFluxProcess
PythonProcess
QuasiDynamicFluxProcess
SSystemProcess
TauLeapProcess
```

/usr/include/ecell-3.1

E-CELL C++ライブラリ（libecs と libemc）のヘッダファイルが格納されています。

/usr/bin

実行可能なアプリケーションが格納されています。

バージョン3.1.106 に含まれる主なアプリケーションは以下の通りです。

ecell3-dmc

ダイナミックモジュールを生成するコンパイラ。

ecell3-em2eml

EM ファイルを EML ファイルに変換するパーサ。

ecell3-eml2em

EML ファイルを EM ファイルに変換するパーサ。

モデル内のオブジェクトのすべての属性を書き出すので、手軽に属性のリストとその値（デフォルト値）を知るのにも使えます。

ecell3-eml2sbml

EML ファイルを SBML ファイルに変換するパーサ。

ecell3-python

普段、直接利用することはありません。

ecell3-sbml2eml

SBML ファイルを EML ファイルに変換するパーサ。

ecell3-session

E-Cell3 をスクリプトモードで操作するためのアプリケーション。

ecell3-session-manager

複数のセッションを自動的に実行するためのアプリケーション。

ecell3-session-monitor

E-Cell3 を GUI モードで操作するためのアプリケーション。

/usr/share/doc

マニュアル、サンプルモデルなどのドキュメントが格納されています。納められているドキュメントとその場所はバージョン毎に異なります。ここでは、バージョン 3.1.106 とともに配布されているドキュメントについて述べます。

/usr/share/doc/ecell3-3.1.106/users-manual

ユーザマニュアル（英語、HTML 形式）が納められています。

/usr/share/doc/ecell3-3.1.106/samples

サンプルモデル、サンプルコードが納められています。

バージョン3.1.106 に含まれるサンプルは以下の通りです。詳しくは Appendix-2 をご覧ください。

```
CoupledOscillator
Drosophila
Drosophila-cpp
Heinrich
LTD
Pendulum
SSystem
Toy_Hybrid
branchG
ga
heatshock
sessionmanager
simple
tauleap
```

/usr/share/doc/ecell3-devel-3.1.106/api

doxygen で生成された、E-CELL C++ライブラリ（libecs と libemc）のドキュメント（英語、HTML 形式）が納められています。

/usr/share/ecell-3.1

/usr/share/ecell-3.1/dms 以下に、共有ライブラリのソースファイルを格納しています。

バージョン3.1.106 に含まれる共有ライブラリのソースファイルは以下の通りです。

```
ConstantFluxProcess.cpp
DAEStepper.cpp
DAEStepper.hpp
DecayFluxProcess.cpp
ESSYNSProcess.hpp
ESSYNSStepper.cpp
ESSYNSStepper.hpp
ExpressionAlgebraicProcess.cpp
ExpressionAssignmentProcess.cpp
ExpressionCompiler.hpp
ExpressionFluxProcess.cpp
ExpressionProcessBase.hpp
FixedDAE1Stepper.cpp
FixedDAE1Stepper.hpp
FixedODE1Stepper.cpp
FixedODE1Stepper.hpp
FluxDistributionStepper.cpp
FluxDistributionStepper.hpp
GMAProcess.cpp
GillespieProcess.cpp
```

```
GillespieProcess.hpp
MassActionFluxProcess.cpp
MichaelisUniUniFluxProcess.cpp
ODE23Stepper.cpp
ODE23Stepper.hpp
ODE45Stepper.cpp
ODE45Stepper.hpp
ODEStepper.cpp
ODEStepper.hpp
PythonFluxProcess.cpp
PythonFluxProcess.hpp
PythonProcess.cpp
PythonProcess.hpp
PythonProcessBase.hpp
QuasiDynamicFluxProcess.cpp
QuasiDynamicFluxProcess.hpp
SSystemProcess.cpp
