Multi-algorithm and multi-timescale cell biology simulation

— Requirements analysis, algorithm design, and software implementation

Thesis by
Kouichi Takahashi

In partial fulfillment of the requirements for
the degree of Doctor of Philosophy

Keio University
Fujisawa, Japan

January 2004

Acknowledgements

I am strongly confident that this work would not have been possible anywhere other than Keio University SFC (Shonan Fujisawa Campus), Fujisawa, Japan, where a number of truly innovative ideas are constantly invented, incubated, and tested. I have been relishing a complete freedom at Dr. Masaru Tomita's lab., devoting myself to an attempt to design a new computational framework and software with numerous trial-and-errors. This work has also been benefited from excellent computer facilities of Institute for Advanced Biosciences of the university in Tsuruoka city. I would like to thank my academic supervisors and thesis committee, Dr. Tomita, Yoichi Nakayama, Yasuhiro Naito, Rintaro Saito, Koichi Furukawa and Hideyuki Tokuda.

Kazunari Kaizu contributed to development of the high-order integration scheme of the meta-algorithm through his help in implementation of Fehlberg and Dormand-Prince algorithms and design of interpolants for these modules. Current and past members of my group, including Takeshi Sakurada, Satya N.V. Arjunan, Gabor Bereczki, Tomoya Kitayama, Masahiro Sugimoto, Tatsuya Ishida, Christpher J.F. Pickett, Yohei Yamada, Daiki Ito, Kazuto Maruoka, Kentaro Takahashi, Yuki Fujita, Jun Imoto, Yusuke Saito, and Masayuki Okayama have participated in development of E-Cell System Version 3. Working with them in a team environment has been a very rewarding experience. Bin Hu kindly gave me a permission to use his *E. coli* heat shock response model to demonstrate the performance of the new computational framework proposed in this work.

Discussions with Dr. Thomas S. Shimizu had been particularily helpful and stimulating. Dr. Toshikazu Ebisuzaki introduced us a prior art in use of Hermite integration in computational astrophysics. Dr. Joerg R. Weimer made me realize the versatility of Cellular Automata.

Michiko Abe has been the greatest joy to me for over three years, and I am thankful to her for being with me. In addition to her kind help in preparing this thesis, I would like to thank Yuri Matsuzaki for educating me of happiness of life in this beautiful world.

This project was supported in part by the Japan Science and Technology Corporation, the Ministry of Agriculture, Forestry and Fisheries of Japan (Rice Genome Project), New Energy and Industrial Technology Development Organization (NEDO), and Taikichiro Mori memorial grant.

Finally, thank you to my parents and grand parents for bringing me up and supporting my decisions all the way along.

Abstract

Computational cell biology is a relatively new branch of computational sciences which sets computer simulation, as well as molecular biology, biochemistry and biophysics, at the center of its disciplines.

This thesis first inspects this new field of science, putting emphases on its computational aspects, and identifies some distinct demands in design of simulation algorithms and software platforms. One important characteristic, amongst others, of computational cell biology is that it is highly heterogeneous in terms of modeling formalisms and computational methods. Different simulation techniques are used for different types of sub-systems in the cell, and these sub-systems are often represented by different timescales. We found that the scientific necessity to integrate these computational sub-components to form higher order, multi-formalism, and composite models is becoming increasingly important.

Secondly, this thesis proposes a novel computational framework based on a discrete-event scheduler, Hermite polynomial interpolation, and object-orientation that realizes efficient multi-algorithm and multi-timescale simulations without sacrifice of modularities of sub-models running on different simulation techniques. It is demonstrated that this framework can give a significant speed-up to a real biological simulation model of *E. coli* heat shock response by coupling Gillespie's stochastic simulation algorithm and ODE solvers. It is also shown that this framework can boost a multi-timescale simulation of coupled harmonic oscillators.

Lastly, this thesis describes design and implementation issues of E-Cell System Version 3, a newly developed software environment for modeling, simulation and analysis in computational cell biology, in which the new computational framework is implemented.

論文題目

「多解法、多重時間尺度細胞生物学シミュレーション
　— 要求分析、解法設計、およびソフトウエア実装」

論文要旨

計算細胞生物学は、分子生物学、生化学、生物物理学などに加え計算機シミュレーションを核とした計算科学の比較的新しい一分野である。

この論文では、まずこの新しい分野についてその計算科学的な面に重点を置いて詳しく議論し、シミュレーション法とソフトウエア基盤の設計における要求を見極める。この分野の重要な特徴の一つは、モデリング様式および計算手法の面において非常に多様なことである。細胞内の部分系にはそれぞれ異なるシミュレーション法が用いられることがあり、またそれぞれ異なる時間尺度が伴うことも多い。これらの部分系をより高次の多様式、混成的モデルに組みたてることには科学的な必要性があることを議論する。

次に、この論文では離散イベントスケジューラー、エルミート多項式、およびオブジェクト指向に基いた新しいシミュレーションの枠組みを提案する。この枠組みは、複数の異なる解法に基き、多重の時間尺度で動作する部分系から成る混成モデルを、部分モデル間のモジュール性を損なうことなく効率よく駆動できる。大腸菌の熱ショック反応を例にとり、この枠組みが実際の生物学的な問題を従来に比べ著しく高速に解けることを示す。また、この枠組みが多重時間尺度の結合調和振動子系モデルのシミュレーションも高速化できることも示す。

最後に、この新しい枠組みを実装した計算細胞生物学におけるモデリング、シミュレーションおよび解析のためのソフトウエア環境である、E-Cell System Version 3 の設計と実装について説明する。

Contents

1 Introduction **13**

1.1 Introduction 13

1.2 Contributions 14

1.3 Organization 15

2 Requirements Analysis **16**

2.1 Simulation of cellular processes 16

2.1.1 Metabolism 17

2.1.2 Signal transduction 18

2.1.3 Gene expression 19

2.1.4 Biophysical phenomena 20

2.1.5 Summary 20

2.2 Computational cell biology 20

2.2.1 Differences from conventional computational sciences 20

2.2.2 Computational cell biology research cycle 21

2.3 Simulation methods in computational cell biology 24

2.3.1 Ordinary differential equation solvers 24

2.3.2 Special types of differential system solvers 38

2.3.3 Stochastic algorithms for mesoscopic chemical systems 39

2.3.4 Cellular automata 52

2.3.5 Other methods and summary 52

2.4 Requirements analysis 53

2.4.1 Multi-algorithm simulation 54

2.4.2 Multi-timescale simulation 55

2.4.3 Object-oriented modeling 56

2.4.4 Object-oriented simulation 58

2.4.5 Runtime user interaction 60

2.4.6 Dynamic structure model 60

2.4.7 Spatial modeling and simulation 61

2.5 Existing software platforms and projects 62

2.5.1 General purpose simulation software 63

2.5.2 Specialized simulation software 64

2.5.3 Software platforms and languages 64

2.6 Conclusion 65

3 Multi-algorithm simulation framework – the meta-algorithm 67

3.1 Introduction 68

3.1.1 Background 68

3.1.2 Classification of simulation algorithms 68

3.1.3 Multi-algorithm framework: design options 69

3.2 Algorithm 72

3.2.1 Data structure 72

3.2.2 Driver algorithm 76

3.2.3 Integration algorithm 77

3.3 Some algorithm modules 80

3.3.1 Differential equation modules 80

3.3.2 Gillespie-Gibson module 84

3.4 Results 85

3.4.1 A multi-algorithm heat-shock model 85

3.4.2 A simple model of multi-timescale oscillators 88

3.5 Discussion . . . 91

3.5.1 ODE / Gillespie composition . . . 91

3.5.2 Related works in stochastic/deterministic hybrid simulation . . . 92

3.5.3 Performance analysis of the heat-shock model . . . 93

3.5.4 Accuracy of the composite heat-shock simulation . . . 95

3.5.5 Legitimacy of the recalculation upon interruption of the Gillespie-Gibson Stepper . . . 96

3.5.6 Performance analysis of the coupled oscillator model . . . 98

3.5.7 Strengths and weaknesses of the meta-algorithm . . . 99

3.5.8 Parallelization of the meta-algorithm . . . 102

3.6 Conclusion . . . 103

4 E-Cell System 104

4.1 Architecture of E-Cell System . . . 104

4.1.1 The E-Cell Simulation Environment . . . 105

4.1.2 The E-Cell Modeling Environment . . . 108

4.2 Object-oriented implementation of the meta-algorithm . . . 110

4.2.1 Fundamental classes . . . 110

4.2.2 Overall structure of the simulator core . . . 112

4.2.3 Implementation of transition functions 113

4.2.4 Implementation of the integration algorithm 116

4.2.5 Simulation procedure 117

4.2.6 Dynamic method invocation using PropertySlot 121

4.2.7 Data logging mechanism 122

4.2.8 Implemented simulation algorithms 126

4.3 Discussion 126

5 Conclusions 128

5.1 Summary 128

5.2 Future directions 129

5.2.1 Dynamic model structure 130

5.2.2 Spatial modeling and simulation 131

5.2.3 Other possible improvements 132

5.3 Conclusion 133

A Selected Manual Chapters 134

A.1 Modeling With E-Cell 134

A.2 Scripting A Simulation Session 157

List of Figures

2.1 Research cycle of computational cell biology. 22

3.1 Multi-formalism simulation: a view from Zeigler's DEVS 69

3.2 Relation between τ and t 79

3.3 The heat-shock demonstration model 88

3.4 Simulation results of the the heat-shock model 89

3.5 The cascade oscillators demonstration model 91

3.6 Normalized comparison of distribution of σ^{32} in heat shock model. . . . 97

4.1 Overview of E-Cell System Development 105

4.2 E-Cell Simulation Session Monitor GUI 106

4.3 Three layer architecture of E-Cell kernel 107

4.4 The E-Cell Modeling Environment 108

4.5 Fundamental class diagram 111

4.6 Stepper base classes 112

4.7 Basic behaviors of Stepper base classes defined in E-Cell SE 113

4.8 Simulator overview class diagram 114

4.9 Classes implementing transition functions 115

4.10 Relations between discrete/continuous Processes and a Variable. 116

4.11 Integration mechanism class diagram 118

4.12 A diagram of classes involved in simulation 119

4.13 Simulation step sequence diagram 120

4.14 PropertiedClass class diagram 122

4.15 A schematic view of the data logging mechanism 123

4.16 Logging class diagram 124

4.17 Logging sequence diagram 125

4.18 Algorithm modules being developed for E-Cell SE 127

List of Tables

2.1 Some cellular processes and typical computational approaches. 21

2.2 Rough comparison of typical numbers characterizing various simulation targets. 23

2.3 Butcher array for Runge-Kutta Fehlberg 2(3) 33

3.1 Dense output coefficients and the Butcher array of Runge-Kutta Fehlberg 2(3) 82

3.2 Reaction list of heat shock model 87

3.3 List of initial values used in heat shock model 90

3.4 Benchmark results of the heat-shock model 90

3.5 Benchmark results of the cascade oscillators model 92

Chapter 1

Introduction

1.1 Introduction

The advent of molecular biology in the twentieth century has led to exponential growth in our knowledge about the inner workings of life. Dozens of completed genomes are now at hand, and equipped with an array of high throughput methods to characterize the way in which encoded genes and their products operate, we find ourselves asking exactly how to assemble the various pieces. The question is whether we can predict the behavior of a living cell given sufficient information about its molecular components. As with any network of interacting elements, the overall behavior becomes nonintuitive as soon as their number exceeds three. Computers have proven to be an invaluable tool in the analysis of these systems, and many biologists are turning to the keyboard. It is worth noting that the motivation of the biologist here is somewhat different from that of the biologist who turns to computing for bioinformatics (such as sequence analysis). Bioinformatics delivers additional information about the biopolymers being studied, and may provide clues as to their function. What is sought in the analysis of cellular systems is a reconstruction of experimental phenomena from the known properties of the individual molecules, and more importantly, the interactions between them. Such a model, if sufficiently detailed and accurate, serves as a reference, a guide for interpreting experimental results, and a powerful means of suggesting new hypotheses.

Modeling, simulation, and analysis is therefore perfectly positioned for integration into the experimental cycle of cell biology. In addition to demystifying nonintuitive phenomena, simulation allows experimentally unfeasible scenarios to be tested, and has the potential to seriously reduce experimental costs. Although "real" experiments will always be necessary to advance our understanding of biological processes, conducting "*in silico*" experiments using computer models can help to guide the wet lab process, effectively narrowing the experimental search space.

The process of building *in silico* models of cells contrasts with the traditional hypothesis-driven research process in biology. Modeling can be described as a holistic approach that opposes the reductionism so widespread in biology. Molecular and cellular biologists have been overwhelmingly successful in identifying, purifying, and characterizing molecules crucial to specific cellular functions. However, results from genome projects reveal that most organisms contain a surprisingly small number of genes, at least relative to the complexity of the phenotype. This provides a striking demonstration of the nontrivial nature of molecular interactions in the cell — the whole, quite simply, is much more than the mere sum of its parts.

1.2 Contributions

The main contribution of this thesis is the development of a new, multi-algorithm and multi-timescale computational framework called the 'meta-algorithm'. On this framework, virtually any, discrete/continuous, stochastic/deterministic, simulation algorithms can be implemented and used in any combination in simulations.

Secondly, E-Cell Simulation Environment, a simulation software for computational cell biology, is developed in a fully object-oriented fashion based on the meta-algorithm framework.

As the rationale of development of these new computational framework and software, this thesis also argues that computational cell biology has some major features those distinguish itself from conventional computational sciences such as computational physics and biochemical simulations.

1.3 Organization

Chapter 2 discusses a domain analysis of computational cell biology from the viewpoint of simulation algorithm design and software engineering, and we identify seven 'desirable features' of cellular simulation software. Chapter 3 defines a new computational framework (the meta-algorithm). We use two demonstration models, a Gillespie/ODE composite *E. coli* heat-shock model and a multi-timescale coupled harmonic oscillator model, and find that the meta-algorithm can successfully combine different simulation algorithms and timescales yielding considerable performance improvements. This chapter also has some discussions on possible design options for multi-algorithm simulators. The implementation of the meta-algorithm on E-Cell Simulation Environment Version 3 is discussed in Chapter 4. E-Cell Simulation Environment is part of E-Cell System, a generic software suite for modeling, simulation and analysis of cellular systems.

Chapter 2

Requirements Analysis

This chapter investigates the cell as a target of simulation, and discusses computational challenges that it poses. We identify required features of cell biology simulators that can be used to reconstruct various levels of physical phenomena in the cell.

2.1 Simulation of cellular processes

The cell is a big system in terms of the number and the diversity of physical phenomena that constitutes its internal dynamics. The small number of genes in most organisms implies that molecular phenomena in the cell are, to say the least, nontrivial. Collective knowledge of parts of the system itself does not directly lead to understanding of the cell as a whole. Necessity and importance of cell simulation as a research method arises here; putting the data into databases is not enough, only a more constructive approach with computer modeling and simulation can provide a way to understand the cell as a system.

In this section, we review such chemical and physical phenomena and discuss some possible computational approaches to simulate these systems.

2.1.1 Metabolism

Energy metabolism is the best characterized part of all cellular behavior, and is particularly well understood in human red blood cells. Given that an erythrocyte is devoid of nuclei and other related features, it serves as an ideal model system for studying metabolism in isolation — this cell is, essentially, “a bag of metabolism”. Biochemists have succeeded in collecting enough quantitative data to allow kinetic models of the entire cell to be constructed, and there is a long history of computer simulations dating back to the 1960s (Zajicek, 1968). These metabolic models, and many others, typically comprise a set of ordinary differential equations (ODE) that describe the rates of enzymatic reactions. These can be solved by numerical integration, for which several well-established algorithms exist.

Modern metabolic pathway models typically consist of primary state variables for molecular species concentrations, one ODE for each enzyme reaction, and a stoichiometric matrix. The rate equations of most modern enzyme kinetics models are derived using the King-Altman method (King and Altman, 1956) , which is a generalized version of the classical formulation of Michaelis and Menten, the Michaelis-Menten equations (Michaelis and Menten, 1913). Additional algebraic equations are commonly employed as constraints on the system. Thus, most metabolism models are described as differential-algebraic equations (DAE). The origin of numerical methods for solving ODEs can be traced back to Euler’s work in the eighteenth century. Variations of the Runge-Kutta method are generally used for simulations. Implicit variations of the methods, are often used for ’stiff’ systems which involve a wide range of time constants. See the following section for discussions about stiffness of ODEs. Although there have been certain major advances in the last few decades, the essence of the numerical algorithms for solving initial value problems of ODE systems was described by Gear in 1971 (Gear, 1971). Press *et al.* also give an introduction to the topic in *Numerical Recipes in C* (Press, 1988). The theoretical and practical bases of simulating metabolic pathways are therefore quite well grounded. However, the design and implementation of simulation software and model construction methods, which this thesis attempts to highlight, are still under active discussion.

Metabolism, of course, is not the only function of the cell, and we must not forget that cells have other important roles such as gene expression, signal transduction, motility, vesicular transport, cell division and differentiation. Although quantitative data on these processes are still relatively sparse compared to erythrocyte energy metabolism, certain systems have been modeled with considerable success. Examples include the simulation of gene expression in phage-λ (McAdams and Shapiro, 1995) and the signal transduction pathway controlling bacterial chemotaxis (Bray *et al.,*, 1993).

2.1.2 Signal transduction

Signal transduction pathways constitute an example of systems with characteristics that prevent the simple application of ODE-based modeling. These pathways are ordinarily composed of much fewer numbers of reactant molecules than metabolic systems, and the underlying stochastic behavior of the molecular interactions becomes evident. Accordingly, there have been attempts to model signal transduction pathways with stochastic computation (Morton-Firth and Bray, 1998; Morton-Firth *et al.,*, 1999) instead of deterministic methods (*e.g.* use of ODEs).

Recently, it has been revealed that two-dimensional stochastic coupled lattice model of receptor localization of *E. coli* chemotaxis signaling better explains the system's hyper sensitivity to stimuli that could not be reproduced by conventional compartmental simulations (Shimizu *et al.,*, 2003).

2.1.3 Gene expression

Gene expression systems, like signaling pathways, tend to be composed of a small number of molecular entities, which include transcription factors, polymerases, and genes. These low copy number molecules orchestrate gene expression in a highly stochastic fashion. For example, the initial phase of gene expression is remarkable because its stochastic behavior has binary consequences: binding of a rare transcription factor and a single gene in a cell can determine whether the gene is turned on or off. It seems that in many cases, gene expression systems might best be modeled with stochastic equations, although there are many other ways to model this phenomena, depending on the modeler's interests. Examples include ODE models (*e.g.* linear models and mass action models), S-System models (Savageau, 1976), and binary and multi-reaction models (Hasty *et al.,*, 2001).

Another characteristic of gene expression systems is the richness of interaction with other cellular processes. These systems can control metabolic flux by changing the concentrations of enzymes, and at the same time being regulated by signaling proteins. Chromosomal structure is dynamically regulated by DNA binding factors, which are in turn derived from other genes. When a whole cell is modeled, elements in the gene expression system sometimes need information about elements in other systems, in order to allow cross-system interactions. Integration of gene expression and other systems at the whole cell level might best be accommodated by object-oriented data structures, as previously described in Hashimoto *et al.,* (1999).

2.1.4 Biophysical phenomena

All of the above simulation examples treat the properties of protein binding and enzyme kinetics reactions. However, certain cellular processes such as cytoskeletal movement and cytoplasmic streaming, need to be modeled at the biophysical level. Cytoplasmic streaming involves diffusion of relatively heavy proteins, whereas the movement of the cytoskeleton causes structural changes, including cell division and cell differentiation. Simulations of these phenomena have been carried out since the 1970s (Novitski and Bajer, 1978). Studies have become more precise with time, concomitant with the increase in our depth of understanding at the molecular level (Gibbons *et al.*,, 2001).

2.1.5 Summary

A few types of cellular processes and typical computational approaches are shown in Table 2.1. The interested reader is referred to two recent reviews which address some of the issues raised thus far: Tyson *et al.* provide an excellent review of computational cell biology with an emphasis on cell-cycle control (Tyson *et al.*,, 2001), while Phair and Misteli review the application of kinetic modeling methods to biophysical problems (Phair and Misteli, 2001).

2.2 Computational cell biology

2.2.1 Differences from conventional computational sciences

Software development for non-trivial cell simulation projects is a notably expensive process. For research projects in the traditional computational sciences, where brute force computation remains operative, it is reasonable to develop new software for each

Table 2.1: Some cellular processes and typical computational approaches.

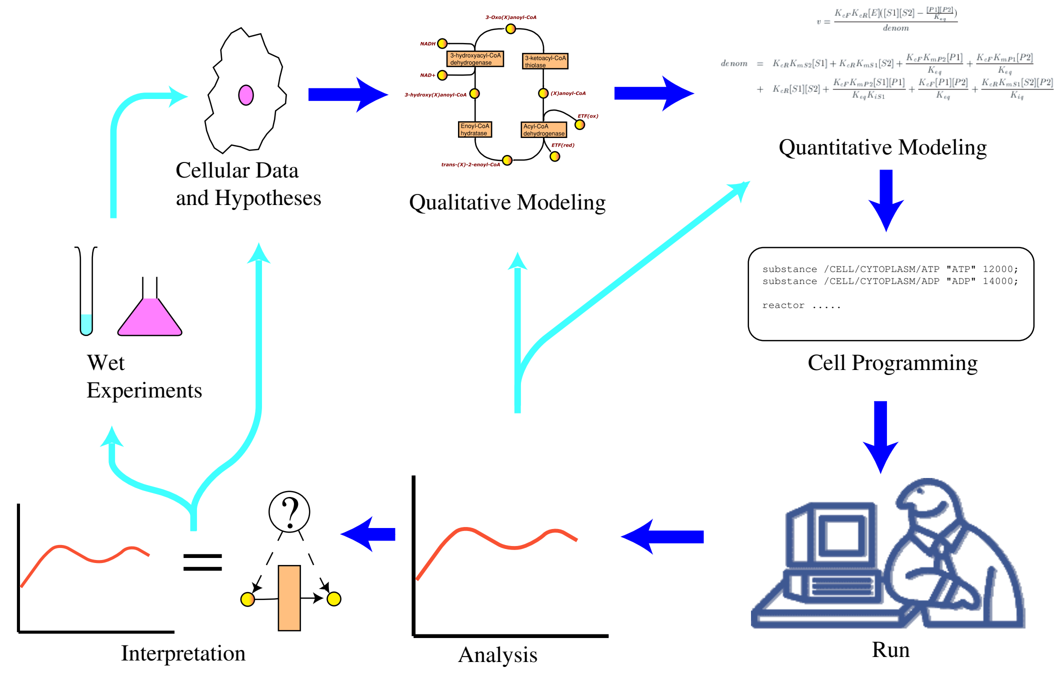
Process type	Dominant phenomena	Typical computation schemes
Metabolism	Enzymatic reaction	DAE, PL, FBA, CA.
Signal transduction	Molecular binding	DAE, stochastic algorithms (StochSim, Gillespie), diffusion-reaction.
Gene expression	Molecular binding, polymerization, degradation	OOM, PL, DAE, boolean networks, stochastic algorithms.
DNA replication	Molecular binding, polymerization	OOM, DAE.
Cytoskeletal dynamics	Polymerization, mechanical forces	DAE (including mechanical models), particle dynamics, OOM.
Cytoplasmic streaming	Streaming	CA (e.g. lattice Boltzman), PDE.

Abbreviations: CA, Cellular Automata; DAE, differential-algebraic equations (rate equation-based systems); FBA, flux balance analysis; OOM, object-oriented modeling; PL, power-law, such as S-System and Generalized Mass Action.

project, sometimes in a disposable fashion. Most of the traditional computational science fields like computational physics are characterized by numerous uniform components and a limited number of simple principles. Cell simulation, in contrast, involves numerous and various components with different properties, interacting in diverse, complicated manners. Typical characteristics of several simulation targets are summarized in Table 2.2. The design and implementation of simulation software inevitably reflects the complexity of the problem.

2.2.2 Computational cell biology research cycle

We envision a research cycle of cell biology that incorporates biosimulation technology (Figure 2.1). Every step of the cycle completed outside of the wet lab depends upon sound methods in software engineering. Consider the storage, processing, and utilization of massive amounts of biological knowledge: only through integration of an intelligent modeling environment with sophisticated data and knowledge bases can the challenge of modeling a very large and complex system be accommodated. Although



```
substance /CELL/CYTOPLASM/ATP "ATP" 12000;
substance /CELL/CYTOPLASM/ADP "ADP" 14000;

reactor .....
```

Figure 2.1: Research cycle of computational cell biology.
The wet lab process is extended to include simulation software for a computational cell biology research project. Qualitative models (*e.g.* pathway maps) are built from *in vivo* and *in vitro* data and hypotheses, or a reference model (Qualitative Modeling). Then, quantitative characterization of cellular properties facilitates the transition to a mathematical system model (Quantitative Modeling). The numerical and discrete properties of the quantitative model are translated into a modeling language (Cell Programming), and the systemic behavior is predicted (Run). Results are then analyzed to suggest new hypotheses (Analysis and Interpretation). Any acquired hypothesis is subsequently tested by wet experiments, and the cycle begins anew.

this thesis mainly considers the first half of the cycle (from 'Qualitative Modeling' to 'Run'), a technological stagnance in any one of the steps may form a bottleneck, and thus threaten the evolution of computational cell biology.

Table 2.2: Rough comparison of typical numbers characterizing various simulation targets.

Target	Compartments	Components	Component types	Interaction modes
Prokaryotes (*E.coli*)	$\sim 10^1$	$\sim 10^{13-14}$ molecules $\sim 10^{3-4}$ species	$\sim 10^1(1)$	$\sim 10^{1-3}(2)$
Eukaryotes (*H. sapiens*)	$\sim 10^{3-4}$	$\sim 10^{17-18}$ molecules $\sim 10^{4-5}$ species	$\sim 10^1(1)$	$\sim 10^{1-4}(2)$
LSI (Electronic circuit)	usually 1	$\sim 10^{6-7}$	a few	1
CFD (Fluid dynamics)	usually 10^{0-1}	$\sim 10^{5-6}$	1	1
MD (Molecular dynamics)	1	$\sim 10^{2-6}$	a few	a few

Some typical numbers, which determine computational hardware and software requirements, are compared among several simulation targets. Large numbers of compartments, component types and interaction modes characterize cell simulation. Notes: (1) Component types which require different data structures or object classes are counted. For example, a 'membrane' object needs a different object class from protein molecules. Different molecular species, however, are not counted. (2) This number depends on whether or not different enzyme kinetics equations, which are roughly proportional to the number of enzyme encoding genes, are counted as different interaction modes. Interaction modes other than enzymatic reactions include molecular bindings (complex formations), molecular collisions, DNA replication, cytoplasmic streaming, cytokinesis, and vesicular trafficking.

2.3 Simulation methods in computational cell biology

As we have seen, simulation of the cell requires heterogeneous approaches according to the levels of abstraction, scales, and types of information available to construct simulation models. Here we briefly review some commonly used numerical simulation techniques.

2.3.1 Ordinary differential equation solvers

Ordinary differential equations (ODEs) is one of the most popular ways of describing continuous dynamical systems. A distinct strength of this formalism is that, with its well-established theory in numerical treatments and availability of high-performance generic solvers described in the following, it can represent virtually any continuous and deterministic dynamics elegantly.

In computational cell biology, the most popular use of ODE formalism is the macroscopic representation of chemically reacting systems by a means of kinetic rate equations. Elementary and some simple reactions are represented in a form of mass-actions, for example,

$$X_1 + X_2 \rightarrow X_3, \tag{2.1}$$

and it can be formulated by using the following differential equation

$$-\frac{d}{dt}[X_1] = -\frac{d}{dt}[X_2] = \frac{d}{dt}[X_3] = k \cdot [X_1][X_2], \tag{2.2}$$

where X_n is the n-th chemical species ($[\cdot]$ denotes concentration), and k is the rate constant. Complex enzymatic reactions are often modeled by using Michaelis-Menten and King-Altman types of rate equations. The Michaelis-Menten equation corresponding to the following simplest enzymatic reaction mechanism

$$E + S \rightleftarrows ES \rightarrow E + P, \tag{2.3}$$

is

$$-\frac{d}{dt}S = \frac{d}{dt}P = \frac{K_{cat} \cdot E_T \cdot [S]}{K_m + [S]}, \quad E_T = E + ES, \tag{2.4}$$

where S, P, and E are the substrate, product, and enzyme, respectively, and K_{cat} is the catalytic constant (the turnover number of the enzyme), and K_m is the Michaelis constant.

Unlike some of other specialized formalisms introduced in the following sections, most differential equation solvers are generic, and can handle a variety of linear and non-linear equations employed in cell biology. Some examples of representations of those phenomena other than rate equations include dynamic changes in environmental factors such as thermodynamic parameters temperature, pH, and volume.

Initial value problems of ODEs

A system of ODE has a general form like this:

$$\frac{d}{dt}\mathbf{x} = \mathbf{f}(t, \mathbf{x}), \quad \mathbf{x}(t_0) = \mathbf{x}_0 \tag{2.5}$$

where $\mathbf{x}$ is a vector of dependent variables, $\mathbf{f}$ is a vector of derivative functions, and t is an independent variable. In time-driven simulations, the independent variable t is usually time. The system is *autonomous* if the system does not explicitly depend on the independent variable t;

$$\frac{d}{dt}\mathbf{x} = \mathbf{f}(\mathbf{x}) \tag{2.6}$$

Time simulation of ODE systems is equivalently reformulated as solving 2.5 for $\mathbf{x}(t)$, where $t \in \mathbb{R}^{+,0}$.

Taylor expansion of $\mathbf{x}$ in 2.5 at time t_0 gives

$$\mathbf{x}(t) = \mathbf{x}(t_0) + \sum_{i \in \mathbb{N}} \frac{\Delta t^i}{i!} \cdot \frac{d^i}{dt^i}\mathbf{x}(t_0), \quad \Delta t = t - t_0 \tag{2.7}$$

or,

$$\mathbf{x}(t) = \mathbf{x}(t_0) + \sum_{i \in \mathbb{N}} \frac{\Delta t^i}{i!} \cdot \frac{d^{(i-1)}}{dt^{(i-1)}}\mathbf{f}(t_0), \quad \Delta t = t - t_0 \tag{2.8}$$

If analytical differentiation of the system $\mathbf{f}$ to arbitrary high order can be derived, this Taylor expansion immediately gives the solution, or the simulation trajectory, $\mathbf{x}(t)$. Practically, however, ODE representations of biological problems often make use of nonlinear equations, and it is very hard to construct a general method of solving these equations analytically. Thus use of iterative numerical methods is the norm. This class of problem is called the *initial value problem* (IVP) of ODEs.

Numerical solution of ODEs has a considerable history, and the oldest and simplest method was given by Euler in 1768. From the definition of derivatives,

$$\frac{d}{dt}\mathbf{x} = \lim_{\Delta t \to 0} \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{t_{n+1} - t_n}, \tag{2.9}$$

where

$$\Delta t = t_{n+1} - t_n, \tag{2.10}$$

Now for sufficiently small Δt we can assume that this formula gives an approximation of the derivatives at the time point t_n,

$$\left.\frac{d\mathbf{x}}{dt}\right|_{t_n} (= \mathbf{f}_n = \mathbf{f}(t_n, \mathbf{x}_n)) \simeq \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{t_{n+1} - t_n}. \tag{2.11}$$

Comparing this with 2.5,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \cdot \mathbf{f}(t_n, \mathbf{x}_n) \tag{2.12}$$

we now get the *explicit Euler method*. It is called explicit because no unknown appears in the right hand side (RHS) of the equation 2.12.

Consistency and Convergence

Two requisite properties of a numerical method to be useful are *consistency* and *convergence*.

A numerical method is *consistent* when the local truncation error,

$$\mathbf{e} \equiv \left|\mathbf{x}_{<t_n>}(t_{n+1}) - \mathbf{x}_{n+1}\right|, \tag{2.13}$$

diminishes to zero as the step size decreases:

$$\lim_{\Delta t \to 0} \mathbf{e} = \mathbf{0}, \tag{2.14}$$

where $\mathbf{e}$ is a vector of the local truncation errors of each variable of the system $x_i \in \mathbf{x}$. Here $\mathbf{x}_{<t_n>}(t_{n+1})$ is the exact solution with the initial condition $\mathbf{x}(t_n)$ at time t_n. Proof of consistency of the explicit Euler method is trivial. Using 2.10, 2.11, and assuming $\mathbf{x}_n = \mathbf{x}(x_{t_n})$, the local truncation error of the method is obtained from 2.12 as

$$\begin{aligned}
\mathbf{e}^{euler} &= \left|\mathbf{x}(t_{n+1}) - \mathbf{x}^{euler}_{n+1}\right| \\
&= \left|\mathbf{x}(t_n) + \sum_{i \in \mathbb{N}} \frac{\Delta t^i}{i!} \cdot \frac{d^{(i-1)}}{dt^{(i-1)}}\mathbf{f}(t_n) - \mathbf{x}(t_n) - \Delta t \cdot \mathbf{f}(t_n)\right| \qquad (2.15) \\
&= \left|\sum_{\{i : i \in \mathbb{N} \wedge i \geq 2\}} \frac{\Delta t^i}{i!} \cdot \frac{d^{(i-1)}}{dt^{(i-1)}}\mathbf{f}(t_n)\right|. \qquad (2.16)
\end{aligned}$$

It can be verified that $\mathbf{e}^{euler} \to 0$ as $\Delta t \to 0$, hence the explicit euler method is consistent. Third and higher components are ignorable when Δt is sufficiently small, thus:

$$\mathbf{e}_{euler} \simeq \Delta t^2 \left| \frac{1}{2} \cdot \frac{d}{dt} \mathbf{f}(t_n) \right|. \tag{2.17}$$

Consequently, it can be seen the error of the method has an order of Δt^2. Thus, a more formal formulation of the method comes with the local error term of $O(\Delta t^2)$:

$$\mathbf{x}(t_{n+1}) = \mathbf{x}_n + \Delta t \cdot \mathbf{f}(t_n, \mathbf{x}_n) + O(\Delta t^2). \tag{2.18}$$

In simulation, the *global truncation error*, which indicates accumulation of the local error after certain period of time, is of practical importance. The global truncation error of a method is given by

$$\mathbf{E} \equiv |\mathbf{x}(t_i) - \mathbf{x}_i|. \tag{2.19}$$

For simplicity, assuming the step size Δt is constant, the simulation requires $(\Delta t)^{-1}$ iterations for a unit of time. Thus, the accumulation of the error can be denoted as

$$\mathbf{E} = (\Delta t)^{-1} \cdot \mathbf{e}. \tag{2.20}$$

It can be read that generally the global truncation error is of order p with respect to the step size Δt, when the local truncation error is $O(\Delta t^{p+1})$. If a method has the global truncation error of $O(\Delta t^p)$, then it is said to be consistent with an order of accuracy p. The explicit Euler method therefore has an consistency of the first order.

One of the most important design goals of numerical methods is to get a good *convergence* with as small as possible computation cost. A method is said to be convergent if

$$\forall i : \lim_{\Delta t \to 0} |\mathbf{x}(t_i) - \mathbf{x}_i| = \mathbf{0}. \tag{2.21}$$

Non-convergent methods are of no use because qualities of outcomes of simulations are not assured.

Higher order methods

Computation cost of numerical methods is proportional to the inverse of the step size, $1/\Delta t$. One way of increasing the step size without a directly proportional increase in the error is use of methods of higher order consistency. Many algorithms of the second, higher, and variable consistency order have been proposed and being used, and most of them can be classified into two categories: single- and multiple-step methods. Multi-step methods make use of information calculated in some past steps to conduct the current simulation step. Single-step methods are 'closed' in this sense; these methods utilizes only the current state of the system. Although historically the multi-step methods once had been a standard, and many popular software packages including LSODE and DASSL are based on this class of methods, recent advancements in single-step methods, especially variants of the Runge-Kutta method, is changing the picture. When applications in computational cell biology simulation is under consideration, single-step methods have some favorable features over the multiple-step methods; it is (1) easier to implement the real-time user interaction efficiently, and (2) better suited in uses in conjunction with other algorithms in multi-formalism simulations (see the next 'requirements analysis' and the following 'multi-algorithm method' sections). Also, it often results in simpler implementation, and unlike multi-step methods, no special procedure is necessary in simulation start up. For those reasons, this paper mainly discusses about the Runge-Kutta methods.

The general form of a s-stage, single-step Runge-Kutta method is:

$$\mathbf{k}_j = \mathbf{f}\left(t_n + c_j\ \Delta t, \mathbf{y}_n + \Delta t \cdot \sum_{i=1}^{s} a_{j,i}\mathbf{k}_i\right),\ j = 1 \ldots s$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot \sum_{j=1}^{s} b_j \mathbf{k}_j \tag{2.22}$$

where a, b, and c are called Runge-Kutta coefficients, or collectively Butcher array.

Setting $s = 2$, a second-order explicit Runge-Kutta method can be derived in this way:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \cdot (b_1\mathbf{k}_1 + b_2\mathbf{k}_2), \tag{2.23}$$

$$\mathbf{k}_1 = \mathbf{f}(t_n, \mathbf{x}_n), \quad \mathbf{k}_2 = \mathbf{f}(t_n + c_2\ \Delta t, \mathbf{x}_n + \Delta t\ \mathbf{k}_1\ a_{2,1}), \tag{2.24}$$

where $a_{2,1}$, b_1, b_2, and c_2 are the Runge-Kutta coefficients to be decided. Now Taylor series expansion of $\mathbf{k}_2$ to the first order gives

$$\mathbf{k}_2 = \mathbf{f}_n + c_2\Delta t\ \left.\frac{\partial \mathbf{f}}{\partial t}\right|_{t_n} + \Delta t\ \mathbf{k}_1\ a_{2,1}\left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{t_n}. \tag{2.25}$$

Putting 2.25 into 2.23,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + (b_1 + b_2)\Delta t\ \mathbf{f}_n + \Delta t^2 \left(b_2 c_2 \left.\frac{\partial \mathbf{f}}{\partial t}\right|_{t_n} + b_2 a_{2,1}\mathbf{f}_n \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{t_n} \right). \tag{2.26}$$

Now we want to compare this with the Taylor series 2.8, which, curtailing the third and higher order terms, becomes:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t\mathbf{f} + \frac{\Delta t^2}{2}\frac{d^2}{dt^2}\mathbf{f} \tag{2.27}$$

$$= \mathbf{x}_n + \Delta t\mathbf{f} + \frac{\Delta t^2}{2}\left(\frac{\partial \mathbf{f}}{\partial t} + \mathbf{f}\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right). \tag{2.28}$$

Then we now have three equations for the four coefficients;

$$b_1 + b_2 = 1, \quad b_2 c_2 = 1/2, \quad b_2 a_{2,1} = 1/2. \tag{2.29}$$

These relations are under determined, and there can be infinite number of second-order explicit Runge-Kutta methods. For example, setting $b_1 = 0$ gives the *midpoint method*

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \cdot \mathbf{f}\left(t_n + \frac{\Delta t}{2}, \mathbf{x}_n + \Delta t\, \frac{\mathbf{f}(t_n, \mathbf{x}_n)}{2}\right). \tag{2.30}$$

In this way, higher order methods of arbitrary high order can be derived from comparison between 2.22 and the Taylor series. Use of higher order methods is preferable. As it can be understood from the equation 2.8 of the Taylor series, the scale of error term decreases rapidly with regard to the order of the method, and we can take exponentially large step sizes with the same level of truncation error. This in turn means that we can complete the simulation with less number of steps, and therefore less accumulation of round-off error.

It is known, however, the fourth order is a kind of optimum and most frequently used. Up to the fourth order it just requires the same number of stages of right-hand side evaluations as the consistency order, while fifth and higher order methods involve more stages than the order of the method. For example, at minimum 6 stages are necessary for the fifth order method, $s = 7$ for sixth order, $s = 9$ for seventh, and $s = 11$ for eighth.

Error control

A commonly used error control scheme of numerical methods is based on the local truncation error as follows:

$$\forall i : |e_i| \leq safety \cdot (\sigma_{abs} + \sigma_{rel} \cdot (a|x_i| + b\Delta t|f_i|)), \tag{2.31}$$

where $e_i \in \mathbf{e}$ is the error of this step of the variable x_i, $safety$ is a safety factor (usually $\sim 110\%$), σ_{abs} and σ_{rel} are absolute and relative error tolerances, a and b are scaling factors for the value and derivative of the variable, respectively. In each step, numerical methods control the error by means of the step size and other parameters such as the order of the calculation. If the error of at least one variable violates the criteria 2.31, the solver rejects the step and redos the calculation with a shrunken step size. Conversely, if the error is sufficiently smaller (say, $\sim 50\%$) than the right hand side of 2.31, the step size can be elongated to reduce the computation cost. There can be many strategies of deciding step sizes. Two most frequently used methods are step halving/doubling and variations of the following basic equation

$$\Delta t_{new} = \Delta t \left| \frac{Tolerance \cdot x_n}{e_n} \right|^{1/s}, \tag{2.32}$$

where $Tolerance$ is the right hand side of 2.31, and n is the index of the variable which gave the maximum error in the current step. It must be noted that generally it is impossible to obtain the exact values of the local truncation errors, and an integrator must somehow numerically estimate the e_n.

A frequently used strategy to estimate the local error is to have a pair of Runge-Kutta calculations of orders p and $p+1$. The difference between solutions from these calculations gives a good approximation of the local error, because as the Taylor series expansion implies, the value of error term diminishes rapidly to the order. A neat trick to obtain this estimation of the local truncation error efficiently is the *embedded Runge-Kutta* method, devised by Fehlberg (Fehlberg, 1969). The underlying idea is to embed a Runge-Kutta calculation of the order p into that of the order $p + 1$. Table 2.3.1 is the Butcher array of a method called Fehlberg 2(3), that specifies Runge-Kutta coefficients. Calculating the equation 2.22 using the coefficient in the b_i and b_i^* columns give the second and the third order solutions, respectively, and it requires only three RHS evaluations.