TauLeapProcess.cpp
TauLeapProcess.hpp
TauLeapStepper.cpp
TauLeapStepper.hpp
```

セッションモニタ マニュアル

Appendix 4

本章では、ecell3-session-monitor の操作方法のうち、頻繁に利用する機能に絞って解説します。
本章は、以下の項目で構成されます。

- セッションモニタとは
- 起動と終了
  - セッションモニタの起動
  - セッションモニタの終了
- モデルファイルの読み込み
  - 起動時に読み込む
  - GUI から読み込む
- シミュレーションの実行
  - メインウィンドウの情報
  - シミュレーションの開始
  - トレーサー：Entity の変化をグラフ化する
  - Stepper ウィンドウ
  - シミュレーション中のパラメータの変更
- データの保存
  - モデル状態の保存
  - 時系列の保存
  - データ記録方式（Logger Policy）の設定

セッションモニタとは

セッションモニタは、E-Cell SE 上で実行されるひとつの Session を操作・観察するための GUI です。
セッションモニタの１回の起動から終了までに取り扱える Session はひとつだけで

す。
そして、ひとつの Session が取り扱うのは、ひとつのモデルのひとつの時間発展です。Session は、E-Cell SE で実行されるシミュレーションの基本単位です。モデルファイルを読み込めるのは１回だけですし、シミュレーションも時刻ゼロから初めてひとつの時間軸を未 来へと計算するだけです。途中でシミュレーションを一時停止してパラメータなどの実行条件を変更し、その後シミュレーションを再開するといった操作は可能 ですが、時刻を過去に戻してある時点からシミュレーションをやりなおすことはできません。そういった試行錯誤は、複数の Session を用いて行います。
したがって、ひとつの Session を操作するアプリケーションであるセッションモニタでも、１回の起動から終了までに読み込めるモデルファイルはひとつだけですし、シミュレーションの時刻も戻すことはできません。別の条件を試したくなったら、もう一度セッションモニタを起動してください。
シミュレーションの途中の状態をモデルファイルに保存しておけば、ある時点から別の条件でシミュレーションを実行することも可能です。
数多くの Session を駆使した研究を進めるには、セッションモニタによる GUI モードよりも、`ecell3-session` を用いたスクリプトモード、セッションマネージャによる複数 Session の自動処理を利用する方が効率的です。

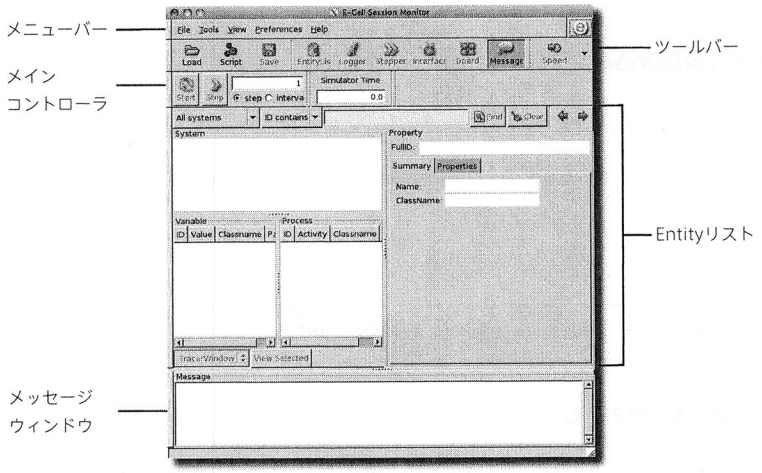
起動と終了

セッションモニタの起動

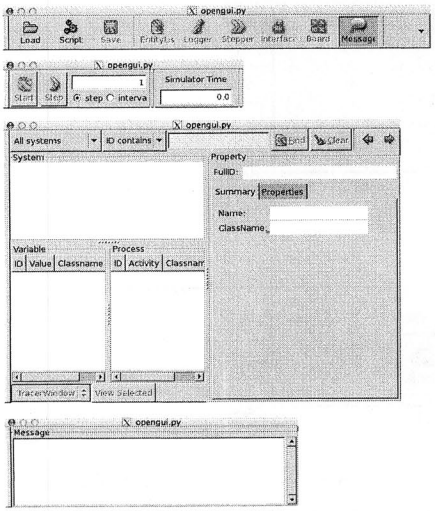
セッションモニタを起動するには、シェルプロンプトで以下のコマンドを入力します。

```
$ ecell3-session-monitor &
```

コマンドを実行すると、次のようなメインウィンドウが表示されます。
右下隅をドラッグすることでウィンドウのサイズを変えることができます。



メインウィンドウの左端をドラッグすることで、下の図のようにメインウィンドウを分割することもできます。分割したウィンドウの左端をドラッグして重ね合わせれば、結合することも可能です。
環境に応じて、操作しやすいウィンドウの配置にしてください。



セッションモニタの終了

セッションモニタを終了するには、メニューから File → Exit を選びます。

モデルファイルの読み込み

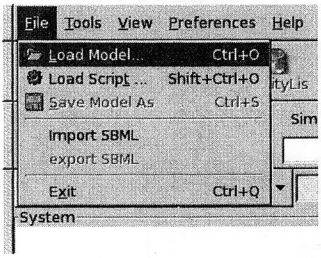
起動時に読み込む

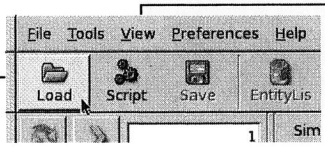
シェルプロンプトから ecell3-session-monitor を起動する際に、-f オプションで EML ファイルを指定することで、 ecell3-session-monitor の起動とモデルの読み込みをまとめて実行できます。

```
$ ecell3-session-monitor -f MODEL.eml
```

GUI から読み込む

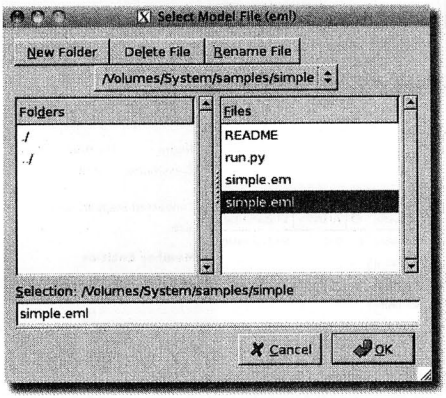
ecell3-session-monitor を起動した後で、EML ファイルを指定して読み込むこともできます。方法は2通りあります。メニューから File → Load Model を選択する方法と、Load ボタンを押す方法で、操作の結果は同じで、Session メソッドの loadModel() メソッドを実行するのと等価です。





Load Model メニューを選択するか、Load ボタンを押すと、ファイルを選択するダイアログが開きます。モデルファイルを選択して、OK ボタンを押します。次の図では、E-Cell に同梱されているサンプルモデル simple.eml を読み込んでいます。

（サンプルモデルには EML ファイルは含まれていません。この操作に先だって、`ece113-em2eml` で EM ファイルを EML ファイルに変換しておく必要があります。）

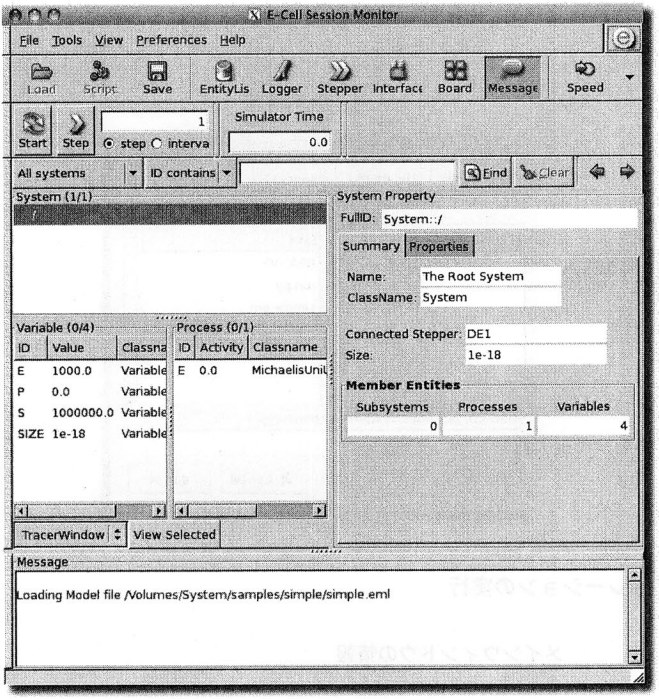


シミュレーションの実行

メインウィンドウの情報

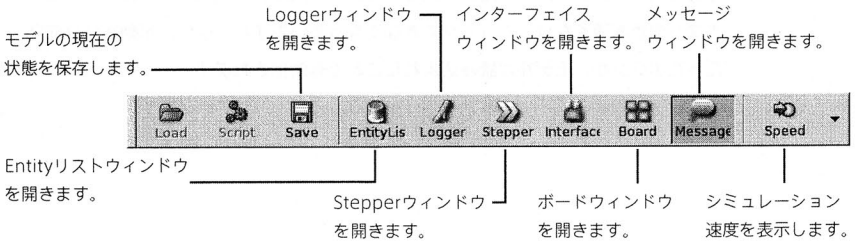
無事モデルファイルが読み込まれると、次の画面のようにモデルの構造がメインウィンドウに表示されます。

ひとつの Session で読み込めるモデルファイルはひとつだけなので、モデルを読み込むと、Load ボタンや Load Model メニューは無効化されます（画面では Load ボタンの色が薄くなり、クリックできなくなっています）。逆に、起動時には無効だったボタンが、モデルが読み込まれたことで有効化されます。



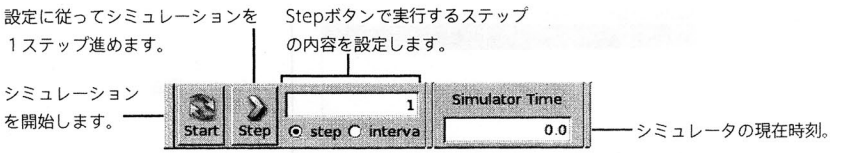
以下、各領域（ペイン）に表示されている情報について説明します。

ツールバー



よく使う機能やウィンドウを呼びだすショートカット・ボタンが並んでいます。モデルを読み込んだ状態のため、Load ボタンと Script ボタンが使用不可になっています。逆に、起動時は無効だった Save から Board までの 6 つのボタンが有効になっています。

メインコントローラ



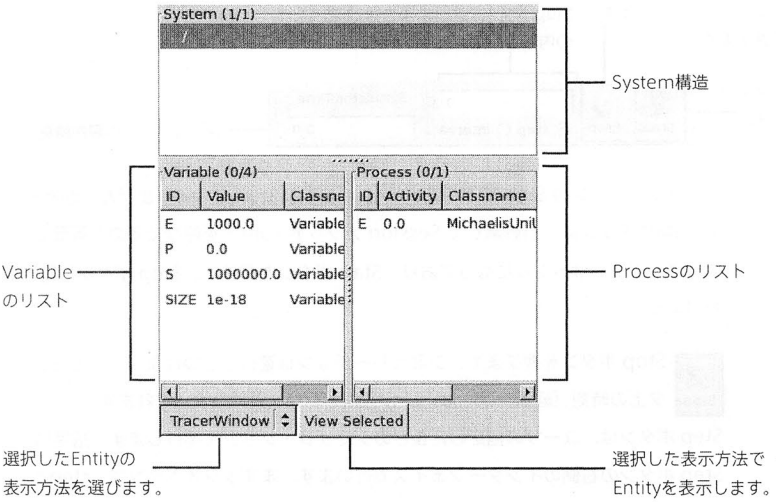
シミュレーションの **Start ボタン**と、**Step ボタン**（Stop ではありません）があります。Start ボタンは、引数無しで Session メソッド `run()` を呼びだすのと等価です。このボタンはトグルになっており、Start ボタンを押すと、**Stop ボタン**に置き換わります。

Stop ボタンを押すまで、シミュレーションは進行しつづけます。シミュレータ上の時刻（経過時間）は、Simulator Time の欄に表示されます。

Step ボタンは、ユーザの指定した長さのシミュレーションを実行します。指定は、Step ボタンの右側のインターフェイスで行います。まずラジオボタンで、step か interval かを選択します。step を選んだ場合、シミュレーションを指定したステップ数だけ実行します。上段のボックスに正の整数 *i* を入力して Step ボタンを押すと *i* ステップだけ、シミュレーションを実行します。Session メソッド `step( i )` を実行するのと同じです。

interval を選んだ場合、指定した時間分のシミュレーションを実行します。上段のボックスに正の実数 *x* を入力して Step ボタンを押すと x 秒だけ、シミュレーションを実行します。Session メソッド `run( x )` を実行するのと同じです。実行されるシミュレーション時間は、厳密には *x* 秒を超過します。詳しくは、5章中の「E-Cell Python ライブラリ API」の Session メソッド `run()` の解説をご覧ください。

Entity リスト



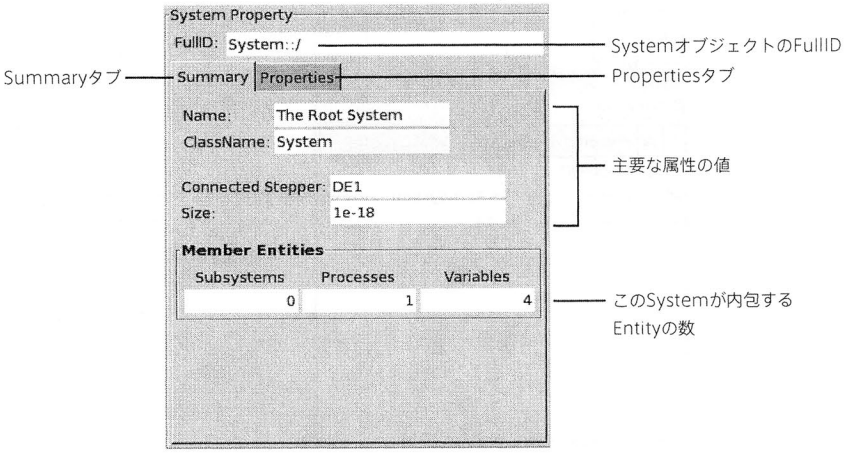
Entity リストペインの左側には、Entity（System、Variable、Process）の構成がまとめて表示されます。

上段にはモデルが持つ System オブジェクトの階層関係が表示されます。

`simple.eml` の持つ System はルートシステムだけなので、System の欄にはルートシステム（/）だけが示されています。ルートシステムの行がグレーに反転しているのは、この System が選択されていることを示しています。

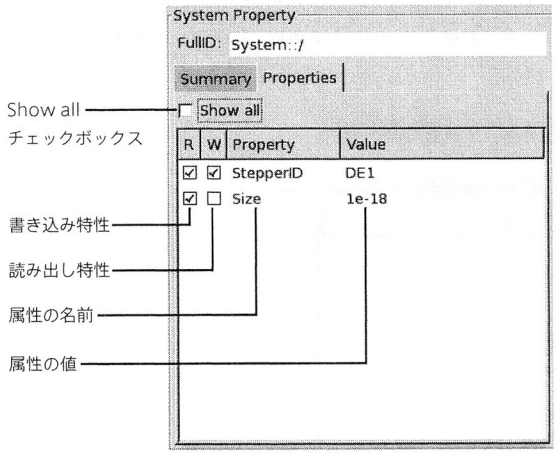
下段には、上段で選択された System に含まれる Variable と Process がリストアップされます。Variable については、ID、Value、クラス名の 3 項目を、Process については ID、Activity、クラス名の 3 項目を、それぞれ表示します。

左側のリストで現在選択されている Entity の詳細な属性情報が、Entity リストの右側に表示されます。



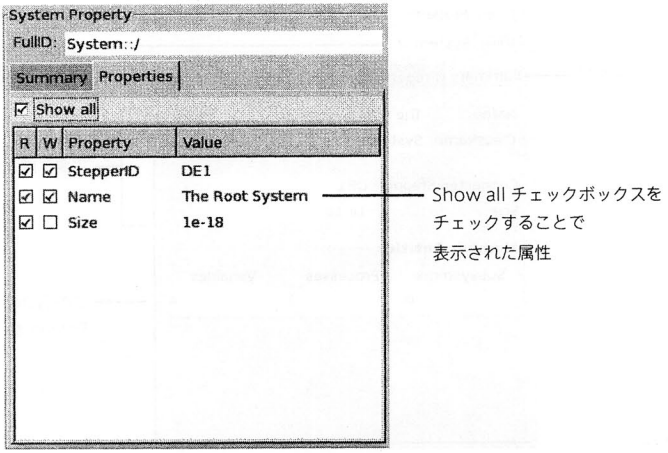
デフォルトで表示されるのは Summary ペインです。Entity の名前やクラス名、Stepper ID などの主要な情報がまとめられています。ほかの Entity についても、表示される属性が若干異なりますが同様の Summary ペインがデフォルトです。

Properties タブを選ぶと、下のような Properties（属性）ペインに切り替わります。



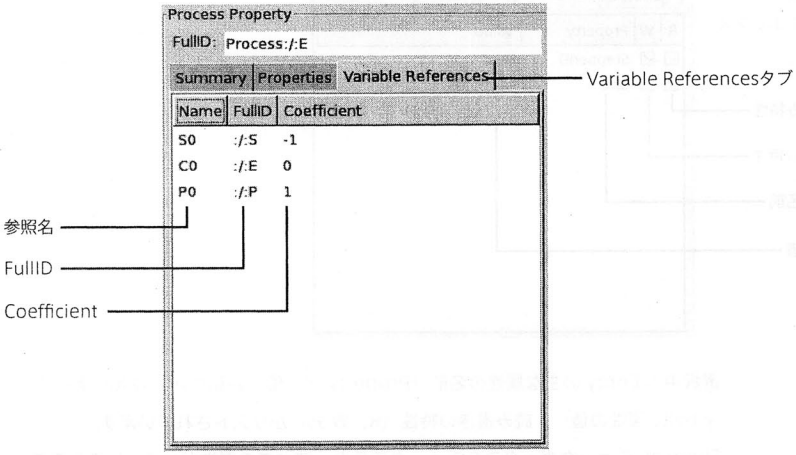