Table 2.3: Butcher array for Runge-Kutta Fehlberg 2(3)

	0			a_{ij}
c_j	1	1		
	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	
	b_i	$\frac{1}{2}$	$\frac{1}{2}$	0
	b_i^*	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{2}{3}$

Stability and stiffness

Some differential systems are *stiff*. Although stiffness is not defined in a mathematically rigid way, here we give it a casual definition as follows: when the system has at least two very different time scales, *and* the trajectory is dominated by the slow movement, then it is stiff. To put this in a bit more formally, if the fast mode of the system has a stable manifold, and the state point of the system is captured by it, the system is stiff. Thus a system can become stiff and non-stiff according to where the state point is. For example, when the system is making a transition from one state point to a distant stable manifold, it is non-stiff, while as long as the trajectory is on the manifold, it is said to be stiff.

Stiffness can be understood in terms of the *Jacobian* matrix which is:

$$J = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}. \tag{2.33}$$

Stiffness is sometimes defined as

$$\left| \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right| \Delta t \cdot C \gg 1. \tag{2.34}$$

where C is a positive constant which represent a typical number of simulation steps (say, 10^6 or 10^{12}).

When the system is stiff, all the explicit methods explained so far experience hardship. Consider the exact solution at time t_n, $\mathbf{x}(t_n)$, on the slow manifold. All the explicit methods primarily uses values of the derivative functions at the current state point of the numerical solution, $\mathbf{x}_n$, to estimate the state of the system after the some amount of time Δt. Any numerical computation involves some amount of error ϵ, and it puts the numerical solution slightly off the slow manifold, thus, $\mathbf{x}_n = \mathbf{x}(t_n) + \epsilon$. When the Jacobian is very large, this error ϵ magnifies the error in the next numerical solution point $\mathbf{x}_{n+1}$, and the assumption behind 2.20, that is, the global error is a simple accumulation of the local error, no longer holds. Although the convergence of the computation 2.21 itself is not necessarily affected, the effectiveness of the error control scheme in 2.31 is destroyed. Because slow manifolds are often stable, the fast flows toward the center of that manifold appear in values of Jacobian around there, and in this case the trajectories disastrously show diverging oscillations. Even if the solver managed to detect the error in the step, it results in frequent step rejections, and forces the solver to take extremely small step sizes to converge.

A measure of tolerance against stiffness is *stability* of numerical methods. Consider a scalar system

$$\frac{dx}{dt} = f(x). \tag{2.35}$$

If the system is linear,

$$\frac{dx}{dt} = \lambda x, \tag{2.36}$$

the stability of a method is defined by the stability function $R(\cdot)$, which is defined as

$$x_{n+1} = R(\Delta t \cdot \lambda)x_n. \tag{2.37}$$

In the case of the explicit Euler method it is,

$$x_{n+1} = (1 + \Delta t \cdot \lambda)x_n. \tag{2.38}$$

Setting $\Delta t \cdot \lambda = z$,

$$x_{n+1} = (1 + z)x_n. \tag{2.39}$$

Thus the stability function of the method is

$$R(z) = 1 + z. \tag{2.40}$$

Now the stability region of the method is given by

$$|R(z)| = |1 + z| \leq 1, \tag{2.41}$$

and it is shown that the explicit Euler method is stable only in a very narrow range $-2 \leq \Delta t \cdot \lambda \leq 0$. More generally, when it is a linear vector system, the constant λ is an eigen value of the constant coefficient matrix A in

$$\frac{d\mathbf{x}}{dt} = A \cdot \mathbf{x}, \tag{2.42}$$

and thus a complex number.

Use of implicit methods overcomes this stability problem. The implicit Euler method, in a scalar form, has the following form:

$$x_{n+1} = x_n + f(t, x_{n+1}). \tag{2.43}$$

Putting the linear scalar system 2.36 into this and rearranging to the form of the stability function 2.37 gives the stability function of this method:

$$x_{n+1} = \left(\frac{1}{1-z}\right) x_n. \tag{2.44}$$

Therefore this method is stable in the region $|1 - z| \leq 1$, or, it is stable in an open region except for the domain $0 < \lambda \cdot \Delta t < 2$ in the case of the scalar system.

If the stability region of a method includes the whole left-half of the complex plane of z, which is the case of the implicit Euler method, it is called absolutely stable, or *A-stable*. That is, when the real part of λ is zero or negative, the method is guaranteed to be stable. Similarly, if the same condition is satisfied for general non-linear systems, it is said to be *B-stable*. The order of the global error of the method for the general non-linear systems is called *B-convergence*.

The exact solution of the linear problem 2.36 at the next time point t_{n+1} is easily derived as $e^{\lambda \Delta t} x_n$. Now recall the definition of the local error 2.13 again, and using 2.37:

$$e_{n+1} \equiv \left| \left(e^z - R(z)\right) x_n \right|. \tag{2.45}$$

A method is called *stiffly accurate* if

$$\lim_{|z| \to -\infty} \left| \left(e^z - R(z)\right) x_n \right| = 0. \tag{2.46}$$

Now check if the implicit Euler method is stiffly accurate:

$$\lim_{|z|\to-\infty} \left| e^z - \frac{1}{1-z} \right| = 0. \tag{2.47}$$

If a method is both A-stable and stiffly accurate, it is said to be *L-stable* or stiffly A-stable or strongly A-stable. The point here is that no matter how large the damping force of the system is (even in the case it is infinite), the numerical solution does not, at least, diverge. This property is important in very stiff problems and some DAE systems. It is desirable that all numerical methods used are L-stable, but only a few A-stable methods are known to be L-stable.

Implicit methods

Implicit Euler method imposes solution of non-linear equations to conduct a step of computation. Newton's method is most popularly used for this purpose. A Newton iteration to get x_{n+1} of 2.43 using the first order Taylor series expansion is:

$$\mathbf{x}_{n+1}^{m} = \mathbf{x}_n + \Delta t f(t_{n+1}, \mathbf{x}_{n+1}^{m-1}) + \Delta t \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{(t_{n+1}, \mathbf{x}_{n+1}^{m-1})} \left(\mathbf{x}_{n+1}^{m} - \mathbf{x}_{n+1}^{m-1}\right), \tag{2.48}$$

or in a programmable form,

$$\begin{aligned} \mathbf{x}_{n+1}^{m} = & \left[\mathbf{x}_n + \Delta t f(t_{n+1}, \mathbf{x}_{n+1}^{m-1}) - \Delta t \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{(t_{n+1}, \mathbf{x}_{n+1}^{m-1})} \mathbf{x}_{n+1}^{m-1} \right] \\ & \cdot \left(\mathbf{I} - \Delta t \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{(t_{n+1}, \mathbf{x}_{n+1}^{m-1})} \right)^{-1} \end{aligned} \tag{2.49}$$

where m is the counter of the Newton iteration, and $\mathbf{I}$ is an identity matrix. The iteration is terminated when the difference $\left|\mathbf{x}_{n+1}^{m} - \mathbf{x}_{n+1}^{m-1}\right|$ goes below a pre-defined threshold. In the same way, it is possible to derive implicit variations of higher order methods.

Despite their good stability of implicit methods that is necessary to solve stiff systems efficiently, a drawback is computation cost. In addition to that of its explicit counterpart, it requires m iterations of 2.49 involving a calculation of the Jacobian matrix and a matrix inversion. Generally, in non-linear computational cell biology problems, Jacobian cannot be obtained analytically, and numerical differentiation is necessary. Although the precision of this calculation of Jacobian does not affect the accuracy of the simulation itself as long as the Newton iteration converges, the cost of this computation is not negligible. The computation cost of a matrix inversion is in the order of $O(N^3)$, where N is the size of the matrix. N becomes proportionally bigger when higher order methods such as implicit Runge-Kutta are used.

Therefore, a good ODE simulator is supposed to be able to adaptively switch between explicit and implicit methods automatically detecting stiffness of the system. The best combination that this thesis suggests is a pair of the fourth order Runge-Kutta with adaptive stepsizing such as Dormand-Prince (Dormand and Prince, 1980) or Runge-Kutta Fehlberg (Fehlberg, 1969) method and Radau IIA (Hairer and Wanner, 1996) methods. Radau IIA is the best implicit Runge-Kutta equation ever known, and is L-stable, and B-convergent of the order s for the consistency of $2s - 1$. Three-stage ($s = 3$) version of Radau IIA with the fifth order consistency, sometimes called Radau 5, is often used.

2.3.2 Special types of differential system solvers

In cellular and biochemical simulations, specialized differential equation solvers are sometimes used. For example, power-law canonical differential equations are often used to model various types of cellular phenomena such as gene expression and metabolic systems. Two of these formalisms are S-System:

$$\frac{d}{dt}x_i = \alpha_i \prod_{j=1}^{n} x_i^{g_{i,j}} - \beta_i \prod_{j=1}^{n} x_i^{h_{i,j}}, \quad i = 1, \cdots, N, \tag{2.50}$$

where N is the size of the system, and α_i, β_i, $g_{i,j}$, $h_{i,j}$ are S-system coefficients, and Generalized Mass Action (GMA):

$$\frac{d}{dt}x_i = \gamma_i \prod_{j=1}^{n} x_i^{f_{i,j}}, \quad i = 1, \cdots, N, \tag{2.51}$$

where γ_i and $f_{i,j}$ are GMA coefficients.

A distinct feature of this kind of formalisms is that it does not distinguish the structure of the system from its dynamics. The whole properties of a system can be described by a single S-System or GMA matrix. When used in simulation, it provides good means of estimating the network structure of the system as well as kinetic orders. In other words, by using S-System and GMA formalisms, the difficult problem of network structure determination can be converted to a matter of numerical parameter estimation, which is compatible with well established technique of non-linear optimization with numerous powerful algorithms such as Genetic Algorithms and the modified Powell method. A special algorithm called ESSYNS method can be used to solve these power-law systems efficiently (Irvine and Savageau, 1990).

2.3.3 Stochastic algorithms for mesoscopic chemical systems

All numerical treatments explained above are continuous and deterministic, which means that those description of chemical processes are macroscopic and approximate. Chemical systems are, at the bottom, composed of discrete molecules, and the assumption behind differential formalisms that state variables change continuously and their trajectories are differentiable does not hold, and thus, the differential descriptions fail to correctly reproduce the stochastic fluctuation in the number of molecules that becomes evident when copy numbers of involving molecular species of reactions are small.

Some variations of stochastic simulation algorithms for chemically reacting systems both in exact and approximate consequences of the chemical master equation are introduced here.

Chemical master equation

Consider a finite and fixed volume Ω in which M reaction channels connect N molecular species. Temperature and other physical parameters that affect the reaction rate are all constant. In meso-scopic representation of the system, we do not track motion of each molecule. Then the state of the system is a vector of random variables $\mathbf{X}(t) \in \mathbb{N}^N$. It also assumes that occurrences of nonreactive collisions are so frequent that (1) it stirs the system between any two reactive collisions, and (2) each occurrence of reaction is a Markov process. Strictly speaking, in cell biology, most systems are in liquid phase and, unlike gas phase, once two solute molecules encounter, it is highly probable that these molecules experience numerous successive collisions because of the existence of solvent, making reactions non-Markovian. However, here we neglect this in the following discussions because experiences so far have shown that we can construct precise simulation methods without taking this non-Markovian argument into account.

Now the *propensity function*

$$a_j(\mathbf{X})dt, \tag{2.52}$$

is defined as the probability that the j-th reaction will occur in Ω in a infinitesimal time interval $[t, t + dt)$, given $\mathbf{X} = \mathbf{X}(t)$. The *state change vector*

$$\boldsymbol{\nu}_j, \tag{2.53}$$

whose each element $\nu_{j,i}$ specifies the number of molecules of the i-th species changed by an occurrence of the j-th reaction.

If the reaction is elementary, the propensity function has the following form:

$$a_j(\mathbf{X}) = c_j \eta_j, \tag{2.54}$$

where $c_j dt$ gives the probability that a pair of reactant molecules will collide and react in a unit time interval, and η_j is the number of distinct combinations of such reactant molecules in the state $\mathbf{X}$. If the reaction has the form of (1) $X_1 \to X_2 \cdots$, $\eta_j = X_1$, if it is (2) $X_1 + X_2 \to X_3 \cdots$, then $\eta_j = X_1 X_2$, and if (3) $2X_1 \to \cdots$, then $\eta_j = \max(X_1(X_1 - 1)/2, 0)$. The macroscopic rate constant k_j of the reaction is related with c_j in: (1) $k_j = c_j$, (2) $k_j/\Omega = c_j$, (3) $2k_j/\Omega = c_j$.

Given the propensity function and the state change vector, an *exact* and *complete* description of the chemically reacting system evolving from the initial condition $\mathbf{X}_0$ at time t_0 can be derived:

$$\frac{\partial}{\partial t} P(\mathbf{X}, t|\mathbf{X}_0, t_0) = \sum_k^M \left[a_k(\mathbf{X} - \boldsymbol{\nu}_k) P(\mathbf{X} - \boldsymbol{\nu}_k|\mathbf{X}_0, t_0) - a_k(\mathbf{X}) P(\mathbf{X}|\mathbf{X}_0, t_0) \right]. \tag{2.55}$$

This is called the *chemical master equation* (CME). This formalism was initially proposed as a simple stochastization of macroscopic rate equations (McQuarrie, 1967), but has recently given a rigid micro-physical ground (Gillespie, 1992).

By definition of $\mathbf{X}$, (that is $\in \mathbb{N}^N$), CME 2.55 is a system of a huge number of differential equations, and cannot be solved in any way (analytical or numerical) unless it represents a very simple system (for example, a single ion channel whose state is either open or close).

Exact stochastic simulation algorithms

Although the CME describes everything about a chemical system, it cannot be used in simulations directly. What we need is a method of trajectory realization based on the CME, which does not require evaluation of the whole state space defined in the system. One way to reduce the computation cost to a point where we can handle with our digital computers is a kind of lazy evaluation, that is, at each simulation step, calculating the values of propensity functions at neighboring state points to the current state point.

Gillespie proposed the *next reaction density function* (Gillespie, 1976, 1992),

$$p(\tau, j|\mathbf{X}, t) = a_j(\mathbf{X}) \exp\left(-\tau \sum_{k}^{M} a_k(\mathbf{X})\right), \tag{2.56}$$

which defines the probability that the next reaction in the system occurs in the infinitesimal time interval $[t+\tau, t+\tau+d\tau)$, and the reaction is the j-th reaction.

With this next reaction density function, a step of simulation is defined as a procedure of generating a pair of numbers (τ, μ), that indicates the step size and the state transition difference function $X(t+\tau) = X(t) + \boldsymbol{\nu}_j$. There can be some possible ways of generating this real-integer pair of random numbers, and one of the straightforward ways is called the Direct method. The joint density function 2.56 can be rewritten as a composition of two simple functions for τ and j, respectively.

$$p(\tau, \mu|\mathbf{X}, t) = p_1(\tau|\mathbf{X}, t) \cdot p_2(\mu|\mathbf{X}, t), \tag{2.57}$$

where $p_1(\tau|\mathbf{X},t)$ is the probability that the first firing of *any* of the reaction channels occurs at time $t+\tau$, and $p_1(\mu|\mathbf{X},t)$ is the probability that μ is the reaction that fired. These two functions are defined as:

$$P(\tau|\mathbf{X},t) = \sum_k^M a_k(\mathbf{X}) \exp\left(-\tau \sum_k^M a_k(\mathbf{X})\right), \tag{2.58}$$

$$Pr(\mu|\mathbf{X},t) = a_\mu(\mathbf{X}) \Big/ \sum_k^M a_k(\mathbf{X}). \tag{2.59}$$

In numerical computer simulations, a type of random numbers that is available in the most efficient way is the uniform, unit-interval random distribution $\mathcal{U}(0,1)$. To generate τ according to 2.58 from a sample of the uniform distribution, u_1, the following Monte Carlo inversion function of 2.58 is used:

$$\tau = \left(\sum_k^M a_k(\mathbf{X})\right)^{-1} \cdot \ln\left(\frac{1}{u_1}\right). \tag{2.60}$$

Determining μ is then a simple task. Using another number u_2 taken from the uniform random number generator,

$$\mu = \min\{n : n \in \mathbb{N} \wedge \sum_k^n a_k(\mathbf{X}) > u_2 \sum_l a_l(\mathbf{X})\} \tag{2.61}$$

Main portion of the computation cost of the Direct method comes from two random number generations and calculation of M propensity functions per a simulation step.

Here is a procedural definition of the Gillespie's Direct method:

1. Initialize: set initial number of molecules, and reset t ($t \leftarrow 0$).
2. Calculate the values of the propensities (a_i) for all the reactions.
3. Choose τ according to 2.60.

4. Determine the next reaction μ according to 2.61.
5. Change the number of molecules: $\mathbf{X} \leftarrow \mathbf{X} + \boldsymbol{\nu}_\mu$.
6. $t \leftarrow t + \tau$.
7. Go to (2).

Another approach to the generation of the (τ, μ) pair is the *First Reaction method*, also devised by Gillespie(Gillespie, 1976). It generates tentative τs for all reaction channels,

$$\tau_l = \frac{1}{a_l(\mathbf{X})} \cdot \ln\left(\frac{1}{u}\right), \tag{2.62}$$

where u is a unit-interval uniform random number. Adopt the smallest τ_l as τ:

$$\tau = \min_l \tau_l. \tag{2.63}$$

μ is then the reaction which got the smallest τ_l:

$$\mu = l \quad \text{s.t.} \quad \tau_l = \tau. \tag{2.64}$$

This algorithm requires M random numbers and M calculations of the propensity functions, thus runs slower than the Direct method. However, it gives a basis for the recently proposed Next Reaction method, which we will discuss next.

The First Reaction method in a procedural appearance is like this:

1. Initialize: set initial numbers, and $t \leftarrow 0$.
2. Calculate the values of the propensities (a_i) for all the reaction channels.
3. For each of the reactions, calculate a putative time τ_i of the next occurrence of the reaction, according to the propensity calculated in (2).

4. Pick a reaction whose τ_μ is the least. $\tau \leftarrow \tau_\mu$.
5. Change the number of molecules: $\mathbf{X} \leftarrow \mathbf{X} + \boldsymbol{\nu}_\mu$.
6. $t \leftarrow t + \tau$.
7. Go to (2).

Twenty four years after Gillespie's original work (Gillespie, 1976), Gibson published a paper that proposes a vastly improved version of the First Reaction method, called the *Next Reaction method*, in 2000 (Gibson and Bruck, 2000). The core idea of the new method is to limit recalculation of τ_l to really necessary cases. In the First Reaction method, τ_l for each reaction is calculated in each simulation step. But with the Markovian assumption of the meso-scopic formalism, the recalculations are not necessary for reactions not affected by the current reaction μ. To take full advantage of this good opportunity of optimization, this method (1) uses absolute time, rather than relative time in the Direct and the First reaction methods, (2) dependencies between reactions are pre-calculated at simulation start-up, and (3) the absolute putative time τ_l of each reaction is stored in a dynamic priority queue data structure to speed up the operation of changing the value of τ_l.

The cost of this algorithm is just one random number generation and evaluations of the propensity functions affected by the current reaction. This is supposed to be near or perhaps on the theoretical limit as long as the simulation precisely tracks each occurrence of reaction events. This method has to, however, maintain the priority queue, of which cost is typically $O(\log_2(M))$ integer operations. Because the numbers of other operations such as the random number generation and propensity function evaluations grows proportionally to the density of the stoichiometry matrix, this logarithmic term can form a bottleneck in computation speed for large Ms. The point of equilibrium between costs of the priority queue and other operations is implementation and platform dependent. It is worth noting, however, that recent advancements in pseudo random number generation methods such as the Mersenne Twister algorithm(Matsumoto and Nishimura, 1998) and integration of multiple high-performance floating-point operation units into CPU chips have been lowering the equilibrium point of M. Vasudeva and Bhalla (2004) has some practical performance comparisons of these methods.

Below is the step-by-step instructions for the Next Reaction method:

1. Initialize: set initial numbers, $t \leftarrow 0$, and for each reaction i, calculate a putative time τ_i.
2. Pick the reaction with the least putative time τ_μ.
3. Change the number of molecules: $\mathbf{X} \leftarrow \mathbf{X} + \boldsymbol{\nu}_\mu$.
4. $t \leftarrow \tau_\mu$.
5. For each affected reaction α,
   (a) calculate new $a_{\alpha,new}$.
   (b) if $\alpha = \mu$, calculate new τ_α.
   (c) if $\alpha \neq \mu$,
   $$\tau_a \leftarrow \frac{a_{\alpha,old}}{a_{\alpha,new}}(\tau_\alpha - t) + t.$$
6. Go to 2.

The scaling operation in step 5(c) is an optimization to avoid using extra random numbers, which is effectively equivalent as reusing the random numbers generated in the previous step. Legitimacy of this random number reusing is discussed in detail in Gibson and Bruck (2000). Of course, generating a new random number here and doing the same as 5(b) yields the same result.