選択中の Entity の主な属性の名前（Property 列）値（value 列 : Value または Activity 属性の値）、読み書きの特性（R、W 列）がリストされています。

Show all チェックボックスをチェックすると、すべての属性がリストに表れます。

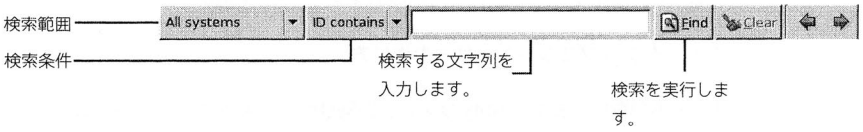


この例（System オブジェクト）の場合、Show all をチェックすることで、Name 属性が新たにリストに加わりました。

System と Variable は、Summary と Properties の 2 種類のペインで情報を表示します。Process は、これに加えて、**Variable References ペイン**を持ちます。これは、VariableReferenceList 属性の要素となっている VariableReference をリストしたもので、参照名（Name 列）、FullID、Coefficient が表示されます。以下の例は、simple.eml に含まれる Process:/:E を選択した際の Variable References ペインです。



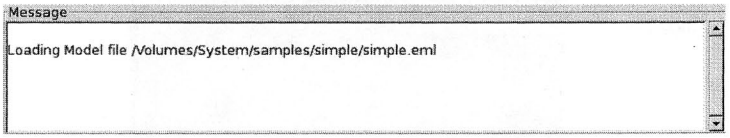
Entity リストには検索インターフェイスが備わっています。



与えられた検索条件に全部または部分一致する ID を持つ Entity を抽出し、Entity リストに表示します。大規模なモデルで威力を発揮します。

Entity リストは、必要に応じて複数表示して、それぞれ別の Entity の情報を表示することができます。ツールバーの EntityList ボタンを押すと、新しい Entity リストウィンドウが開きます。

メッセージ



メッセージウィンドウには、システムからのメッセージが表示されます。スクリプトモードでターミナルに出力される内容と同一です。モデルの読み込みや、シミュレーションの実行に失敗した際に、原因の同定の手がかりとなるメッセージが得られる場合もあります。

シミュレーションの開始

モデルを読み込んだ状態で Start ボタンを押す、あるいは条件を設定して Step ボタンを押せば、シミュレーションが実行され、メインコントローラ上の時刻（Simulator Time）が更新され、それに応じて、Entity リストに表示される各 Entity の値もリアルタイムに変化していきます。
しかし、これだけでは表示される数字が目まぐるしく変わっていくだけで、モデルの状態変化を把握するのは難しいですし、シミュレーション結果もまったく記録されません（シミュレーションを停止した時点でのモデルの状態を保存することはできます［後述］）。

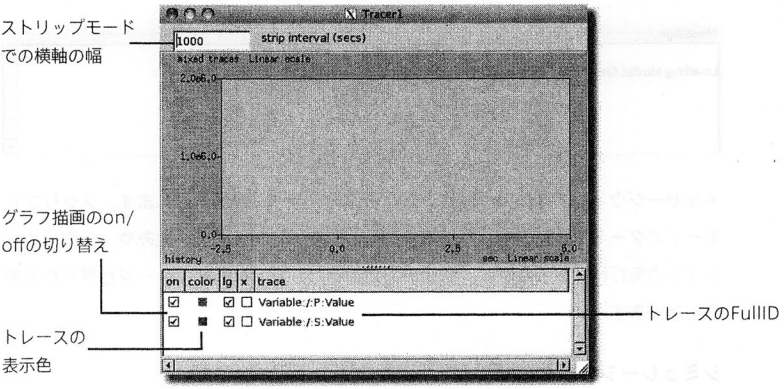
トレーサー：Entity の変化をグラフ化する

シミュレーション中の Entity の変化を把握するための簡単な方法として、**トレーサー**があります。ユーザは、任意の Entity の属性をトレーサーに登録し、その時間

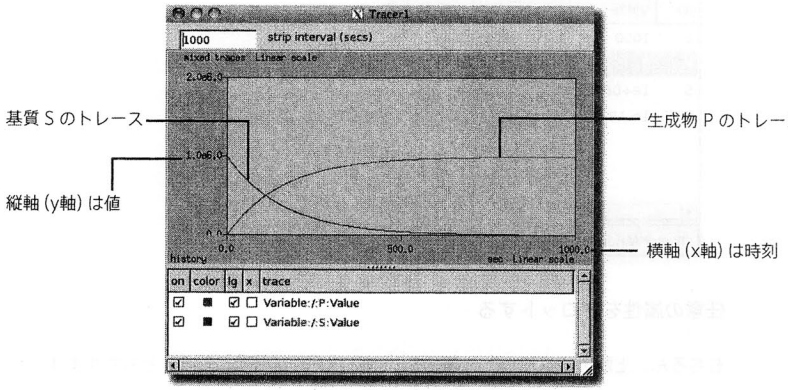
変化をグラフ化することができます。
トレーサーを作成する方法はいくつかありますが、どれも簡単です。

もっともシンプルなトレーサーの作り方

もっとも簡単な方法は、Entity リスト上で System、Variable、Process のいずれかの Entity を選択し、左下のプルダウンメニューが TracerWindow となっている状態（起動時の設定です）で、View Selected ボタンを押すだけです。複数の Entity を選択して一挙にトレーサーに登録することもできます。
例えば、モデル `simple.eml` を読み込み、`Variable:/:P` と `Variable:/:S` を選択して View Selected ボタンをクリックすると、以下のようなウィンドウが新しく開きます。選択した状態でダブルクリックしても、同じようにトレーサーウィンドウを開くことができます。



ここで、メインコントローラで、interval を 1000（秒）に設定して Step ボタンを押すと、1000 秒分のシミュレーションが実施されます。この間、トレーサーにはリアルタイムで 2 つの値の変化の軌跡が描画されます。1000 秒のシミュレーションが終了した時点のトレーサーを以下に示します。



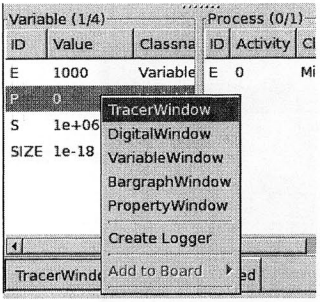
simple.eml は、Michaelis-Menten 反応で基質 S が生成物 P に変換されていく単純な生化学反応をモデル化したものです。基質 S が徐々にゼロに近づき、それに応じて P が増加、500 秒付近ではほとんどの S が P に変換され、1000 秒の時点では、トレーサーで見る限り、S がほぼ消費され尽くしたことが読み取れます。