To summarize, as long as the number of reactions is sufficiently large, and the computation needs to be exact (each occurrence of a reaction must be counted), Gibson's Next Reaction method is the best known way of realizing simulation trajectories according to CME.

Approximate stochastic simulation algorithms

Stochastic simulation methods can be approximate. For *macroscopically infinitesimal* time interval τ that does not change values of the propensity functions appreciably, the number of firings of j-th reaction K_j in the time interval $[t, t+\tau)$ can be approximated that

$$K_j(\tau; \mathbf{X}(t), t) \approx \mathcal{P}(a_j(\mathbf{X}(t)), \tau), \tag{2.65}$$

where $\mathcal{P}$ is the *Poisson random variable.* The net change $\mathbf{L}(\tau; \mathbf{X}(t))$ of the system state approximated from $\mathbf{X}(t)$ after τ is

$$\mathbf{L}(\tau; \mathbf{X}(t)) = \sum_j K_j(\tau; \mathbf{X}(t)) \boldsymbol{\nu}_j. \tag{2.66}$$

A simulation step is then defined as

$$\mathbf{X}(t+\tau) = \mathbf{X}(t) + \mathbf{L}(\tau; \mathbf{X}(t)). \tag{2.67}$$

This procedure is called *tau leaping* (Gillespie, 2001). To ensure the τ to be macroscopically infinitesimal, it must satisfy the following inequality:

$$\forall j : \left| a_j(\mathbf{X}(t) + \mathbf{L}(\tau; \mathbf{X}(t))) - a_j(\mathbf{X}(t)) \right| \leq \varepsilon \cdot \sum_k a_k(\mathbf{X}(t)), \tag{2.68}$$

where $0 < \varepsilon \ll 1$ is the error control parameter. An alternative for the right hand side of this condition is proposed to be $\varepsilon \cdot \max_k a_k(\mathbf{X}(t))$. Either way yields essentially the same result as ε is an artifact.

There can be many possible ways of finding the largest possible τ that meets the condition 2.68, and one of the best known procedure is given in Gillespie and Petzold (2003). Denote the estimated difference of the propensity function in the left hand side of 2.68 as $\Delta a_j(\tau, \mathbf{X})$,

$$\Delta a_j(\tau, \mathbf{X}(t)) \equiv a_j(\mathbf{X}(t) + \mathbf{L}(\tau; \mathbf{X}(t))) - a_j(\mathbf{X}(t)) \tag{2.69}$$

It is a random variable, and 2.68 can be reformulates as

$$\forall j : \left|\langle \Delta a_j(\tau, \mathbf{X}(t)) \rangle\right| + \mathrm{sdev}(\Delta a_j(\tau, \mathbf{X}(t))) \lessapprox \varepsilon \cdot \sum_k a_k(\mathbf{X}(t)), \tag{2.70}$$

where $\langle \cdot \rangle$ is the mean and sdev$(\cdot)$ is the standard deviation of the random variable. This bounds about two thirds of the samples of $\Delta a_j(\tau, \mathbf{X}(t))$ to be less than the right hand side. Because of the arbitrariness in the specification of ε, this is assumed to work just fine in simulations. To calculate these numbers efficiently, take a first order Taylor expansion of 2.69.

$$\begin{aligned} \Delta a_j(\tau, \mathbf{X}(t)) &\approx \mathbf{L}(\tau; \mathbf{X}(t))) \cdot \boldsymbol{\nabla} a_j(\mathbf{X}(t)) \\ &= \sum_i L_i(\tau; \mathbf{X}(t)) \frac{\partial a_j(\mathbf{X}(t))}{\partial X_i}. \end{aligned} \tag{2.71}$$

When only elementary reactions are considered, this partial differentiation should easily be calculated. Recalling 2.65 and 2.66, this becomes

$$\Delta a_j(\tau, \mathbf{X}(t)) \approx \sum_j \sum_i \frac{\partial a_j(\mathbf{X}(t))}{\partial X_i} \nu_{j,i} \cdot \mathcal{P}_j(a_j(\mathbf{X}(t)), \tau). \tag{2.72}$$

Setting

$$h_{i,j}(\mathbf{X}(t)) \equiv \sum_i \frac{\partial a_j(\mathbf{X}(t))}{\partial X_i} \nu_{j,i}, \tag{2.73}$$

and using the fact that $\langle \mathcal{P}(a,b) \rangle = \mathrm{var}\left(\mathcal{P}(a,b)\right) = ab$,

$$\langle \Delta a_j(\tau, \mathbf{X}(t)) \rangle \approx \sum_j h_{i,j}(\mathbf{X}(t)) a_j(\mathbf{X}(t))\tau, \tag{2.74}$$

$$\mathrm{var}\left(\Delta a_j(\tau, \mathbf{X}(t))\right) \approx \sum_j \left(h_{i,j}(\mathbf{X}(t))\right)^2 a_j(\mathbf{X}(t))\tau, \tag{2.75}$$

hence

$$\mathrm{sdev}\left(\Delta a_j(\tau, \mathbf{X}(t))\right) \approx \sqrt{\sum_j \left(h_{i,j}(\mathbf{X}(t))\right)^2 a_j(\mathbf{X}(t))\tau}. \tag{2.76}$$

By letting 2.74 and 2.76 into 2.70, and solving for τ gives a procedure of determining the leaping interval. However, the square root over τ in 2.76 makes the solution slightly costly to be evaluate in every step of the simulation. An essentially equivalent way, again owing to the arbitrariness of the value of ε, to this that Gillespie suggests is to ensure each of 2.74 and 2.76 separately bounded by the limit.

$$\forall j : \left| \langle \Delta a_j(\tau, \mathbf{X}(t)) \rangle \right| \leq \sum_k a_k(\mathbf{X}(t)) \quad \wedge \quad \mathrm{sdev}(\Delta a_j(\tau, \mathbf{X}(t))) \leq \sum_k a_k(\mathbf{X}(t)). \tag{2.77}$$

Finally we get the formula to find the largest possible τ that does not violate 2.68:

$$\tau = \min_j \left(\min \left(\frac{\varepsilon \sum_k a_k(\mathbf{X}(t))}{\sum_j h_{i,j}(\mathbf{X}(t)) a_j(\mathbf{X}(t))}, \frac{\left(\varepsilon \sum_k a_k(\mathbf{X}(t))\right)^2}{\sum_j \left(h_{i,j}(\mathbf{X}(t))\right)^2 a_j(\mathbf{X}(t))} \right) \right). \tag{2.78}$$

When τ is near or less than $(\sum_k a_k(\mathbf{X}(t)))^{-1}$, which is the expected mean of the step size taken by the exact methods, the tau leaping method loses the performance advantage, and exact methods should be used for the step.

The system of propensity functions can be stiff in the same sense as in deterministic ODE systems, and the implicit scheme, which is exactly the same strategy to fight it, can be used. Rathinam *et al.,* (2003) implicitized the propensity function part of the tau leaping equation 2.67, leaving the random variable part explicit.

$$\mathbf{X}(t+\tau) = \mathbf{X}(t) + \sum_j \boldsymbol{\nu} a_j(\mathbf{X}(t+\tau))\tau + \sum_j \boldsymbol{\nu} \left(\mathcal{P}(a_j(\mathbf{X}(t),\tau) - a_j(\mathbf{X}(t)))\right). \tag{2.79}$$

Here the equation is divided into the deterministic mean part and the stochastic fluctuation, and the state of the system at the next time point $t+\tau$, which is unknown, appears only in the deterministic part. In numerical implementations, the third term of 2.79, which includes the Poisson random variables, are calculated first using samples of the random variables, and then the value of the second term is determined by using the Newton iteration as in essentially the same way we used to solved the implicit Euler equation in 2.49. This method 2.79 is called the *implicit tau leap method.*

When all the reaction channels fire a very large number of times in $[t, t+\tau)$, *and* the leap time τ is still small enough that it can be assumed that the propensities of all the reactions does not change their values so much, the following simplification to the tau-leaping can be made: If the mean of the Poisson random variable $\mathcal{P}(a_j(\mathbf{X}(t),\tau))$, which is $a_j(\mathbf{X}(t))\tau$ is very large, it can be approximated by normal random variables. Therefore 2.65 becomes

$$\begin{aligned} K_j(\tau;\mathbf{X}(t),t) &\approx \mathcal{P}(a_j(\mathbf{X}(t)),\tau) && (2.80)\\ &\approx \mathcal{N}(a_j(\mathbf{X}(t))\tau, a_j(\mathbf{X}(t))\tau) && (2.81)\\ &= a_j(\mathbf{X}(t))\tau + \sqrt{a_j(\mathbf{X}(t))\tau}\cdot\mathcal{N}(0,1). && (2.82)\end{aligned}$$

Using this consequence and 2.66, the tau-leaping method 2.67 becomes

$$\mathbf{X}(t+\tau) = \mathbf{X}(t) + \sum_j \left(a_j(\mathbf{X}(t))\tau + \sqrt{a_j(\mathbf{X}(t))\tau} \cdot \mathcal{N}(0,1) \right) \boldsymbol{\nu}_j \tag{2.83}$$

$$= \mathbf{X}(t) + \sum_j \boldsymbol{\nu}_j a_j(\mathbf{X}(t))\tau + \sum_j \boldsymbol{\nu}_j \sqrt{a_j(\mathbf{X}(t))\tau} \cdot \mathcal{N}(0,1). \tag{2.84}$$

This is called the *Langevin method*, because if τ is replaced with dt it is identical to the *Chemical Langevin equation* (CLE).

$$\mathbf{X}(t+dt) = \mathbf{X}(t) + \sum_j \boldsymbol{\nu}_j a_j(\mathbf{X}(t))dt + \sum_j \boldsymbol{\nu}_j \sqrt{a_j(\mathbf{X}(t))} \mathcal{N}(0,1)\sqrt{dt}. \tag{2.85}$$

Unlike the exact methods and the tau leaping methods, state variables in the Langevin method are *continuous*.

Consider a *thermodynamical limit*, where we assume the limit of infinite volume without changing concentrations of the species. Then the number of reaction events occurring in the time duration τ in the infinite volume becomes infinite. In this case, the third term of the right-hand side of the Langevin equation 2.84 vanishes.

$$\mathbf{X}(t+\tau) = \mathbf{X}(t) + \sum_j a_j(\mathbf{X}(t))\tau\boldsymbol{\nu}_j \tag{2.86}$$

This is essentially the explicit Euler method for ODEs. From this observation, it can be concluded that the exact methods, tau leaping and Langevin methods are a family of simulation techniques that spans multiple scales. In the limit where the system is described with statistics rather than individual reaction events, the approximation of using deterministic differential equations is justified.

2.3.4 Cellular automata

Cellular automata (CA) is a structure:

$$C = \langle L, S, N, F \rangle, \tag{2.87}$$

where L is an *infinite* regular lattice of cells, S is a *finite* set of states , N is a set defining neighbors, and F is a state transition function. F has the following form:

$$F : \{s_j\}_{j \in N(i)} \to s \tag{2.88}$$

where $s_j \in S$ is the state of the jth cell, and $i \in L$ is the current cell. This function can be stochastic.

The origin of CA traces back to John von Neuman's work in theoretical computer science. Subsequently, it became a major way of investigating qualitative simulations of complex phenomena in diffusion-reaction systems, such as for excitable media and Turing patterns. Recently it is becoming popular as a means of quantitative modeling and simulation of physical phenomena such as fluid dynamics and diffusion-reaction(Dawson *et al.*,, 1993). Related to computational cell biology is a type of CA for enzymatic reaction networks with spatial extent represented by molecular diffusion proposed in Weimar (2002). Patel *et al.,* (2001) has an application of CA to the model of tumor growth. Ermentrout and Edelstein-Keshet (1993) has some more references.

2.3.5 Other methods and summary

We reviewed some simulation methods in computational cell biology, and there are several more schemes those are already in popular use or still under development. An instance of a recent development is a hybrid dynamic/static pathway simulation method that combines conventional ODE-based rate equations with a static, flux-based sub-components (Yugi *et al.,*, 2002). The static part is based on a time-less metabolic flux analysis (MFA), and its (re-)calculation is triggered by simulation steps of the dynamic, ODE sub-components. Some more examples are Monte-Carlo Brownian dynamics method originally developed for simulation of micro-scopic simulation of synaptic transmission (Bartol *et al.,*, 1996), boolean network, coupled map lattice, and difference equations.

This diversity in computational approaches is a natural consequence of the heterogeneity and multiple scales of the cell as a dynamical system, and therefore is an obvious reason of the need for the multi-formalism modeling and multi-algorithm simulation discussed in the next section.

2.4 Requirements analysis

In this section, the necessary features of what we consider to be a flexible and multi-purpose biosimulation software platform are identified and described. Critical features common to the vast majority of modern platforms, such as model scalability and efficiency, are ignored, whereas those not present in all software form the primary focus. Whilst we recognize that these requirements may vary according to many factors, including simulation granularity, scale, and purpose, we hope to establish a generic description of a *cellular* platform.

The following is the list of seven specific requirements that we argue to be important for design and implementation of cell simulation software.

1. Multi-algorithm simulation
2. Multi-timescale simulation

3. Object-oriented modeling
4. Object-oriented simulation
5. Runtime user interaction
6. Variable model structure
7. Spatial modeling and simulation

In the following sections, we give a definition, rationale, and some typical or possible solutions to each of these requirement.

2.4.1 Multi-algorithm simulation

In the cell, as we have seen in the previous sections, various components with different properties interact in diverse manners. All cellular subsystems are highly non-linear, and couplings of the subsystems are often non-linear as well. The nonlinearity indicates that the whole system is not equivalent to the sum of the subsystems. Although a subsystem in isolation can be investigated to some extent by assuming steady and simplified boundary conditions, the real behavior and role of the subsystem cannot be elucidated unless it is considered as part of the whole.

Cell simulators must therefore allow simulation of cell subsystems in both isolated and coupled forms. Simulation of coupled subsystems requires performing computations on mutually interacting subsystems with different computational properties on a single platform. There is, however, no universal algorithm which can efficiently conduct simulation of all the subsystems at once, and so simulators must allow multiple computation algorithms to coexist in a single model.

In order to support multi-algorithm computation, the software must therefore provide a single abstract programming interface, which (1) allows indiscriminate interaction among the modules, and (2) gives frontend programs a standard means of visualizing and manipulating said modules. This also means that the implementations of the algorithm modules must be isolated from the system-provided common interface.

Related to this necessity of multi-algorithm simulator is representation schemes of the target physical entity (the cell) in computer. The primary state variables of a cell model are the quantities of species, and these can be modeled using two different approaches. In the first, each state variable is a positive real number, and the fractional parts are not discarded; this number format is suitable for working with the empirical rate equations of biochemistry. The second approach keeps molecular quantities as natural numbers, and a variety of methods may be used to maintain the semantics of the fractional part, and these can be either stochastic or deterministic. In addition, realistic cell models usually require a mixture of continuous state variables like temperature, electric potential, and free energy, and discrete variables like the state flags of multi state molecules (*e.g.* transiently modified proteins). Therefore, the simulator needs to handle both types of positive, non-zero molecular quantities, real negative or positive numbers, and discrete states of molecules.

2.4.2 Multi-timescale simulation

Cell simulators must be able to handle the coexistence of a wide range of time scales in a single cell model. Typical time scales of cellular phenomena vary from femtoseconds ($10^{-15}s$) to hours ($10^{3}s$). One example of the relatively high resolution necessary for realistic simulation of cellular behavior is found in signal transduction systems dependent on molecular bindings, these need step sizes on the order of ($10^{-6}s$). Metabolic pathways, on the other hand, are characterized by enzymatic reactions, and have close to milli-second ($10^{-3}s$) time scales. Gene expression systems are microscopically driven by macromolecular interactions (below $10^{-6}s$), yet resulting phenomena are often reported and analyzed in minutes, hours, and days ($10^1s - 10^4s$).

In differential equation based simulators, such 'stiffness' is usually overcome by adopting implicit 'stiffness-proof' integration algorithms. Although these techniques can be used in part for ODE based subsystems, they are not fully applicable to multi-algorithm simulators. Implementation of a multi-timescale and multi-algorithm simulator requires a new scheduling algorithm which can orchestrate many subsystems running with different algorithms and variable temporal step sizes.

2.4.3 Object-oriented modeling

Cell modeling involves the integration of a diverse array of processes. As outlined in Table 2.2, this is a very different situation from typical modeling problems in physics where computer simulation has been widely applied for quite some time. Fluid mechanics, gravitational N-body problems, and quantum field simulations are characterized by numerous uniform components (atoms, particles) that are governed by a single or a few equations (*e.g.* Navier-Stokes equation, Newton's equation, quantum wave functions). Preparation of structural information and initial conditions for such simulation models is relatively easy. If the components are uniform, one can assume that the values of some or most constants and parameters are common among the components. If there are only a few interaction modes, one does not need to carry out surveys and derive an equation for each of the reactions. Initial conditions are often automatically generated. The major challenges in these problems for software designers are maximizing computational efficiency in existing hardware and making the simulation software flexible without sacrificing the computational efficiency.

In cell simulation and modeling, one must either dig through the literature or conduct wet experiments to identify reaction mechanisms, derive appropriate rate equations, and obtain values of rate constants. This phase of research, review, data acquisition, and model construction often takes more than 50% of the man hours available to the project. Of the remainder, most are spent in model debugging. A proportionately greater amount of time is distributed amongst these activities as the model gets larger, and intuition and productivity are more important than computational efficiency. Most simulation engines will have similar runtime efficiencies anyways if the same kind of algorithms are used, and the greatest performance increase can be achieved in the pre-simulation phases. This means that

1. the modeling frontend should directly reflect the biologist's understanding of the cellular system, and,
2. the constructed models should be modularized for reuse in whole or in part as sub-models in a different context.

Ontological diversity is inherent to biology. Although it is somewhat obvious that there is no universal modeling scheme which can be used to model all subsystems elegantly and intuitively, cell simulators must attempt to provide one, or biological models will never be able to meet the staggering complexity of life. Biology is a science of exceptions, and new, non-intuitive discoveries are made all the time — as an example (amongst millions), we ask how one could predict with any confidence that non-coding complementary RNA might inhibit translation of the coding mRNA. Such considerations lead the software designer to provide a cell modeling framework that comes packaged with as few presumptions as possible.

The object-oriented paradigm provides an appropriate solution. This programming model is characterized by a data-centric framework in which procedures are tied to the data. The actors of a system are described as 'objects', which have data and methods (operations on this data) as their 'properties', and the objects are grouped into 'classes', interacting with each other by exchanging 'messages'. Inheritance (is-a), aggregation (has-a, consists-of), and dependence (uses-a) relationships may exist between classes. These relationships can be described in the Unified Modeling Language (UML), and coded with object-oriented computer languages such as C++, Java, Smalltalk and SIMULA. The object-oriented paradigm was in fact originally devised as a means of programming discrete simulations. Also relevant to the biosimulation problem is the fact that many object-oriented data structures are known to have strong affinity with databases, owing to their data-centric nature. It is also important to note that owing to its widespread 'world domination', effectively everyone with a software engineering background is supposed to be familiar with this paradigm, and also it is very easy to get educational resources.

The object model of cell simulators needs to be developed so that:

1. Users are able to define their own classes.

   As suggested above by the RNA example, a cell model might require new data structures currently not provided by the simulation engine.

2. Abstraction support (*e.g.* via base classes) is provided for common use by any algorithm.

   This allows interactions between different algorithms and subsystems, and across scales of time and space, critical features of an integrated simulation model.

2.4.4 Object-oriented simulation

Traditionally, simulation models of dynamic systems are either described in equation based form and then compiled into computer programs, described in an object-oriented fashion before translation to an equation based form, or written directly as computer programs. The second of these appears to provide the best of both worlds: clean design in the modeling phase and efficient numerical routines in the simulation phase.

Although computationally efficient, all three of these approaches have several inherent disadvantages:

1. The semantics of the model are not preserved.

   Modeling knowledge about structural and/or behavioral features of a system are abstracted out, and only implicitly represented in the mathematical description.

2. Model and program reuse is limited.

   If the model and its internal representation do not directly reflect each other, changes in one side require a change on the other, making it difficult to reuse models in modified contexts.

3. The classical flat description hides critical features and assumptions, such as the order of computation and the causal structure of a model.

Hitz and Werthner (1997) showed how object-orientation can help to overcome these drawbacks in dynamic system simulation, and how the paradigm, which is well established in discrete event-driven simulations, can even be extended to dynamic system simulations where differential or difference equations are normally used.

Another advantage of using object-oriented data structure inside the simulator is a one-to-one relationship between the model and simulation data structures. Saving, loading, and resuming simulation sessions can be made easy by implementing support for object persistence technology. Keeping the simulation in the same semantic plane as the model helps users to understand the ontologically complicated system via simulation results and runtime interaction.

2.4.5 Runtime user interaction

One purpose of a simulation project is to characterize the dynamic behavior of a target system. Mathematical analysis methods such as metabolic control analysis, bifurcation analysis, and parameter estimation are usually automated and user interaction is not necessary.