この一連の操作でトレーサーに登録したのは、2 つの Variable でした。Variable はオブジェクトですから、単一の値を持つものではありません。値を持っているのは、Variable の属性（Property）です。このトレーサーに描かれている値の正体は、指定された Variable の Value 属性です。

Entity を指定してトレーサーに登録した場合、実際には、自動的に指定した Entity の規定値に設定されている属性の値がプロットされます。Variable の場合 Value 属性、Process の場合 Activity 属性、そして System の場合、その System が持つ SIZE Variable の Value 属性（System の容積）が規定値として登録されます。

上の例では、トレーサーウィンドウの下段にも表示されているように、Variable:/:P:Value と Variable:/:S:Value が登録されています。

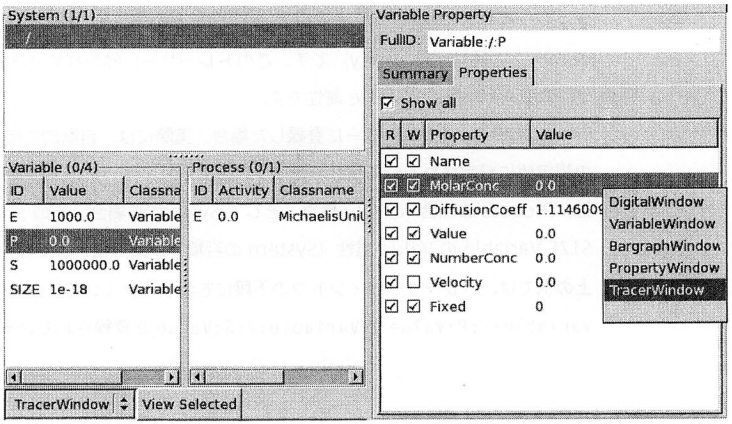
同じ操作は、Entity を選択した状態で、右クリックなどでコンテクストメニューを開いて TracerWindow を選択することでも可能です。この場合、同時に複数の Entity に対して操作を行うことはできません。



任意の属性をプロットする

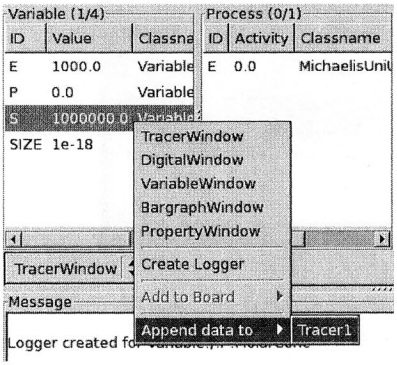
もちろん、上記の規定値以外の属性をトレーサーにプロットすることもできます。Entity リストの右側で Properties ペインを表示し、トレーサーに登録したい属性を選択し、コンテクストメニューから TracerWindow を選びます。この操作では、新しいトレーサーウィンドウが作成されます。

下の例では、モデル `simple.eml` 中の `Variable:/:P` の `MolarConc` 属性（モル濃度）をトレーサーに表示する操作を行っています。



トレーサーに属性を追加する

すでに開いているトレーサーウィンドウに属性を追加することもできます。
Entity リストで Entity を選択し、コンテクストメニューを開いて Append data to → Tracer*X* を選択します。 Tracer*X* は、選んだ Entity の値をプロットするトレーサーの名前です。複数のトレーサーを開いている場合はリストされるので、その中から選択します。
下の例では、`Variable:/:S` を Tracer1 に追加する操作をしています。

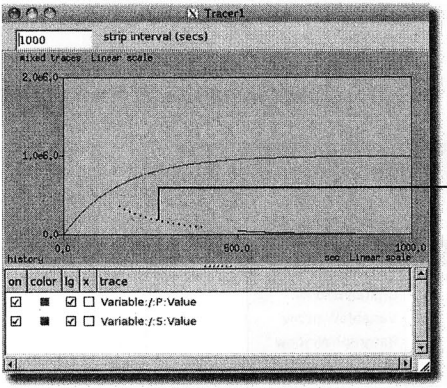


現バージョンでは、GUI から追加可能なのは、Entity の規定値に限られます。
Properties ペインで属性を選択してコンテクストメニューを開いても項目 Append data to は現れません。

トレーサーに表示されるデータ

E-Cell がメモリ上に保存しているモデルの状態は、原則として現時刻の状態だけです。現時刻に至るまでの途中経過（時系列）を記録しておくには、データ記録器 Logger を、記録しておきたい属性ごとに作成しなければなりません。Logger についての詳細は、次の節か 5 章を参照してください。
ところで、トレーサーには過去のデータが表示されています。例えば、前述のように起動時にトレーサーを作って S と P を登録し、1000 秒のシミュレーションを実行すれば、時刻 0 から現在時刻までの 2 つの Variable の量的変化がトレースとして描かれます。これは、セッションモニタが、トレーサーに属性が登録されるのと同時に、その属性の Logger を作成することで実現されています。
シミュレーション開始後、ある時間が経過した時点でトレーサーを作成した場合、それ以前のデータは記録されていないため、作成以前の軌跡はトレーサーに表示できません。

下の例は、simple.eml に対して、起動時に P のトレーサーを作り、500 秒のシミュレーションを実行した後、S のトレーサーを作成、さらに 500 秒のシミュレーションを実行した状態です。S については、0〜500 秒のトレースが表示されません。メモリ上に記録がないためです。

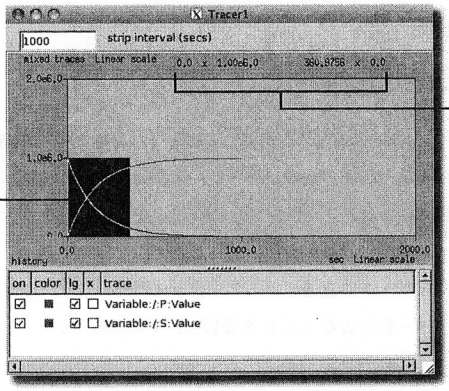


基質 S のトレーサーを作成する以前（0〜500秒）については、S のトレースは表示されません。

グラフの拡大・縮小

トレーサーに表示されているグラフを拡大・縮小することができます。関心のある部分を詳細に観察する際などに便利です。

拡大するには、拡大したい領域をドラッグして選択します。

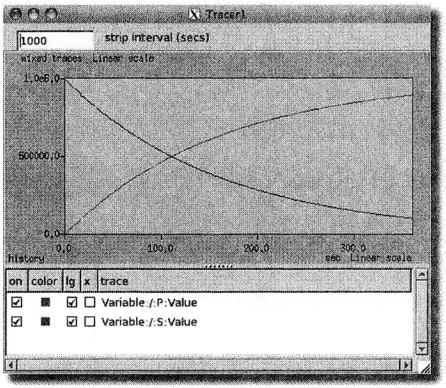


ドラッグした領域

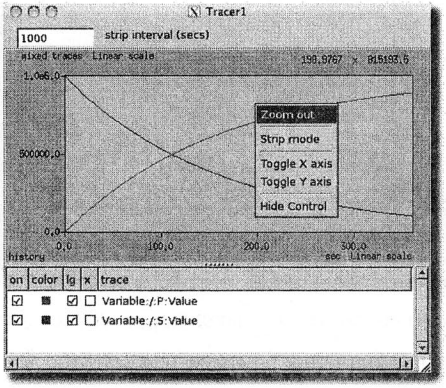
ドラッグした領域の正確な座標

上の図では、1000 秒実行したシミュレーションの前半を拡大しようとドラッグしています。選択している領域の正確な座標が、グラフの右上に表示されます。上の図では、（時刻 0.0, 値 1.0e+6.0）と（時刻 360.9756, 値 0.0）を対角とする長方形の