However, as the model gets larger and more complicated, the advantages of runtime interaction with the simulation engine become significant. Even for prototype models in early stages of development, modelers would like to observe the effects of stimulations at arbitrary time points. In the debug phase of a modeling project, runtime interactions constitute a powerful means to check for the existence and location of 'bugs' in the model. Debugging support is often neglected in cell simulation software, but experience has shown that it can eliminate some severe bottlenecks, especially in large-scale, multi-author projects. After completion of the model, dynamic properties can be investigated intuitively through runtime interactions, and the insight gained is educational to say the least; indeed, interaction with a model in such a fashion could help bring cell simulation to the classroom.

It goes without saying that the same interaction functionality is required for predefined scripts that dynamically affect the variables and structure of the model, and so we argue that simulators need to provide users with both batch-mode and runtime graphical interaction mode user interfaces in a seamless manner.

2.4.6 Dynamic structure model

Cells are dynamic environments in which the components are constantly being synthesized, processed, and degraded. Structural properties of the system are subject to change throughout the entire cell cycle, a clear example being the events associated with cell growth and division. With the differential equation based approach, for example, representation of structural change and the dynamic synthesis and degradation of components requires convoluted mathematical techniques. Even if the model is constructed in an object-oriented fashion in the modeling phase, it becomes extremely difficult to explore the dynamic nature of the system during simulation if using a compiled representation.

Object orientation, however, does not immediately mean that it can easily support the dynamic structures. However, a combination of object-orientation and discrete-event simulation formalism may provide a useful foundation of intuitive representation and efficient implementation of the simulator that supports dynamically changing model structures.

2.4.7 Spatial modeling and simulation

Even within a single compartment of a cell, there exist many systems where material distributions are not uniform in space and the localization of molecules plays an important role. Examples might include vesicle trafficking in the secretory process, diffusion of ions and molecules after crossing a membrane, and propagation of an action potential along the axon of a nerve fiber.

In whole or multi cell scale modeling, the assumption that chemical concentrations are homogeneous throughout a given compartment is reasonable if the reactions are confined to a small region of space, and when the diffusion rate is fast compared to the rate of localized reaction. However, if neither of these conditions is met, which will often be the case for whole or multi cell-scale modeling, the spatial information support becomes essential. Thus, simulation engines require native support for factors such as the structure and location of compartments, spatial concentration gradients for all species, and interactions between moving molecules.

One solution for incorporating spatial information into the simulation of cellular processes is to use a finite element mesh to divide the sub-cellular space, and describe the location of each element with some coordinate system. Any spatial support as such requires an interface shared by all algorithm modules. The syntax of geometric coordinates would be defined by a common framework, whereas the semantics of the coordinates would be defined by each computation module. The interface needs to describe not only molecular concentrations for systems with a large number of molecules, but also individual molecules for systems with a small number of particles, and for purely structural phenomena.

If multiple computation modules with different simulation algorithms and spatial/time step sizes are used to allow diverse simulation schemes with a wide range of temporal and spatial scales to coexist, spatial information support may become extremely complicated. The spatial interface therefore not only needs to be flexible enough to allow support for every module, but also needs to be simple enough to prevent confusion. Enabling multiple pluggable modules with contrasting semantics, dimensions, and coordinates to share a common syntax or description method via common framework will allow both requirements to be met. Thus, a generic, flexible, and simple spatial information interface is a key feature of next generation simulation engines.

2.5 Existing software platforms and projects

Cell simulation is a relatively new area of computational science, but attempts to numerically simulate biochemical pathways has a considerable history. This section briefly reviews available software related to this work.

2.5.1 General purpose simulation software

One of the oldest publicly available software packages specialized for rate equation based numerical simulation of cellular processes is KINSIM (Barshop and Frieden, 1983). It is a DOS application which is still actively used by biochemists for kinetic analysis of enzyme reaction systems. GEPASI (Mendes, 1993) is also a rate equation based simulator with an integrated and easy to use interactive Windows GUI, and is widely used by biochemists for both research and education. GEPASI's successor, COPASI, is being developed with a focus on large scalability and distributed parallel computing. DBSolve (Goryanin *et al.*,, 1999) is another ODE-based simulator. ProMoT (Ginkel *et al.*,, 2003) is a Lisp based object-oriented modeling environment that uses the DIVA numerics solver as a simulation backend. A commercial tool for block-oriented generic simulation of complex systems, SCoP (Kootsey *et al.*,, 1986) is used to run differential and difference equation based cell models. A-Cell (Ichikawa, 2001) is a GUI-based piece of software used to construct biochemical reactions and electro-physiological models of neurons and other types of cell. Bio/Spice (McAdams and Arkin, 1998) was initially intended for genetic circuit simulation, and is now being developed as a generic modeling and simulation environment linked to object-relational databases. Jarnac, or SCAMP II, is a successor of the SCAMP simulator (Sauro, 1993), and equipped with a powerful and flexible scripting language that enables users to program a dynamic object-oriented cell model. Virtual Cell (Loew and Schaff, 2001) provides an intuitive WWW Java applet based modeling environment which allows modelers to construct spatial and biochemical models in both biological and mathematical semantic planes, and has support for empirical 3D data from microscopy. PLAS is a simple yet powerful tool for modeling, simulation, and analysis of S-Systems. And still many groups use a generic modeling package such as Mathematica or MatLab, although these tools are not customized to support biosimulation *per se.*

2.5.2 Specialized simulation software

StochSim (Le Novere and Shimizu, 2001) is a stochastic biochemical simulator in which individual molecules and molecular complexes are represented as individual software objects. Its unique algorithm makes for effective simulation of biochemical systems like the bacterial chemotaxis signaling pathway, where only a small number of molecules are involved and some of them are multi-state. MCell(Bartol *et al.,*, 1996) is another simulator in which individual molecules are treated not statistically, but individually, with a Monte Carlo type random-walk algorithm for Brownian dynamics. MCell is designed for the simulation of interactions between ligands and binding sites on receptors, enzymes, and transporters (amongst other molecules). Both simulators, StochSim and MCell, are somewhat infrequently used for simulation of sub-cellular dynamics because of high computational costs, yet the algorithms employed are likely to become indispensable if given appropriate roles in a multi-algorithm whole cell model.

2.5.3 Software platforms and languages

The Caltech ERATO Kitano Systems Biology Project is developing an XML-based Systems Biology Markup Language (SBML) as a means of interchanging biosimulation models (Hucka *et al.,*, 2003). The group has set out to encompass all the features of Bio/Spice, DBSolve, E-Cell, Gepasi, Jarnac, StochSim, and Virtual Cell. A similar modeling language being developed by the University of Auckland and Physiome Sciences is CellML (Headley and Nelson, 2001). The Systems Biology Workbench (SBW) is also being developed by the Caltech ERATO team(Hucka *et al.,*, 2002). SBW is a distributed object computing environment for biological modeling and simulation. It provides an infrastructure that unifies various software components like model editors, simulators and data analyzer/visualizers. Using SBML as its language, it may well become the standard platform of model exchange, data exchange, and inter-operation.

2.6 Conclusion

A new computational approach is needed for simulation at the whole cell scale, and sophisticated software engineering plays a vital role in cell simulation projects. Although simulation of biochemical processes has a considerable history, the approach of integrating partial simulation models into a whole or multiple cell model is still an emerging field.

Let us see the seven requirements of cell simulator again;

1. Multi-algorithm simulation
2. Multi-timescale simulation
3. Object-oriented modeling
4. Object-oriented simulation
5. Runtime user interaction
6. Dynamic model structure
7. Spatial modeling and simulation

This list is a reorganized version of the earlier version that has first been given in a previous work Takahashi *et al.,* (2002).

The requirements 1 (multi-algorithm) and 2 (multi-timescale) are tackled in chapter 3 by developing a new computational framework called the 'meta-algorithm', which can drive multiple sub-model components running on different simulation algorithms and different step-sizes. This new framework has a good affinity with object-orientation, and in chapter 4 we discusses how E-Cell System presents a solution to the requirement 3 (OO modeling). In the chapter we will also see that the design of the software allows frequent and real-time interactions between the simulator core and frontend software, and the requirement 5 (runtime user interaction) is met. Chapter 5 has some discussions on how the computational framework introduced in Chapter 3 and the software system described in Chapter 4 can be further extended to give good solutions to the remaining two items, the requirement 6 (dynamic model structure) and 7 (spatial modeling and simulation).

Chapter 3

Multi-algorithm simulation framework – the meta-algorithm

In this chapter, we define a computational framework that can be used as a platform of integrating various cellular submodels. First, we review some design options to be considered for that framework, and find a prior art in discrete event simulation. We will then define the framework named the *meta-algorithm* using a discrete event scheduler and Hermite polynomial interpolation. Some uses of this 'meta-algorithm' for biological and artificial models will be shown with an assessment of accuracy and analysis of performance gain. The meta-algorithm framework has first published in Takahashi *et al.,* (2004).

3.1 Introduction

3.1.1 Background

Computational cell biology is a rapidly growing simulation-oriented research field that has been greatly stimulated by the array of high throughput methods developed in recent years. In a previous chapter, we have argued that cell simulation poses many significant computational challenges that are distinct from those encountered in problems of other disciplines such as molecular dynamics, or computational physics, and even conventional biochemical simulations (See also Takahashi *et al.,* (2002)). A vast array of molecular processes occur simultaneously in the cell. The involved physical and chemical processes include molecular diffusion, molecular binding, enzymatic catalysis and higher order phenomena such as cytoplasmic streaming, complex macromolecular interactions (such as the dynamics of RNA polymerase on the DNA molecule), and global structural changes caused by cell division and cell differentiation.

Many different kinds of simulation algorithms have been proposed and are currently being used for simulation of cellular processes. Different algorithms have different strengths, and are often suited to different spatial and temporal scales. What we need here to reconstruct the whole cell on computers is a computational framework that can integrate those various simulation algorithms.

3.1.2 Classification of simulation algorithms

Time-driven simulation algorithms can be classified into three categories: differential equation solvers, discrete time systems, and discrete event systems (Figure 3.1). Those formalisms can be combined by or embedded in a "discrete event world view" (Zeigler *et al.*,, 2000). Although this work is not a direct derivation of this rigid discrete event

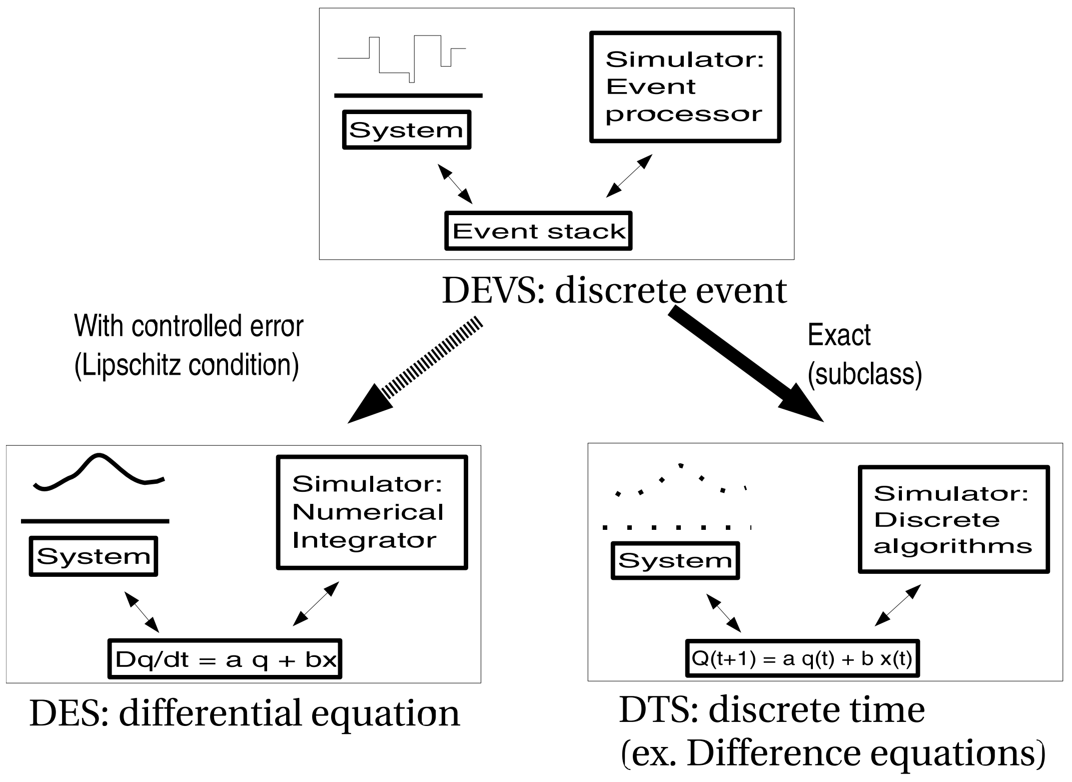

Simulator: Event processor
System
Event stack
DEVS: discrete event
With controlled error (Lipschitz condition)
Exact (subclass)
Simulator: Numerical Integrator
System
Dq/dt = a q + bx
DES: differential equation
Simulator: Discrete algorithms
System
Q(t+1) = a q(t) + b x(t)
DTS: discrete time (ex. Difference equations)

Figure 3.1: Multi-formalism simulation: a view from Zeigler's DEVS

theory of modeling and simulation, some fundamental concepts such as the classification of simulation algorithms and the use of an event scheduler have been borrowed from that framework.

3.1.3 Multi-algorithm framework: design options

Vangheluwe (2000) proposed three different modeling approaches to couple sub-models expressed in different formalisms.

- Meta-formalism approach.

  The different formalisms of the sub-models can be subsumed and described in a single meta-formalism. An example of this meta-formalism is Zeigler's DEVS&DESS

(Zeigler *et al.,*, 2000), which integrates discrete DEVS and continuous differential equation systems. (Note that, although the names are somewhat similar, the 'meta-algorithm approach' described in the following sections is not classified into this category.)

- Formalism Transformation approach.

  Vangheluwe proposed a Formalism Transformation Graph (FTG), in which each formalism is shown as a verticle, and possibilities of formalism transformations are denoted as directed edges.

  If each of the submodels is transformed to one common formalism, the resulting model is a single-formalism model, and it can easily be manipulated and simulated.

- Co-simulation approach.

  In this approach, each of the sub-models is simulated with a simulator specific to its formalism, and interaction due to coupling is resolved at the level of trajectory. This approach forms a basis of the DoD High Level Architecture (HLA) (Zeigler *et al.,*, 1997), which is a framework for simulator interoperability.

Vangeluwe argued that the Transformation approach makes most fruitful results for *modeling* of complex systems, because, (1) meaningful meta-formalisms which truly add expressiveness as well as reduce complexity are rare, and (2) compared to transformation to a common formalism before simulation, though its object-orientation is appealing from a software engineering viewpoint, co-simulation approach discards a lot of useful information about the model structure (questions can only be answered at the trajectory level).

However, when design of *simulation* software is under consideration, as Vangeluwe pointed out, the object-orientation, thus modularity, of the co-simulation approach becomes rather promising. Especially, under the light of the requirement 4 (Object-oriented simulation), one must be careful about adopting the meta-formalism and transformation approaches, because as we have seen in the previous chapter, these ways of multi-formalism simulation can harm the preservation of model semantics between modeling and simulation phases.

Zeigler also gives two ways of multi-formalism modeling in his book (Zeigler *et al.,* (2000), chapter 9):

- Formalism combining.

  This approach aims to bind strengths of existing simulation algorithms to produce a unified simulation algorithm of wide utility.

  This roughly corresponds to the 'meta-formalism' approach above, thus an example of this approach is also Zeigler's DEVS&DESS.

- Formalism embedding.

  In this approach, existing algorithms are implemented as pluggable modules wrapped to meet a common interface specification of a generic framework (*e.g.* DEVS).

  This corresponds to the 'co-simulation' approach above.

In this paper we adopt this Zeigler's terminology (combining / embedding) because the aim of our discussion is mainly about simulation, and we do not have a strong need of considering Vangeluwe's formalism transformation approach, which relies mainly on pre-simulation procedures.

Now the way we should go is becoming obvious. In this chapter, we explore the latter embedding approach, and develops an algorithmic framework for integrating heterogeneous units of a composite model. We call this framework the 'meta-algorithm', and it specifies the common interface of implementing (or wrapping) time advance and inter-module communication functions over existing simulation algorithms.

A virtue of this style of scheme integration is that there is orthogonality between subsystems, that is to say a degree of modularity can be assumed for each subsystem simulated using a particular algorithm. If this requirement is satisfied, one can expect established and well-studied algorithms, and the numerous existing models that exploit them, to be useful when combined in a "plug-in" fashion to let them take part in composite simulations that utilize the multiple algorithms.

3.2 Algorithm

As noted above, the novel approach we propose depends on the design of a meta-algorithm, which we describe here in three parts: (1) data structure, which outlines the organization of information required for model definition and execution, (2) driver algorithm, which describes how interactions between sub-modules are handled, and (3) integration algorithm, which explains the procedure by which state variables are updated.

3.2.1 Data structure

In the followings, a vector is denoted with a small bold letter (*e.g.* $\mathbf{x}$). Capital letters (*e.g.* X) are used for sets. Class names are capitalized (*e.g.* Stepper). Small and capital letters are used for scalars and objects.

Model is a class:

$$M = \langle \mathbf{x}, S \rangle \tag{3.1}$$

where

$$\mathbf{x} = \{x_0, x_1, \dots, x_n\} \tag{3.2}$$

(typically $\in \mathbb{R}^n$) is a vector of state variables, and S is a set of *Steppers.*

A Stepper is a class which represents a computational sub-unit of the model. The Stepper class is defined as:

$$S_i = \langle P_i, \tau_i, \Delta\tau_i, \Lambda_i, I_i \rangle \tag{3.3}$$

where $P_i = \{P_{i,0}, P_{i,1}, \dots P_{i,m}\}$ is a set of *Processes*, τ_i is the current local time, $\Delta\tau_i$ is the current time step interval, Λ_i is the step method, and I_i is the interruption method, of the ith Stepper. Ranges of τ and $\Delta\tau$ are the same: $(\mathbb{R}^{+,0} \cup \infty)$.

Definition of the Process class is given as follows:

$$P_{i,j} = \langle F_{i,j}, R_{i,j} \rangle \tag{3.4}$$

where $F_{i,j}$ is a transition function, and $R_{i,j}$ is a set of variable references such that $\{x_\rho\}_{\rho \in R} \subseteq \mathbf{x}$, of the jth Process instance which belongs to Stepper S_i. $R_{i,j}$ consists, in turn, of two subsets.

$$\check{R}_{i,j} \subseteq R_{i,j} \tag{3.5}$$

is the accessor variable reference subset, and

$$\hat{R}_{i,j} \subseteq R_{i,j} \tag{3.6}$$

is the subset of mutator variable references, such that $\check{R} \cup \hat{R} = R$. For brevity we denote R_i to mean $\bigcup_j R_{i,j}$ in the following sections.

F has a general form of:

$$F : (\check{R}^*, \Delta\tau, \tau) \mapsto (\hat{R}^*, \delta\hat{R}) \tag{3.7}$$

where $\delta\hat{R}$ is a set of contributions to derivatives of the mutator variables of F. $(^*)$ means that each reference element is de-referenced to get variable instances; thus $R^* \subseteq \mathbf{x}$.

We refer to a specialized form of F,

$$F : (\check{R}^*, \Delta\tau, \tau) \mapsto (\delta\hat{R}) \tag{3.8}$$

as a *continuous transition function* and

$$F : (\check{R}^*, \Delta\tau, \tau) \mapsto (\hat{R}^*) \tag{3.9}$$

as a *discrete transition function.* For instance, differential equations generally have an even simpler form of the continuous transition function,

$$F : (\check{R}^*, \tau) \mapsto (\delta\hat{R}) \tag{3.10}$$

Difference equations can be viewed as a type of discrete transition function. Discrete event processes, which are constrained to discrete-valued changes in state variables at scheduled time points, are formulated as:

$$F : (\check{R}^*, \tau) \mapsto (\hat{R}^*) \tag{3.11}$$

There is another possible form of discrete transition function:

$$F : (\check{R}^*) \mapsto (\hat{R}^*) \tag{3.12}$$

Although we call this a 'transition' function, this does not explicitly depend on time, and no actual 'transition' is expressed. However, in time-driven simulations, this type of function is sometimes used in conjunction with other time-dependent functions to represent, for example, constraints of system states in a form of algebraic equations, and non-spontaneous discrete transitions triggered by other transition functions.

We call a Stepper made up of Processes with continuous transition functions a *continuous Stepper*, and a *discrete Stepper* if it has Processes with discrete transition functions.

In addition to the Model data structure presented above, the meta-algorithm requires some additional data structures to execute a simulation.

The current global time, T, during a simulation is defined as:

$$T \equiv \min_i(\tau_i + \Delta\tau_i) \tag{3.13}$$

An *updated-time vector* $\mathbf{t} = \{t_0, t_1, \ldots, t_n\} \in (\mathbb{R}^{+,0} \cup \infty)^n$ is used to store the last updated times of all the variables.

A binary relation, which we call the *Stepper Dependency*, D, is defined on an ordered pair of Steppers S_a and S_b as:

$$D = \{(S_a, S_b) | (S_a, S_b) \in S \times S \wedge S_a \neq S_b \wedge \check{R}_a \cap \hat{R}_b \neq \emptyset\} \tag{3.14}$$

The interpretation is very simple; if a Stepper S_b can change a value of at least one variable that the other Stepper S_a can read, S_a is said to be dependent on S_b, that is, $(S_a, S_b) \in D$ (or equivalently, $S_a D S_b$). This relation is not transitive (*i.e.* $S_a D S_b \wedge S_b D S_c$ does not imply $S_a D S_c$), and not symmetric (*i.e.* $S_a D S_b$ does not imply $S_b D S_a$).

3.2.2 Driver algorithm

Time is advanced during the simulation in an iterative fashion. Each iteration of the simulation consists of the following nine procedures.

1. Initialize $\mathbf{t}$ and all the τs to zero.
2. Compute the Stepper Dependency D defined by Equation 3.14 for all combinations of Steppers.
3. From S, pick a Stepper S_i which has the minimum scheduled time $\tau_i + \Delta\tau_i$ over the set.
4. Update the local time of S_i: $\tau_i \leftarrow \tau_i + \Delta\tau_i$.
5. *integrate phase*: Update values of variables R_i^* according to the integration algorithm given below.
6. *step phase*: Allow the Stepper S_i to execute a "step" by calling its implemented "step method" Λ_i. Λ_i may call transition functions $F_{i,j}$ of Processes $P_{i,j}$ in the set P_i according to its implemented algorithm. Λ_i may also update the step size $\Delta\tau_i$, the timescale parameter θ_i, and any other implementation-specific parameters of the Stepper S_i.
7. *dispatchInterruptions phase*: In this procedure the Stepper S_i notifies the change of the variables to other Steppers. S_i calls interruption methods I_j of all Steppers S_j that depend on S_i, *i.e.*, $\{S_j | S_j \in S \wedge (S_j, S_i) \in D\}$. The interruption method I_j may change $\Delta\tau_j$, θ_j, and any other implementation-specific parameters of the Stepper S_j.
8. *log phase*: For post-simulation data processing, record values of variables related to this Stepper, R_i^*, if necessary.

9. End if the termination condition given by the user (such as the value of T and the number of iterations) is met. Otherwise go to 3.

The time is advanced and the values of the state variables are updated in three actions (procedures 4, 5, and 6). Procedure 4 updates the local time τ_i of the Stepper S_i and the global time T. In procedure 5, the variables are updated by continuous Steppers (See Integration Algorithm section). In procedure 6, Λ_i of a discrete Stepper may call its transition functions to change the values of variables R_i^*. A continuous Stepper usually does not change the values of variables in this procedure, but uses Λ_i to recalculate input parameters to its interpolants from $\delta \hat{R}_i$ (See Integration Algorithm and Implementation sections). $\Delta\tau$ is also recalculated in this procedure, and is used in procedure 4 of the next step of the Stepper to advance the local time. In procedures 6 and 7, Steppers set θ_i. θ_i ($\in \mathbb{R}^{+,0} \cup \infty$) is an ancillary parameter which indicates the time scale of change of $\hat{R}_i^*$ and $\delta \hat{R}_i$. Normally $\theta_i > 0$ for continuous Steppers. If S_i is a discrete Stepper, usually $\theta_i = 0$. In procedure 7, Steppers which depend on S_i may use this value as a clue to optimize computation by ignoring some of the interruptions from this Stepper. See the Implementation section for example usage and calculation of θ. If more than one Stepper has the same scheduled time in procedure 3, the one with the highest 'priority' value (defined as part of the model) is chosen.

3.2.3 Integration algorithm

In the integrate phase of the meta-algorithm (procedure 5 above), interpolation is used to recompute the set of variables R_i^*.

In the simplest case, where all the continuous Steppers in the model implement first-order numerical integration algorithms, the procedure has a form of Euler's method with a second order error term. For each $j \in R_i$,

$$x_j(\tau_i) = x_j(t_j) + \Delta t_j \sum_{k \in \tilde{S}} x'_{j,k} + O(\Delta {t_j}^2), \quad \Delta t_j = \tau_i - t_j \tag{3.15}$$

$$x'_{j,k} = \sum_{l \in P_k} \delta\rho_{k,l}, \quad \rho_{k,l} \in R_{k,l} \tag{3.16}$$

where $\tilde{S} \subseteq S$ is the set of continuous Steppers, and t_j is the last updated time of x_j. Here $\delta\rho_{k,l}$ is a contribution to the velocity of change of x_j given by the lth Process of the kth *Stepper*, and t_j is the last updated time of x_j.

Although mathematically concise, this scheme spoils the benefits of higher order algorithms. No matter how high the order of the internal computational procedure being used, the outcome will be integrated as a linear sum of the most recently computed first order derivatives, resulting in a first order global precision. First-order algorithms, however, only occasionally satisfy practical requirements, in terms of accuracy, efficiency and stability, for numerical simulations of continuous systems. Therefore this procedure should be extended to have a higher order error term.

Replacing the first order derivative term of Equation 3.15 by a difference of higher order interpolants we get:

$$x_j(\tau_i) = x_j(t_j) + \sum_{k \in \tilde{S}} \Phi_{k,j}(\tau_i, \Delta t_j) + O(\Delta {t_j}^s) \tag{3.17}$$

where $\Phi_{k,j}(\tau_i, \Delta t_j)$ is an interpolant difference given by the kth Stepper for x_j. The order of the global error term is $s = \min_{k \in \tilde{S}} d_k$. Here d_k is the order of the kth Stepper.

The interpolant difference $\Phi_{k,j}(\tau_i, \Delta t_j)$ is defined as a difference between values of the interpolant $\iota_{k,j}$ at time points τ_i and $\tau_i - \Delta t_j$.

$$\Phi_{k,j}(\tau_i, \Delta t_j) = \iota_{k,j}(\tau_i) - \iota_{k,j}(\tau_i - \Delta t_j) \tag{3.18}$$

The relationship between τ_i and t_j is depicted in Figure 3.2.

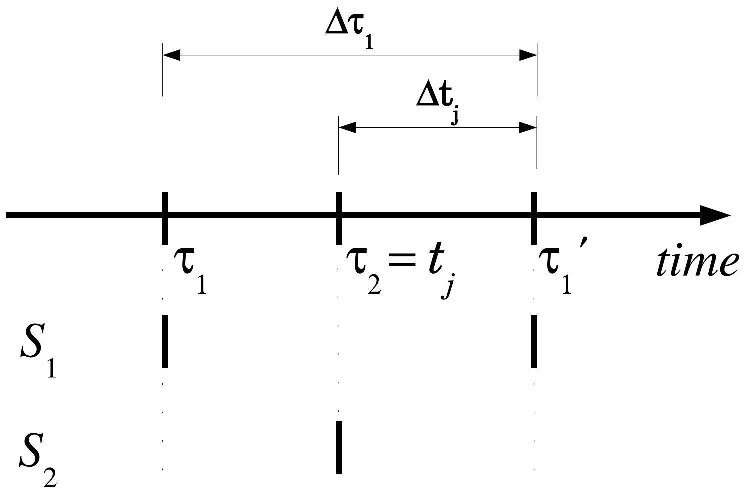


Figure 3.2: Relation between τ and t
This example has only two continuous Steppers, S_1 and S_2. Both share a variable x_j. Time points τ_1 and τ_2 are where the last steps of these Steppers occurred, and τ_1' $(= \tau_1 + \Delta\tau_1 = T)$ is the current time. S_1 steps at τ_1 and τ_1'. S_2 steps at τ_2. x_j is updated at τ_1, τ_1', and τ_2. If the last Stepper stepped before τ_1' is S_2, then $\tau_2 = t_j = \tau_1' - \Delta t_j$.

To minimize the total expected error over the whole model, the input parameters to each interpolant should be recomputed upon receiving an interruption (procedure 7 above). This functionality is not guaranteed by the meta-algorithm, and is a requirement in the implementation of specific algorithm modules, some examples of which are described below.

3.3 Some algorithm modules

Three sub-classes of Stepper, *DiscreteEventStepper*, *DiscreteTimeStepper* and *DifferentialStepper*, are provided according to Zeigler's classification of simulation algorithms (Zeigler *et al.,*, 2000). Each concrete implementation of an algorithm is a subclass of one of these base classes.

At least the step method Λ, the interruption method I, and a function to determine the timescale parameter θ must be defined in each implementation of a Stepper.

3.3.1 Differential equation modules

As subclasses of the DifferentialStepper, two general-purpose ordinary differential equation (ODE) solvers have been implemented. Many variants of the Runge-Kutta algorithm exist, and the general form of the algorithm is given as follows:

$$k_j = f\left(x_n + c_j h, y_n + h\sum_{i=1}^{s} a_{j,i} k_i\right),\ j = 1 \dots s$$
$$y_{n+1} = y_n + h\sum_{j=1}^{s} b_j k_j \tag{3.19}$$

where h is the step size, s is the number of right hand side (RHS) evaluations required for a single step. Specification of the matrix of coefficients a, b, and c determine each specific variation of the algorithm.

Two instances of embedded explicit Runge-Kutta algorithms with dense output capabilities have been implemented: second-order Fehlberg with a third-order error estimation (Fehlberg 2(3)) (Fehlberg, 1969), and fourth-order Dormand-Prince with a fifth-order error term (Dormand-Prince 5(4)7M) (Dormand and Prince, 1980).

For some integration algorithms including these two, it is possible to derive interpolants by using Hermite polynomial interpolation, the general form of which is:

$$g(x) = \sum_i \sum_j \alpha_{i,j}(x) f_i^{(j)} \left[\frac{d^j}{dx^j} \alpha_{i,j}(x) \right]_{x=x_k} = \delta_{i,k}$$

where $x_k (k = 1, 2, \cdots, n)$ is the time point where input is given, $f_i^{(j)}$ is the input function defined at x_i as the jth order derivative, and $\delta_{i,k}$ is the Kronecker delta. If $f_i^{(j)}$ defines data points at m different combinations of i and j, then it can make an interpolant of order $m-1$.

A C^0 interpolant of the second-order Fehlberg algorithm can easily be derived because we have three data points, the initial value, $(x_j(\tau_k))$, the value at $\tau_i + \Delta\tau_i$, $(x_j(\tau_k) + \sum_{i=1}^{3} b_i k_i)$, and the first-order derivative value at the initial point, (k_1).

$$\begin{aligned} \iota_{k,j}(t) = (1-\sigma^2)\ x_j(\tau_k)\ +\ (-\sigma^2+\sigma)\Delta\tau_k k_1 \\ + \sigma^2 \left(x_j(\tau_k) + \Delta\tau_k \sum_{i=1}^{3} b_i k_i \right) + O({\Delta t_j}^3) \end{aligned} \tag{3.20}$$

$$\sigma = (t - \tau_k)/h, \quad 0 < \sigma \leq 1 \tag{3.21}$$

where τ_k and $\Delta\tau_k$ are the current time and the current step size of this Stepper, respectively. Here $\alpha_{0,0} = (1-\sigma^2)$, $\alpha_{0,1} = (-\sigma^2 + \sigma)$, and $\alpha_{1,1} = \sigma^2$. $\Delta\tau_k$ appears in the second and the third terms of the RHS because of the change of variable from t to σ. Table 3.3.1 is the Butcher array of the method with the dense output coefficients σ and σ^2.

Shampine (1986) gives a C^1 interpolant of the Dormand-Prince 5(4)7M with additional

Table 3.1: Dense output coefficients and the Butcher array of Runge-Kutta Fehlberg 2(3)

	0			a_{ij}
c_j	1	1		
	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	
	b_i	$\frac{1}{2}$	$\frac{1}{2}$	0
	b_i^*	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{2}{3}$
	σ	1	0	0
	σ^2	$-\frac{1}{2}$	$\frac{1}{2}$	0

coefficients b_i^*:

$$\begin{aligned}
\iota_{k,j}(t) = x_j(\tau_k) + \sigma h k_1 \; &+ \; 4\sigma^2 h(\sigma - 1)(k_1 - v_{\frac{1}{2}}) \\
&+ 2\sigma^2 h(\sigma - \frac{1}{2})(v_1 - k_1) \\
&+ 2\sigma^2 h(\sigma - \frac{1}{2})(\sigma - 1)(k_7 - k_1 + 4v_{\frac{1}{2}} - 4v_1) \\
&+ O(\Delta t_j{}^5), \\
v_{\frac{1}{2}} = \sum_{i=1}^{7} b_i^* k_i,\; v_1 &= \sum_{i=1}^{7} b_i k_i
\end{aligned} \tag{3.22}$$

The number of RHS evaluations required in each step of the simulation is three for Fehlberg 2(3).

Interestingly, $k_7^{[n]} = k_1^{[n+1]}$ for the Dormand-Prince 5(4)7M algorithm if there is no interruption in a time interval $(\tau_n, \tau_{n+1}]$. Thus the number of RHS evaluations required by the algorithm is seven for starting up, and six or seven during the simulation.

The step method Λ for these ODE modules are implemented with a standard adaptive step sizing mechanism (Press *et al.*,, 2002). In addition, time step sizes are constrained by the following condition to ensure that the Stepper does not change variable values too much at once in a single step:

$$\forall j \in \hat{R}_i : x_j \epsilon_{rel} + \epsilon_{abs} \geq \sum_{l \in P_i} \delta\rho_{i,l} \cdot \Delta\tau_i \tag{3.23}$$

where ϵ_{abs} and ϵ_{rel} (usually $\lesssim 0.1$) are absolute and relative constraint parameters of changes of the variables in a step, and $\rho_{i,l}$ is a variable reference of the lth Process in the ith Stepper pointing to x_j. The constraint parameters ϵ_{abs} and ϵ_{rel} should be provided according to external conditions given to this Stepper, such as the total number of continuous Steppers and their implemented algorithms.

In this implementation θ_i has the same value as $\Delta\tau_i$.

The interruption method I_i for the ith ODE solver is defined as follows:

1. If the timescale of the interrupter is smaller than the step size of this Stepper, $\theta_k < \Delta\tau_i$, then use θ_k as the initial guess of the next step size. Otherwise, just return ignoring this interruption.
2. If the next step of the interrupter is before this Stepper, $(\tau_i + \Delta\tau_i > \tau_k + \Delta\tau_k)$, set $\Delta\tau_i$ to $T - \tau_i$.

Procedure 2 makes sure that the interrupted Stepper S_i steps at least once between the current and the next steps of the interrupting Stepper S_k. This guarantees that the constraint in Equation 3.23 takes effect on the interrupted Steppers. Procedure 1 is an optimization. If the interrupted steps more frequently than the interrupter's timescale parameter θ_k, the constraint is assumed to be satisfied without the interruption. By using θ_k as the next step size of the interrupted Stepper, this optimization more likely happens again in the next step.

3.3.2 Gillespie-Gibson module

A standard Gillespie-Gibson algorithm, or the Next Reaction Method, has been implemented as a subclass of the DiscreteEventStepper in a fully object-oriented fashion. The probability density function for the next reaction to occur at time τ, $P(\tau)$, and the probability that this reaction is of type μ, $Pr(\mu)$, are given by the following two equations.

$$P(\tau) = \left(\sum_j a_j\right) \exp\left(-\tau \sum_j a_j\right) \tag{3.24}$$

$$Pr(\mu) = a_\mu / \sum_j a_j \tag{3.25}$$

where a_j is the propensity function of the jth reaction. In this implementation, we only consider uni- and bi-molecular reactions, and if X_i is not an integer, a floor is taken to calculate the propensity.

The interruption method I has been implemented to enable synchronization with other modules. The procedure is:

1. Recalculate the propensities of the reactions.
2. Redetermine the time of the next reaction $\tau_i + \Delta\tau_i$ and the next reaction μ according to the main algorithm.
3. Reschedule the Stepper to time $\tau_i + \Delta\tau_i$ on the scheduler.

The timescale θ_i is determined as:

$$\theta_i = \xi \cdot \Delta\tau_i \min_{\rho \in R_{i,\mu}} \frac{\rho^*}{|\nu_\rho|} \tag{3.26}$$

where ξ is a tolerance parameter, i is an index of the Stepper, and ν_ρ is the stoichiometric coefficient for the variable reference ρ defined as a part of the model. ξ takes a small number such as 0.1, or 0. $\xi \neq 0$ directs it to pretend to be a continuous component. This may make the simulation more efficient if this Stepper takes smaller step sizes than other parts of the model. For instance, $\xi = 0.1$ lets θ be the expected time for the reactant of the current Process with the smallest population to change its value by 10 %. If precision precedes efficiency, set $\xi = 0$.

The Mersenne-Twister algorithm (Matsumoto and Nishimura, 1998) has been employed as a pseudo-random number generator.

3.4 Results

Two relatively simple models have been constructed to examine the performance of the meta-algorithm. The first is a model of the heat-shock response, and has three variations, deterministic (ODE), stochastic (Gillespie), and composite (ODE/Gillespie). This example is to show multi-algorithm and multi-timescale capabilities of the algorithm. The second example is a simple cascade of coupled harmonic oscillators with gradually changing timescales that span six orders of magnitude.

All the timings and results given in this work have been taken on a PC running RedHat Linux 9 operating system with a dual Hyper-Threading-enabled Intel Xeon 2.8GHz CPU and 1GB of RAM. This implementation is a single-thread application.

3.4.1 A multi-algorithm heat-shock model

When cells are exposed to high temperature, the synthesis of a small number of proteins known as heat-shock proteins becomes selective and rapid (Gross, 1996). This process is called the heat-shock response. σ^{32}, a variation of the σ subunit of RNA polymerase, has been implicated as the global regulator for this system (Grossman *et al.,*, 1987). DnaJ is a molecular chaperon that plays an important role in regulating the activity and stability of σ^{32} (Gamer *et al.,*, 1996). Many molecular species involved in this process are present in small numbers, most of which relate to gene expression processes, and require stochastic simulation. In contrast, protein folding involves large quantities of molecules, making stochastic simulation computationally challenging.

To evaluate the performance of a composite simulation scheme, we set up a pure stochastic model, a pure ODE model and a stochastic/ODE composite model, based on Srivastava's stochastic petri net model of the *E. coli* heat-shock (Srivastava *et al.,*, 2001). Protein folding/unfolding processes have been added to the model, and modeled as differential equations in the ODE and composite models, and using the Gillespie algorithm in the stochastic model (Figure 3.3). The list of reactions and initial values of this model are shown in Table 3.2 and 3.3, respectively. The difference between Srivastava's original model and the model used in this work is also described in the legend for Figure 3.3. All parameter settings in the three variations of the model are identical.

We ran each model ten times for 100 seconds and traced the quantities of σ^{32} and DnaJ to benchmark the performance. These two species were selected because σ^{32} is biologically important, as it controls the expression level of heat shock proteins, and DnaJ is on the boundary of the different algorithms in the composite model. The composite model ran about 2.6 times as fast as the ODE model, and 351 times faster than the stochastic model (Table 3.4). The difference in mean steady-state levels in these three variations were within 2.7% for σ^{32}, and 0.0008% for DnaJ. The standard deviation of σ^{32} in the composite run was indistinguishable from that of the stochastic run. In contrast, the trajectory of DnaJ was devoid of stochastic fluctuations in the composite model (Table 3.4, Figure 3.4). The total number of protein molecules in this

Table 3.2: Reaction list of heat shock model

Reaction	Parameter
$DNA.\sigma^{32} \rightarrow mRNA.\sigma^{32}$	$1.4 \times 10^{-3} s^{-1}$
$mRNA.\sigma^{32} \rightarrow \sigma^{32} + mRNA.\sigma^{32}$	$0.07 s^{-1}$
$mRNA.\sigma^{32} \rightarrow degradation$	$1.4 \times 10^{-6} s^{-1}$
$\sigma^{32} \rightarrow RNAP\sigma^{32}$	$0.7 s^{-1}$
$RNAP\sigma^{32} \rightarrow \sigma^{32}$	$0.13 s^{-1}$
$DNA.DnaJ + RNAP\sigma^{32} \rightarrow DnaJ + DNA.DnaJ + \sigma^{32}$	$4.41 \times^{6} M^{-1} s^{-1}$
$DnaJ \rightarrow degradation$	$6.4 \times 10^{-10} s^{-1}$
$\sigma^{32} + DnaJ \rightarrow \sigma^{32}.DnaJ$	$3.27 \times 10^{5} M^{-1} s^{-1}$
$\sigma^{32}.DnaJ \rightarrow \sigma^{32} + DnaJ$	$4.4 \times 10^{-4} s^{-1}$
$DNA.FtsH + RNAP\sigma^{32} \rightarrow FtsH + DNA.FtsH + \sigma^{32}$	$4.41 \times 10^{6} M^{-1} s^{-1}$
$FtsH \rightarrow degradation$	$7.4 \times 10^{-11} s^{-1}$
$\sigma^{32}.DnaJ + FtsH \rightarrow DnaJ + FtsH$	$1.28 \times 10^{3} M^{-1} s^{-1}$
$DNA.GroEL + RNAP\sigma^{32} \rightarrow GroEL + DNA.GroEL + \sigma^{32}$	$5.69 \times 10^{6} M^{-1} s^{-1}$
$GroEL \rightarrow degradation$	$1.8 \times 10^{-8} s^{-1}$
$^{*}Protein \rightarrow UnfoldedProtein$	$0.2 s^{-1}$
$^{*}DnaJ + UnfoldedProtein \rightarrow DnaJ.UnfoldedProtein$	$9.7256 \times 10^{6} M^{-1} s^{-1}$
$^{*}DnaJ.UnfoldedProtein \rightarrow Protein + DnaJ$	$0.2 s^{-1}$

Except the parameters for σ^{32} transcription, translation, and the last three reactions, all the parameters are from Srivastava's model. The partitioning of this heat-shock model for the composite run was based on the propensities of the reactions. When reactants are in small numbers, the propensity of the reaction is smaller and the fluctuation or noise is bigger, which make stochastic method essential. Reactions with asterisks (*), which consume protein molecules, have at least two order higher numbers of reactant molecules than most of the other reactions. These reactions were modeled using ODE method in the composite and pure deterministic models, and Gillespie in the pure stochastic model. All the other reactions were modeled with Gillespie method in the pure stochastic and the composite model, and ODE in the pure deterministic model.