領域が選択されています。マウスでドラッグするので、正確にある領域を拡大するのは難しいです。領域を選択すると自動的にその領域が拡大されます。



ズームアウトして全体表示に戻るには、グラフ表示領域でコンテキストメニューを呼びだし、Zoom out を選択します。

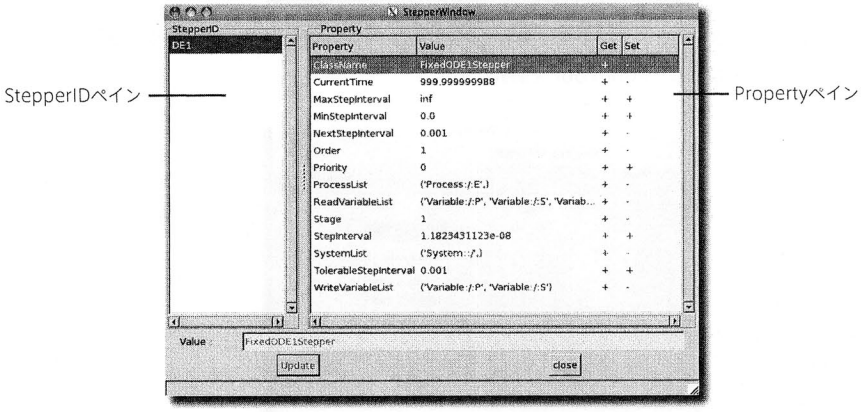


拡大は何度でも繰り返して行うことができます。つまり、拡大表示したグラフの一部を選択することで、さらに拡大することができます。この際、Zoom out を実行すると、直前の拡大操作の取り消しとして機能します。拡大操作を４回繰り返した画面から、元の全体表示に戻るには、Zoom out を４回行う必要があります。
シミュレーション実行中にもグラフを拡大することができます。この場合、拡大操作を行った後に計算されたシミュレーション結果はグラフに反映されません。全体表示まで Zoom out することで、現在のシミュレーションの状況がリアルタイムに反映されます。

Stepper ウィンドウ

Entity（System、Variable、Process）の状態は Entity リストで把握することができます。一方、シミュレーションの進行を司る Stepper の状態を知るには、Stepper ウィンドウを開きます。Stepper ウィンドウは、メニューの Tools → Stepper Window またはツールバーの Stepper ボタンから開くことができます。下の図は、以下の操作の後に Stepper ウィンドウを開いたときの状態です。

- セッションモニタを起動する。
- `simple.eml` を読み込む。
- `Variable:/:S`、`Variable:/:P` をトレーサーに登録する。
- 1000 秒のシミュレーションを実行する（interval = 1000 として、Step ボタンを押す）。



左側の StepperID ペインには、モデル中の Stepper がリストされます。`simple.eml` の持つ Stepper オブジェクトはひとつなので、その ID である `DE1` だけが表示され、選択された状態になっています。

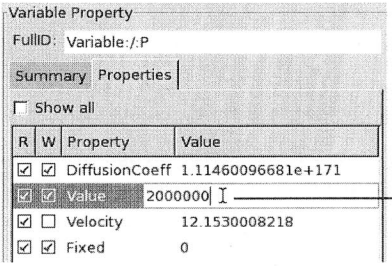
右側の Property ペインには、StepperID ペインで選択された Stepper の属性について、名前（Property 列）、値（Value 列）、読み書き特性（Get、Set）がリストされます。上の例では、Stepper DE1 のクラス名などの属性がリストされています。Stepper の現在時刻（CurrentTime）など、時刻とともに変化する属性値はリアルタイムに更新されます。

シミュレーション中のパラメータの変更

シミュレーション中にパラメータ（モデル中の属性の値）を変更して、その影響がモデル全体にどう波及していくかを観察したい場合があります。セッションモニタを使ってパラメータを変更するのは簡単です。

Entity の属性を変更するには、シミュレーションが停止している状況で、Entity リスト右側に Properties ペインを表示し、変更したい属性の Value を書き換えるだけです。書き換えたらリターンキーを押し、変更内容を確定します。リターンキーを押さずに次の動作を行うと、変更が反映されず、元の値に戻っていることがありますので、必ず確認してください。

変更できる属性は書き込み可能なもの（Properties ペインの W 列にチェックの入っているもの）に限られます。書き込み不可の属性については、書き換えられなくなっています。



Stepper の属性を変更する場合には、シミュレーションが停止している状況で、Stepper ウィンドウで属性を選択すると、ウィンドウ最下方にある Value 欄に選択した属性の Value が表示されますので、これを書き換えて Update ボタンを押します。

Entity と同様、変更できる属性は書き込み可能なもの（Property ペインの Set 列が + のもの）に限られます。書き込み不可の属性については、書き換えられなくなっています。

データの保存

シミュレーションを実行した際のさまざまな時刻におけるモデルの状態、モデル中のさまざまな変数の時系列をファイルに保存することができます。

モデル状態の保存

ある時刻のモデルの状態を EML ファイルとして保存することができます。方法は、モデルを読み込む際と同様に 2 種類あり、メニューバーから File → Save Model As を選択する方法と、ツールバーの Save ボタンを押す方法です。いずれの操作も実体は同一で、Session メソッド `saveModel()` を呼びだしています。
保存を実行するとファイルダイアログが開くので、ファイル名と保存するディレクトリを選択し、ファイルを保存します。保存した EML ファイルを EM 形式に変換したければ、`ecell3-eml2em` を用います。