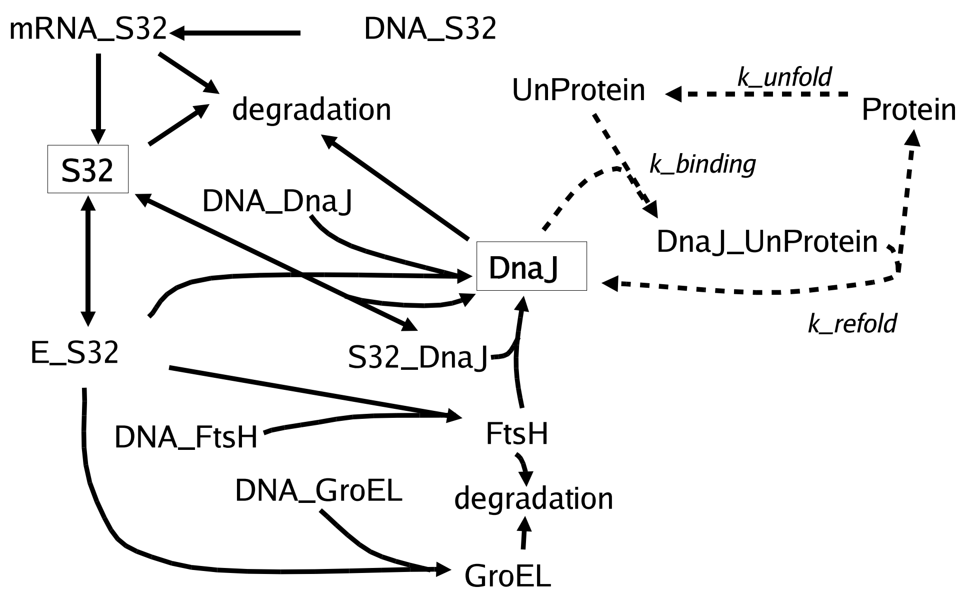


Figure 3.3: The heat-shock demonstration model
Model scheme for the heat-shock model. S32 means σ^{32}; E_S32 means the RNAP core enzyme with σ^{32}; Protein means folded protein and UnProtein means unfolded protein. Dashed lines in the upper-right corner represent reaction modeled using ODE in the composite model, and Gillespie in the stochastic model. Rate constants for σ^{32} transcription and translation are 1.4×10^{-3} and 7×10^{-2}, respectively, which are the only different parameters from the Srivastava's model. $k\_unfold = k\_binding = 0.2$, and $k\_refold = 9.73 \times 10^{6}$.

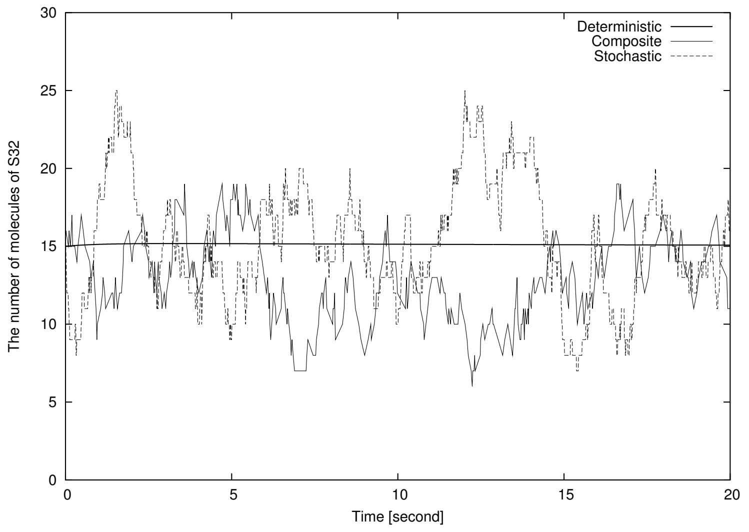
example model is of order $10^6 - 10^7$.

3.4.2 A simple model of multi-timescale oscillators

A cascade of twenty harmonic oscillator components $X_0, X_1, \cdots, X_{19}$ have been constructed. Each component X_n is coupled with the next one X_{n+1} with a positive feed-forward. The rate constant of X_n is doubled for X_{n+1} (Figure 3.5), resulting in a total timescale difference in the model of about $\log_{10}(2^{20}) \simeq 6.0$ orders of magnitude. Five sets of the cascade are included in a model.

Benchmarking was conducted on the model with one, two, four, ten, and twenty Step-

(a)



(b)

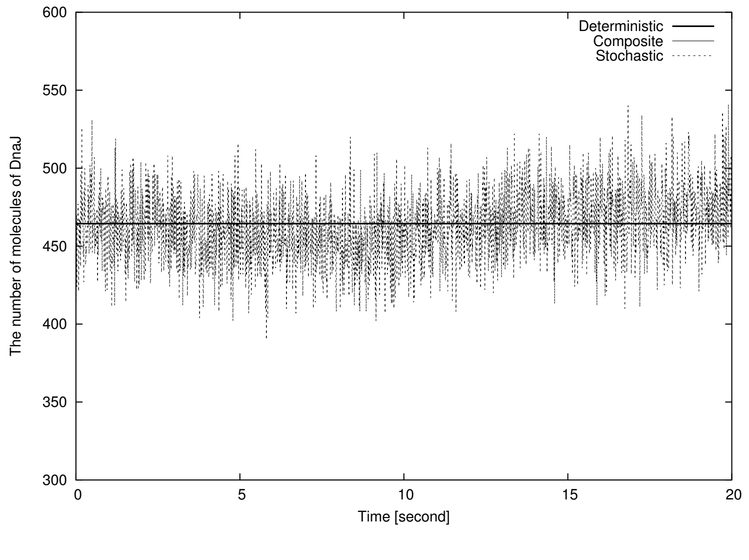


Figure 3.4: Simulation results of the the heat-shock model
Comparisons of (a) σ^{32} and (b) DnaJ time courses of the first 20 seconds of the simulation for the stochastic model, the composite model, and the deterministic model. The composite and the ODE trajectories of DnaJ overlap. It can be seen that means of the ODE (fine curve), the stochastic (dashed curve), and the composite (thick curve) models agree well both in σ^{32} and DnaJ cases. The standard deviations of σ^{32} in the stochastic and composite models are equivalent. In contrast, the stochastic fluctuations cannot be seen for the DnaJ trajectory of the composite run, making it equivalent to the ODE model (see Table 3.4 for numerical values).

Table 3.3: List of initial values used in heat shock model

Variable	Initial Values	Unit
DNA.σ^{32}	1	number
mRNA.σ^{32}	17	number
σ^{32}	15	number
RNAPσ^{32}	76	number
DNA_DnaJ	1	number
DNA_FtsH	0	number
DNA_GroEL	1	number
DnaJ	464	number
FtsH	200	number
GroEL	4314	number
DnaJ_UnfoldedProtein	5e6	number
Protein	5e6	number
$\sigma^{32}.DnaJ$	2959	number
Cell volume	1.5e-15	liter
UnfoldedProtein	2e5	number

pers (Table 3.4.2). As an example, in the two Steppers variant, the components were divided into two groups, $X_0, \cdots, X_9$ and $X_{10}, \cdots, X_{19}$. Relative and absolute differences between results from these variations were smaller than the error tolerance parameter given to the Steppers (10^{-6}). The simulation speed improved as the number of Steppers was increased. The twenty Steppers variant achieved almost an order of magnitude better performance than the original.

Table 3.4: Benchmark results of the heat-shock model

	Timing (sec.)	Speed	σ^{32} Std. Dev.	σ^{32} Level	DnaJ Std. Dev.	DnaJ Level
Stochastic	507.7±0.76	1	3.74±0.072	15.18±0.391	5.273±0.206	464.4±0.498
Composite	1.445±3.5e-4	351	3.64±0.112	14.83±0.348	3.156e-2±0.282e-2	464.4±0.007
Deterministic	3.703	137	0	15.01	0	464.4

The three variations, stochastic, composite, and deterministic, of the heat-shock model were run for 100 seconds. Each timing is an average of five runs. $\xi = 0$. The mean levels and the standard deviations of σ^{32} were calculated over the simulation results, and trapezoid method was used for the mean calculation. Dormand-Prince 5(4)7M algorithm has been used for ODE parts in the composite and deterministic runs. The results do not include times for program invocation and parsing of the XML model file. See also Figure 3.4.

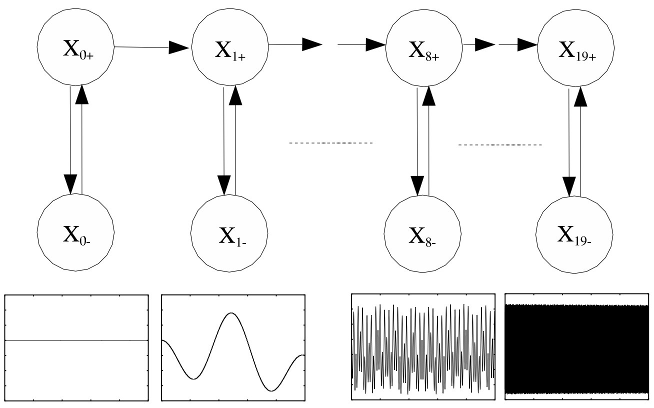


Figure 3.5: The cascade oscillators demonstration model
The model has a basic unit of two variables coupled by the following equations: $\frac{d[X_{i+}]}{dt} = -k2^i[X_{i-}] + k2^{i-1}[X_{(i-1)-}]$, $\frac{d[X_{i-}]}{dt} = k2^i[X_{i+}]$ where k is a time scaling factor. A cascade has twenty instances of the unit. The model has five sets of the cascade. Four sample solutions of the cascade oscillators model are shown below the first (slowest), second, ninth, and twentieth (fastest) components.

3.5 Discussion

3.5.1 ODE / Gillespie composition

In the limit of large numbers of reactant molecules, stochastic and deterministic simulations are equivalent (Gillespie, 1977). In contrast, if the system has low copy numbers of species, the deterministic law of mass action breaks down because the steady-state fluctuations in the number of molecules (which is proportional to the square root of the number of molecules) becomes a significant factor in the behavior of the system (Singer, 1953). ODE models isolate the biochemical system into a group of deterministic and continuous reactions, and tacitly ignore fluctuations in the pathway (Rao *et al.*,, 2002). With identical parameter settings, the stochastic and deterministic models can produce different results, and stochastic models are generally believed to be more accurate (Marion *et al.*,, 1998; Srivastava *et al.*,, 2002).

Table 3.5: Benchmark results of the cascade oscillators model

$\lvert S \rvert$	Predicted Cost	Timing	Predicted Speed	Speed
1	$2^{19}{\cdot}20$ [100%]	68m43s	1.00	1.00
2	$2^9{\cdot}10 + 2^{19}{\cdot}10$ [$\sim$ 50%]	35m13s	2.00	1.95
4	$5(2^4 + 2^9 + 2^{14} + 2^{19})$ [$\sim$ 26%]	18m16s	3.88	3.76
10	$4{\cdot}\frac{4^{10}-1}{4-1}$ [$\sim$ 13%]	9m33s	7.50	7.20
20	$\frac{2^{20}-1}{2-1}$ [$\sim$ 10%]	7m10s	10.00	9.5

The cascade oscillators model was run for 0.5 seconds. The components were equally divided into one, two, four, ten, and twenty Steppers. An average of three runs is shown for each configuration. Dormand-Prince 5(4)7M algorithm has been used for all the components. The timings include a program startup and an XML parsing. The predicted simulation costs are calculated by assuming that it is directly proportional to values of time constants.

Despite its advantages, however, Gillespie's scheme of exact stochastic simulation has limited utility due to its computational cost which is proportional to the number of molecules. The composite simulation of the heat-shock demonstration model shows that, if carefully chosen, ODEs can replace some parts of the stochastic model, resulting in a dramatic improvement in performance without jeopardizing the benefits of the original purely stochastic model. A logical extension to this "static" combination of the deterministic and stochastic schemes would be "dynamic" switching between these different simulation methods. This will be further explored in future work.

3.5.2 Related works in stochastic/deterministic hybrid simulation

We know of at least three related works in hybridizing deterministic and stochastic biochemical simulation schemes. Zak *et al.,* (2003) (see supplementary information) introduces a hybrid method coupling the implicit Euler method and Gillespie's Direct method. However, the design of this method relies on some assumptions specific to their gene regulation model, such as unidirectional coupling from the deterministic to the stochastic sub-model. Kiehl *et al.,* (2004) also has a type of hybrid algorithm combining the Gillespie's Direct method and Runge-Kutta ODE solvers. This method has a mechanism of interpolation of ODE-side trajectories that is similar to what we have employed in the implementation of the ODE algorithm modules. In addition, this method generates time-varying random distribution to make full use of the information given by the interpolated trajectory. This improvement will be considered in our framework too. Vasudeva and Bhalla (2004) has a fixed-step size method in which reactions adaptively switch between stochastic and deterministic methods. This adaptive switching is another possible improvement to our framework of ODE/Gillespie composition.

Unlike the novel framework that takes the embedded formalism approach, all works introduced here basically aim at coupling of Gillespie-type stochastic methods and ODE solvers, and thus are taking the combined approach with which modularity is limited.

Although this is not a combination of stochastic and deterministic methods, Haseltine and Rawlings (2002) proposed a stochastic partitioning of chemically reacting systems into slow and fast subsets, and applying an exact and an approximate (Langevin) variations of the Gillespie algorithms for fast and slow parts, respectively.

3.5.3 Performance analysis of the heat-shock model

The composite heat-shock model runs faster than both the ODE and the Gillespie models. The following crude approximation of computational costs of these models withstand some discussion.

$$\begin{aligned}
O_{Gillespie} \propto & \left[\sum a\right] \log N \\
O_{ODE} \propto & \gamma \left[\frac{\partial f}{\partial x}\right] N \\
O_{Composite} \propto & \left[\sum a\right]_m \log N_m + \gamma \left[\frac{\partial f}{\partial x}\right]_f N_f
\end{aligned}$$

where $[\frac{\partial f}{\partial x}]$ is a measure of the degree of stiffness, N is the number of reactions, $[\sum a]$ denote the total propensity, and γ is a constant parameter to relate $[\frac{\partial f}{\partial x}]$ and $[\sum a]$. Subscripts m and f indicate the main part and the folding part of the model, respectively. Obviously $N = N_m + N_f$. Computational cost of explicit ODE solvers such as Dormand-Prince 5(4)7M is largely determined by the degree of stiffness, which is dominated by $\frac{\partial f}{\partial x}$ (where x is a dependent variable, and f is the derivative function for x). The large (242-fold) performance difference between the ODE and Gillespie implies the following:

$$\gamma \left[\frac{\partial f}{\partial x}\right] N \ll \left[\sum a\right] \log N \tag{3.27}$$

In fact, $\Delta\tau$ of the ODE Stepper was about 10^{-3}, while the step size of the Gillespie-Gibson Stepper was typically in the range of 10^{-6} – 10^{-9}. The composite model is faster than the ODE model, that is,

$$\left[\sum a\right]_m \log N_m + \left[\frac{\partial f}{\partial x}\right]_f N_f < \left[\frac{\partial f}{\partial x}\right] N \tag{3.28}$$

if

$$\left[\frac{\partial f}{\partial x}\right]_f N_f < \left[\frac{\partial f}{\partial x}\right] N \tag{3.29}$$

and

$$\left[\sum a\right]_m \log N_m \ll \gamma \left[\frac{\partial f}{\partial x}\right] N \tag{3.30}$$

Because the degree of stiffness is dominated by the fastest component of the system, it can be assumed that

$$\left[\frac{\partial f}{\partial x}\right]_f \simeq \left[\frac{\partial f}{\partial x}\right] \tag{3.31}$$

Thus the first proposition (3.29) can obviously be verified by

$$N_f < N \tag{3.32}$$

The latter proposition (3.30) is consistent with the $\Delta\tau$ observed in Gillespie simulation of the main model in isolation (10^{-1} – 10^{-3}). This speed difference, however, is less important than that of the Gillespie and the composite models because the dependency of the ODE solvers to stiffness can partially be overcome by use of an implicit or solver (Gear, 1971). Now the difference between the composite model and Gillespie model, which is denoted as:

$$\left[\sum a\right]_m \log N_m + \gamma \left[\frac{\partial f}{\partial x}\right]_f N_f \ll \left[\sum a\right] \log N \tag{3.33}$$

is explained because

$$\left[\sum a\right]_m \ll \left[\sum a\right] \tag{3.34}$$

and

$$\gamma \left[\frac{\partial f}{\partial x}\right]_f N_f < \gamma \left[\frac{\partial f}{\partial x}\right] N \ll \left[\sum a\right] \log N \tag{3.35}$$

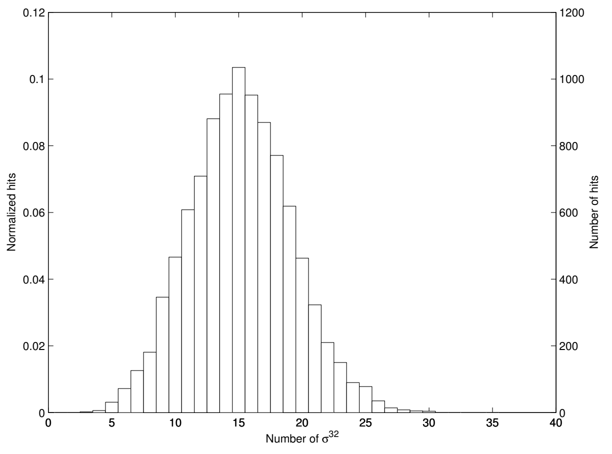
3.5.4 Accuracy of the composite heat-shock simulation

In the composite model, σ^{32} behaves identically with the pure stochastic model, and the trajectory of DnaJ almost matches that of the deterministic model. σ^{32} is one step away, and DnaJ is on, the boundary between the deterministic and the stochastic parts of the model. In the composite simulation, stochastic fluctuation of DnaJ is absorbed by the fast rates of the deterministic reactions involved in controlling its steady-state value. The primary path from the boundary (DnaJ) to σ^{32} is a slow reaction (dissociation of σ^{32} / DnaJ complex, $k = 4.4 \times 10^{-4}$) which filters higher frequency fluctuations. Thus σ^{32} is insensitive to such rapid fluctuations even in the pure stochastic model, and no difference is observed between the stochastic and composite runs in the first and second moments of the time course. An analysis of higher moments showed a slight difference in skewness, and no difference in kurtosis (Figure 3.5.4). However, as the behavior of DnaJ exemplifies, the composite simulation must be used with caution if the stochastic effect on species near the boundary is of interest, or is expected to influence the overall behavior of the model.