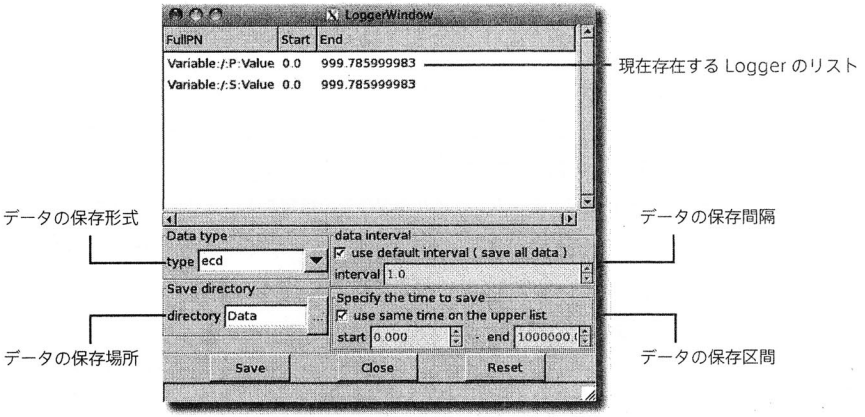
時系列の保存

GUI を通して、Logger オブジェクトに記録したデータを ECD ファイルに保存することができます。Logger や ECD についての詳細は、5章「スクリプトによるセッションの操作」をご覧ください。

Logger ウィンドウ

セッションモニタで Logger の状態を確認するには Logger ウィンドウを開きます。Logger ウィンドウは、メニューの Tools → Logger Window またはツールバーの Logger ボタンから開くことができます。下の図は、以下の操作の後（画面 A4-8 に相当）に Logger ウィンドウを開いたときの状態です。

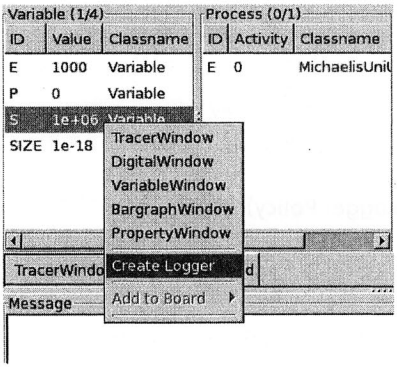
- セッションモニタを起動する。
- `simple.eml` を読み込む。
- `Variable:/:S`、`Variable:/:P` をトレーサーに登録する。
- 1000 秒のシミュレーションを実行する（interval = 1000 として、Step ボタンを押す）。



上段には、現在 Session が持っている Logger がリストされます。各行には、それぞれの Logger が記録する属性の FullPN と、データ記録が記録された最初の時刻と最後の時刻（Start、End）が表示されています。特に Logger を作成する操作は行っていませんが、2つの Variable S と P の Value に対して Logger が作られていることがわかります。これは、属性をトレーサーに登録する際に、自動的に Logger が作成されるためです（前述）。

各 Entity の規定値の属性については、トレーサーを作らず、Logger だけを作ることもできます。Entity リストで Entity を選択してコンテクストメニューを呼びだし、Create Logger を選ぶことで、Logger だけを作成できます。

Logger を作成しておけば、シミュレーション進行後にトレーサーに登録した場合でも、Logger に記録されているデータについてはトレーサー作成時点より過去まで遡ってプロットされます。



Logger ウィンドウの下段は左右に分かれています。左側は、データファイルの保存に関する設定です。

Data type はデータを保存する形式の設定で、メニューでは ecd と binary が選択できるようになっていますが、現バージョンでは binary は未サポートですので、ecd を選択したままにしてください。

Save directory は、データファイルの保存先の指定です。デフォルトでは、カレントディレクトリの直下に Data というディレクトリを作成して保存します。別のディレクトリを保存先にする場合、ここで設定します。

下段右側は保存するデータに関する設定です。Logger に記録されたデータの全部を保存するのか、一部を保存するのか、一部を保存するならどの部分を保存するのかを設定します。

data interval は Logger に記録された時系列を間引いて保存する設定を行えます。use default interval をチェックすると（デフォルトでチェックされています）、Logger が記録している時系列を間引かずにすべてファイルに保存します。間引いて保存する場合には、チェックを外して保存間隔 interval を設定します。例えば、100 に設定すると、Logger 上の時系列から、100 時点おきにデータが保存されます。データの精度は損なわれますが、ファイルの容量は大幅に小さくなります。

Specify the time to save では、保存するデータの区間を、保存開始および終了の時刻で指定できます。use same time on the upper list をチェックすると（デフォルトでチェックされています）、上段の Logger リストの Start から End まで、Logger が記録している全区間のデータを保存します。その中から一部を保存したい場合、チェックを外して保存されるデータの開始時刻（start）と、終了時刻（end）を指定してください。

ECD ファイルの保存

データを ECD ファイルとして保存するには、Data type で ecd を選択した状態で Save ボタンを押します。Save directory で指定した場所に、ECD ファイルが保存されます。これは、ECDDataFile メソッドの `save()` と等価です。

データ記録方式（Logger Policy）の設定

モデルによっては、E-Cell SE のデフォルトのデータ記録方式が不適切な場合があります。データ量が過剰、あるいは少なすぎて問題が生じるといったケースです。Logger Policy を変更することで、任意のデータ記録方式を設定することができます。Logger Policy を変更するには、メニューバーから、Preferences → Logging

Policy を選びます。下のウィンドウが現れます。この操作は、LoggerStub メソッドの `setLoggerPolicy()` と等価です。

設定は「Logging frequency」「Action if disk space runs out」「Disk space to be used for one logger」の3つの部分に分かれています。

Logging frequency では、データを記録する頻度を設定します。まず、ラジオボタンで、Log after every [　] {steps / seconds}. の steps または seconds のいずれかを選択します。

steps を選択した場合、上段のボックスに正の整数 *i* を入力してください。Logger は、シミュレータが *i* ステップ進む毎に1回、データを記録します。これは、`setLoggerPolicy()` の第1引数で最小ステップ数を指定するのと等価です。

seconds を選択した場合、下段のボックスに正の実数 *x* を入力してください。Logger は、シミュレータが *x* 秒進む毎に1回、データを記録します（厳密には *x* 秒よりもやや大きくなります）。これは、`setLoggerPolicy()` の第2引数で最小時間間隔を指定するのと等価です。

Action if disk space runs out では、ディスクの記憶容量を使い切ってしまった場合の対処方法を決めます。Throw exception を選択した場合、ディスク容量がなくなるとセッションモニタは例外を発生して停止します。Overwrite earliest data を選択すると、もっとも早い時刻のデータを新しいデータで上書きしていきます。これは、`setLoggerPolicy()` の第3引数の設定に相当します。

Disk space to be used for one logger では、Logger ひとつあたりに割り当てるディスク容量を設定します。No Limit では無制限です。Maximum [　] kilobyte では、設定した容量（キロバイト単位）が Logger ひとつあたりの最大ディスク使用量として設定されます。これは、`setLoggerPolicy()` の第4引数の設定に相当します。

設定が終了したら、OK を押して保存します。設定した内容は、次回以降セッションモニタを起動した際にも引き継がれます。

Logging frequency と data interval

Logger Policy で設定する Logging frequency と、Logger ウィンドウで設定する data interval は異なるものです。Logging Policy の Logging frequency は、Logger による記録の間隔を設定しますので、これを大きい値に設定すると、シミュレーション結果の多くが、記録されずに捨て去られます。Logger ウィンドウで設定する data interval は、Logger に記録されたデータのうちどれだけをファイルに保存するかという data interval を設定するものなので、Logger にはファイルに保存するデータ以外も記録されている場合があります。
大量の Logger を作成すると、ディスク容量を圧迫したり、計算速度が低下する場合があります。これらを軽減するためには、Logger Policy を変更してください。
Logger ウィンドウの data interval の設定は、シミュレーションの実行速度や、Logger が消費するディスク容量に影響しません。

執　筆

髙橋 恒一

内藤 泰宏

小泉 守義

協　力

牛尼 翔太

熊本 博美

佐野 ひとみ

武内 麻里亜

西野 泰子

宮部 碧

山田 一翔

謝　辞

E-Cell システムの開発の一部は、独立行政法人 科学技術振興機構 戦略的創造研究推進事業の研究領域「シミュレーション技術の革新と実用化基盤の構築」2004年度採択研究「システムバイオロジーのためのモデリング・シミュレーション環境の構築」（研究代表者：慶應義塾大学 環境情報学部、同 先端生命科学研究所 冨田 勝 教授）によって実施されました。また、本書の執筆・出版に関わる費用の一部も同事業の助成を受けています。

本書の表紙・裏表紙は David S. Goodsell 博士が public use に提供しているイラストレーションをベースに制作しました。

本書の１～７章は、以下の文書を翻訳・改変・加筆したものです。

E-Cell Fundamentals

発行日　2010年3月15日

著　者　慶應義塾大学湘南藤沢キャンパス　先端生命科学研究会

編　集　内藤 泰宏

発行所　慶應義塾大学　湘南藤沢学会

印刷所　株式会社　ワキプリントピア

ISBN978-4-87762-233-6

SFC-RM2009-004

ISBN978-4-87762-233-6
SFC-RM2009-004