3.5.5 Legitimacy of the recalculation upon interruption of the Gillespie-Gibson Stepper

The Gillespie-Gibson Stepper recalculates putative times of all the reactions upon interruptions by other Steppers. The correctness of these recalculated times in a composite simulation is guaranteed because the interruption procedure itself does not affect the probability density $P(\tau)$. To verify this, consider an interruption at τ', ($\tau' < \tau$), the probability density of the next reaction time, $\tilde{P}(\tau)$, is then calculated by the following

(a)



(b)

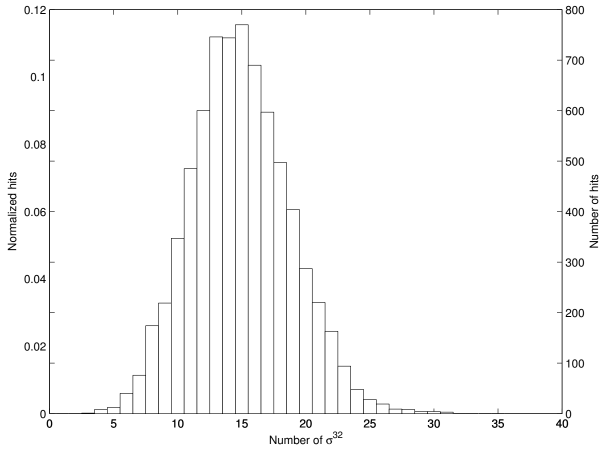


Figure 3.6: Normalized comparison of distribution of σ^{32} in heat shock model. Distributions of σ^{32} values in the (a) stochastic and (b) composite simulations of the heat-shock model. These histograms are drawn over ten runs of the 100 second simulations. As shown in the Table 1 of the main paper, means and standard deviations agree well. Unlike these lower moments, the shapes of the distributions have a slight difference. This difference appears in skewness of the statistics: 0.1148 ± 0.0384 in the stochastic run and 0.2079 ± 0.0474 in the composite run. Again, the fourth order moments (kurtosis) are equivalent: 2.9060 ± 0.1005 in stochastic and 2.9191 ± 0.0821 in composite.

equation.

$$\begin{aligned}
\tilde{P}(\tau) &= \left(1 - \int_0^{\tau'} P(t)dt\right) \cdot P(\tau - \tau') \qquad (3.36)\\
&= \exp\left(-\tau' \sum_j a_j\right) \cdot P(\tau - \tau')\\
&= \exp\left(-\tau' \sum_j a_j\right) \cdot \left(\sum_j a_j\right) \exp\left(-(\tau - \tau') \sum_j a_j\right)\\
&= \left(\sum_j a_j\right) \exp\left(-\tau \sum_j a_j\right)
\end{aligned}$$

This is equal to the original function in Equation 3.24.

3.5.6 Performance analysis of the coupled oscillator model

The coupled oscillator model demonstrates the efficiency with which multi-timescale phenomenon can be modeled in our novel framework. In this toy model, slower components were unidirectionally connected to faster components. This type of coupling is commonly observed in the cell. For instance, expression of enzymes from genes, which is relatively slow, controls metabolic reactions. Conversely, if there is a feedback from the faster components to the slower ones, which is also common in the cell, frequent synchronizations are required. In this case the performance of this implementation is expected to be the same as, or worse than, conventional synchronous single-algorithm simulators because of the interruption overhead. However, if the slow component is assumed to be insensitive to perturbations on the faster component's timescale, synchronization costs could be reduced. This improvement will also be considered in future work to improve the meta-algorithm.

3.5.7 Strengths and weaknesses of the meta-algorithm

As discussed in the 'design options' section, there are two design schemes of multi-formalism simulation frameworks; combining and embedding, and our meta-algorithm takes the latter embedding approach.

A major source of strengths of the meta-algorithm approach is that it enforces modularity, or context-independent design, on implemented modules. Each simulation algorithm module and thus submodel is required to have a well-defined interface of time-scheduling and communications with other modules. Good outcomes of the modularity of the meta-algorithm are:

- Once an algorithm is implemented in a modular way, it can be used in combination with any other algorithm modules.
- Implementations of algorithm modules are often simpler and maintainable than combined forms of the same set of algorithms.

On the other hand, one drawback in this modular design is that, because of the inherent 'interface barriers' between sub-models, dynamic switching between algorithms implemented in different modules is still a future work, and currently not available. This is immediately an advantage of the combined approach. For example, Gillespie's tau leaping method can be viewed as a combination of an exact method and the approximate tau-leaping scheme, and it switches to the exact method when the approximation becomes inappropriate. These methods connect smoothly to each other in a natural way as the scales of concentration and propensity changes.

Let's take a more close look at this problem. When integrating two different simulation algorithms, there can be two cases. The first being a combination of two completely different simulation algorithms that treat different formalisms. An example is a combination of a stochastic elementary reaction submodel, for which Gillespie algorithm is often used, and an ODE sub-model of Michaelis-Menten-type enzymatic complex reactions. In a biochemical sense, these two sub-models require different types of kinetic parameters and modeling approaches, and these are not inter-convertible. Another example is when we have a model of metabolism as a diffusion-reaction cellular automata, and want to co-simulate this with a boolean-network model of gene expression. Call this the multiple modeling formalism case.

The other instance is that when we have a set of physical processes (such as chemical reactions) of the same abstraction level. One common circumstance in computational cell biology is that the model to be simulated consists exclusively of elementary reactions. As we have seen, this is the case where Gillespie-family of algorithms are frequently used. The system is homogeneous in terms of modeling formalism. We can apply both single and multi-algorithm simulators for the same type of problems. Tau-leaping algorithm, as discussed above, is a good example of this. Dynamic switching between the exact methods and Langevin or ODE methods could sometimes be used too. The biggest reason to adopt multi-algorithm simulator here is efficiency and accuracy, because except for these factors single- and multi-algorithm runs are essentially identical to each other. We call this the single modeling formalism case.

From these observations, it can be concluded that while in general embedding is preferable because of its modularity, combining can be advantageous when different algorithms are combined to handle changing scales in single-formalism models.

In fact, in addition to embedding, the meta-algorithm framework supports combining by allowing algorithms in combined forms to be implemented as algorithm modules. Also, meta-algorithm's embedding framework itself could be extended to support adaptive switching between algorithm by providing a mechanism of 'Process migration', which allows Process objects to migrate among multiple Steppers of different algorithms during a simulation.

Consequently, there are four cases in multi-algorithm simulations.

1. Single modeling formalism / combined simulation algorithms.

   This is the case of the elementary reactions / Gillespie-family methods.

2. Single modeling formalism / embedded simulation algorithms.

   This is what we explored in the composite *E. coli* heat-shock model. Although adaptive switching between simulation algorithms require a special mechanism (*e.g.* Process migration), a good point is that the modularity in each algorithm module is assured.

3. Multiple modeling formalisms / combined simulation algorithms.

   Generally, this approach only occasionally results in a good simulator design, as it requires a new simulation algorithm to be developed for each distinct combination of modeling formalisms, unless there is an existing algorithm generic enough to run these formalisms simultaneously.

   Zeigler's DEVS&DESS is an exceptionally useful example here, because it combines two very general formalisms; discrete event and differential equation.

4. Multiple modeling formalisms / embedded simulation algorithms.

   This is where the meta-algorithm framework shows its potency to the full. Without this kind of computational platforms, composition of sub-models that require different simulation approaches is impossible in the first place.

Needless to say, when a model has more than two modeling formalisms or simulation algorithms, things are more complicated. The meta-algorithm framework supports all of those cases.

3.5.8 Parallelization of the meta-algorithm

There can be two levels of parallel computation in the meta-algorithm framework. The first possibility is parallelization of the step method Λ_i and the interruption method I_i of each Stepper S_i. This sub-Stepper level parallelization scheme would be effective when the computation to be done by a Stepper at each simulation step is heavy and readily parallelizable, for instance, calculations of molecular diffusion, using parallel computation frameworks such as OpenMP. This level of parallelism can be achieved by the implementation of each Stepper module, and basically does not require special support by the meta-algorithm itself.

The other strategy of parallelization is at a Stepper level, as opposed to the sub-Stepper parallelism. In procedure (3) of the driver algorithm, after picking up the next Stepper S_i, the scheduler can check the other Stepper S_k immediately after S_i on the schedule queue satisfied the following condition,

$$\{(S_i, S_k), (S_k, S_i)\} \nsubseteq D \wedge \{S_j | \tau_i + \Delta_i \leq \tau_j + \Delta_j \leq \tau_k + \Delta_k\} = \emptyset, \tag{3.37}$$

and if it is true, procedures (4) to (8) of Steppers S_i and S_k can be executed concurrently. If S_i is replaced with the set of already running Steppers, $S_{running}$, this parallel scheduling framework can easily be extended for multiple Stepper cases where more than two Steppers may run in parallel.

$$\begin{aligned} S_{startable} = \quad & \{S_k | S_i \in S_{running} \wedge \{(S_i, S_k), (S_k, S_i)\} \nsubseteq D \\ & \wedge \{S_j | \tau_i + \Delta_i \leq \tau_j + \Delta_j \leq \tau_k + \Delta_k\} = \emptyset\}. \end{aligned} \tag{3.38}$$

Steppers in the set $S_{startable}$ are started by the scheduler, and the set is joined with $S_{running}$. This procedure is repeated until $S_{startable}$ becomes empty. When all Steppers in $S_{running}$ finish executions, the global time is updated, $S_{running}$ is flushed, and proceeds to the procedure (9). Shared-memory multi-threading is suitable to this parallelization scheme, and an attempt of implementation is ongoing with Satya Arjunan (Arjunan *et al.*,, 2003).

3.6 Conclusion

Due to the non-linear nature of the subsystems and the intimate couplings between them, simulation is crucial for cell biology research. In the past, however, it has been the norm to adopt different simulation algorithms for different sub-systems of the cell. This had made it difficult to combine the sub-cellular models and in many cases limited the applications of simulation to single-scale, sub-cellular problems.

In this chapter, we have provided a modular meta-algorithm with a discrete event scheduler that can incorporate any type of time-driven simulation algorithm. It was shown that this meta-algorithm can efficiently drive simulation models with different simulation algorithms with little intrusive modification to the algorithms themselves. Only a few additional methods to handle communications between computational modules are required.

The best algorithm for a specific model is determined by the nature of the target system. Our meta-algorithm provides a solution for situations in which it is necessary to simulate processes concurrently across multiple scales of time, space or concentration (Rao *et al.*,, 2002).

Chapter 4

E-Cell System

In the previous sections, an implication of the 'ontological complexity' of the cell that leads to need for a sophisticated software platform has been discussed. Unlike some of conventional approaches to physical and biochemical systems, such complicated nature of the cell as a target system is inevitably reflected by the design and implementation of the software (Takahashi *et al.,*, 2002).

In this chapter, we discuss design and implementation of E-Cell System, a software environment for modeling, simulation and analysis of the cell. We put an emphasis on architecture and implementation of E-Cell Simulation Environment, a part of the E-Cell System of which core is an object-oriented simulation engine that implements the cell simulation 'meta-algorithm' discussed in the previous chapter.

4.1 Architecture of E-Cell System

E-Cell System is a suite of software that provides an environment for modeling, simulation and analysis of large-scale complex systems such as biological cells. The following

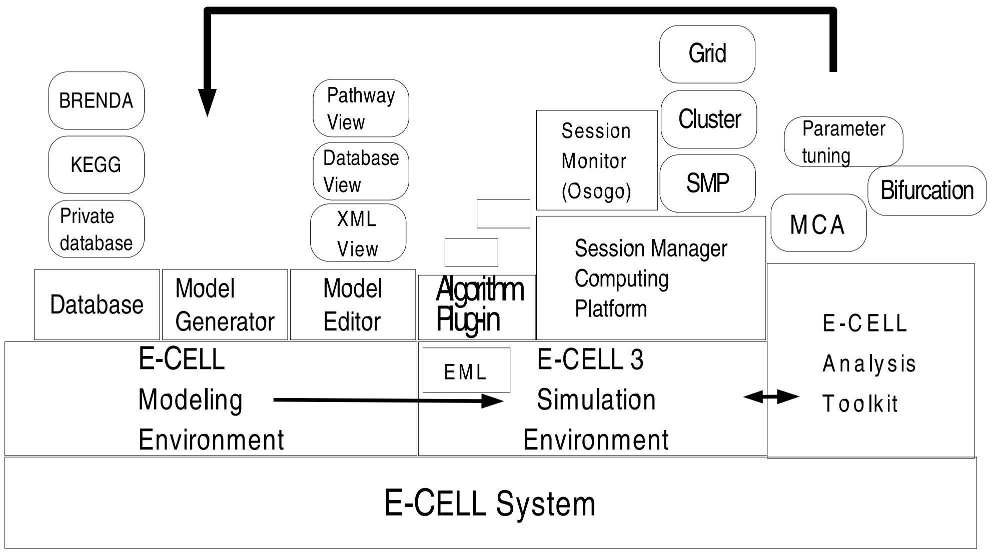


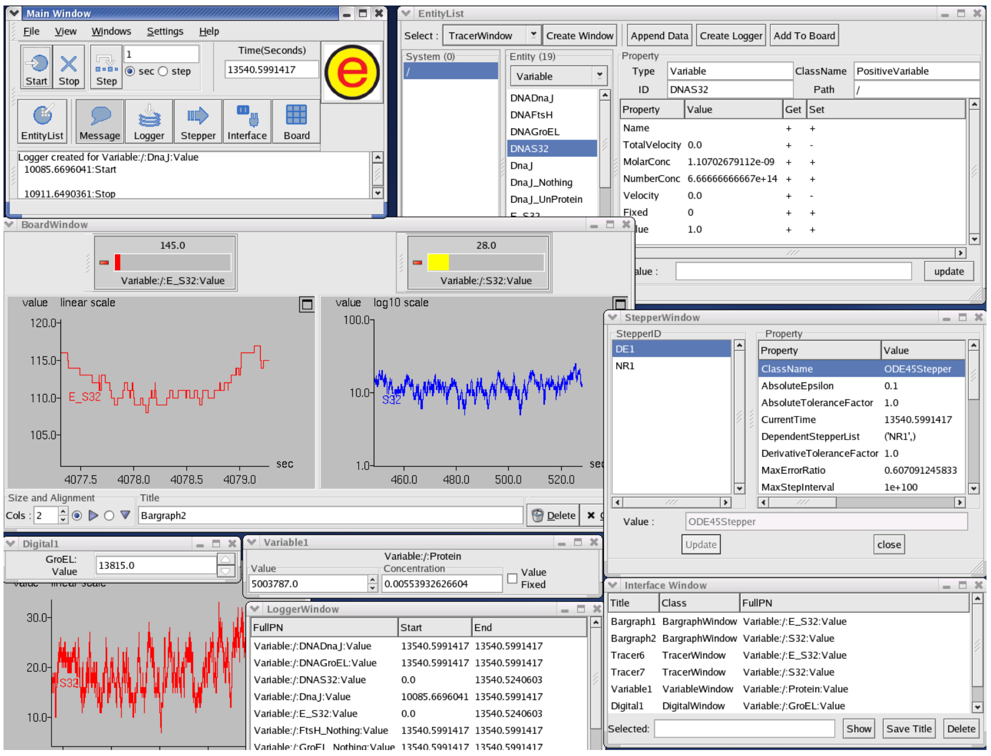
Figure 4.1: Overview of E-Cell System Development

three parts constitutes E-Cell System Version 3 (Figure 4.1): (1) E-Cell Simulation Environment (E-Cell SE), which runs the simulation, (2) E-Cell Modeling Environment (E-Cell ME), that helps users to conduct a collaborative modeling of large-scale cell models, and (3) E-Cell Analysis Toolkit, a library of mathematical analysis scripts for E-Cell SE. This paper mainly discusses E-Cell SE.

4.1.1 The E-Cell Simulation Environment

E-Cell Simulation Environment, the core of E-Cell System, provides the rest of the system an ability of trajectory realization. E-Cell SE, in turn, is consisting of two parts; (1) the kernel and (2) frontend components. The kernel part of E-Cell SE is the libecs library written in ISO C++, and has a layered architecture (Figure 4.3). Libecs class library defines object classes for fundamental functionalities of the system, such as an implementation of the 'meta-algorithm' and data-logging. Giving a facade to the set of object classes is libemc library which is also written in C++. This library defines a C++ API of the system for frontend scripting. Frontend components of

E-Cell SE are written in Python language, and utilizes a Python binding of libemc, pyecs module. Upon the pyecs layer, various peripheral software components such as graphical and script-based frontends, simulation session managers, analysis modules, data management modules, cell model editors/processors and other user-level scripts are developed. Figure 4.2 shows a GUI simulation session monitor frontend built upon pyecs API.



Step: 1 [✓ sec] step

Time(Seconds): 13540.5591417

Logger created for Variable:/:DnaJ:Value
 10085.6696041:Start

 10911.6490361:Stop

Select : TracerWindow — Create Window

System (0): /

Entity (19): Variable

DNADnaJ
DNAFtsH
DNAGroEL
DNAS32
DnaJ
DnaJ_Nothing
DnaJ_UnProtein

Property

Type	Variable	ClassName	PositiveVariable
ID	DNAS32	Path	/

Property	Value	Get	Set
Name		+	+
TotalVelocity	0.0	+	-
MolarConc	1.10702679112e-09	+	+
NumberConc	6.66666666667e+14	+	+
Velocity	0.0	+	-
Fixed	0	+	+
lue	1.0	+	+

145.0 — Variable:/:E_S32:Value

28.0 — Variable:/:S32:Value

Size and Alignment: Cols : 2 — Title: Bargraph2

StepperID: DE1, NR1

Property	Value
ClassName	ODE45Stepper
AbsoluteEpsilon	0.1
AbsoluteToleranceFactor	1.0
CurrentTime	13540.5991417
DependentStepperList	('NR1',)
DerivativeToleranceFactor	1.0
MaxErrorRatio	0.607091245833
MaxStepInterval	1e+100

Value : ODE45Stepper

GroEL: Value 13815.0

Variable:/:Protein — Value: 5003787.0 — Concentration: 0.00553932626604 — Value Fixed: ☐

FullPN	Start	End
Variable:/:DNADnaJ:Value	13540.5991417	13540.5991417
Variable:/:DNAGroEL:Value	13540.5991417	13540.5991417
Variable:/:DNAS32:Value	0.0	13540.5240603
Variable:/:DnaJ:Value	10085.6696041	13540.5991417
Variable:/:E_S32:Value	0.0	13540.5240603
Variable:/:FtsH_Nothing:Value	13540.5991417	13540.5991417
Variable:/:GroEL_Nothing:Value	13540.5991417	13540.5991417

Title	Class	FullPN
Bargraph1	BargraphWindow	Variable:/:E_S32:Value
Bargraph2	BargraphWindow	Variable:/:S32:Value
Tracer6	TracerWindow	Variable:/:E_S32:Value
Tracer7	TracerWindow	Variable:/:S32:Value
Variable1	VariableWindow	Variable:/:Protein:Value
Digital1	DigitalWindow	Variable:/:GroEL:Value

Figure 4.2: E-Cell Simulation Session Monitor GUI

Libecs, libemc and pyecs layers are written in C++ language, and frontend scripts uses Python language. C++ has been chosen for components for which runtime efficiency, coding strictness and flexibility are critical. Python is an ideal language for the other user-side components, where productivity and readability are demanded. Python language is a popular, interpreted, object-oriented, and very high-level programming language which shows prominent productivity even with non-professional programmers.

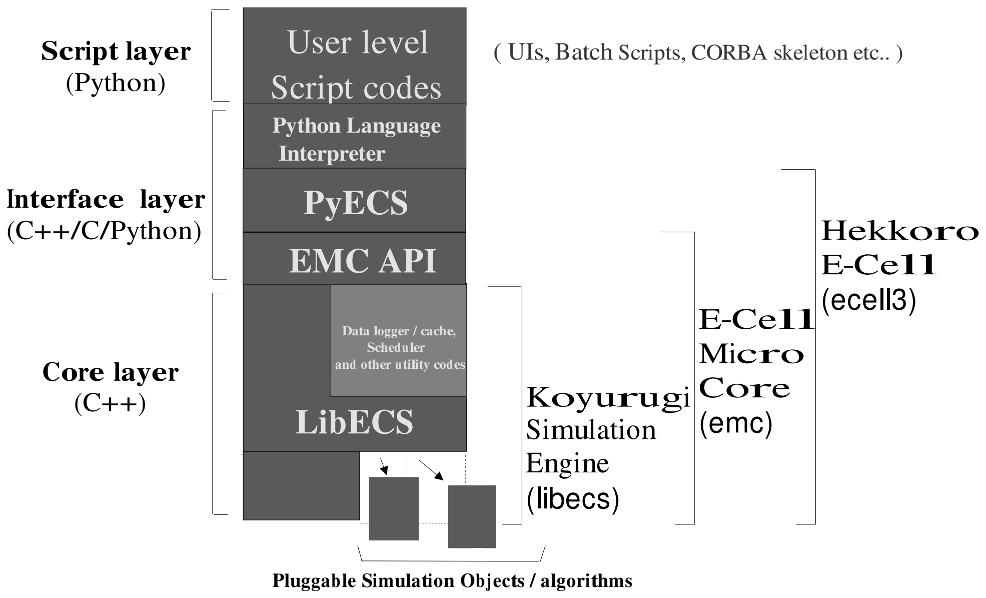


Figure 4.3: Three layer architecture of E-Cell kernel

Owing to its modular design, any type of user interface can be developed as user-level Python scripts. Preferred styles of user interfaces vary according to many factors, such as the stage of the simulation project, purpose of the simulation, the analysis method currently in use, available computation resources, and the user's preferences. Two main types of user interfaces are GUI (Graphical User Interface) and CUI (Command-line User Interface or Character-based User Interface). GUI is well suited for interactive model editing, simulation running and analysis if many trial and error is needed. CUI is suitable for automation of very long sessions and parametric computings. E-Cell SE supports both styles of the operation modes.

E-Cell SE supports two levels of parallel computation. At a lower level, a simulation session of a multi-stepper model can be parallelized on shared-memory multi processor computers (Arjunan *et al.,*, 2003). Secondly, simulation experiments that involves many runs of simulation sessions, such as parameter tuning, metabolic control and bifurcation analysis, can also be parallelized on cluster and grid environments. This distributed computing is supported by *E-Cell SessionManager* module, with which users can write

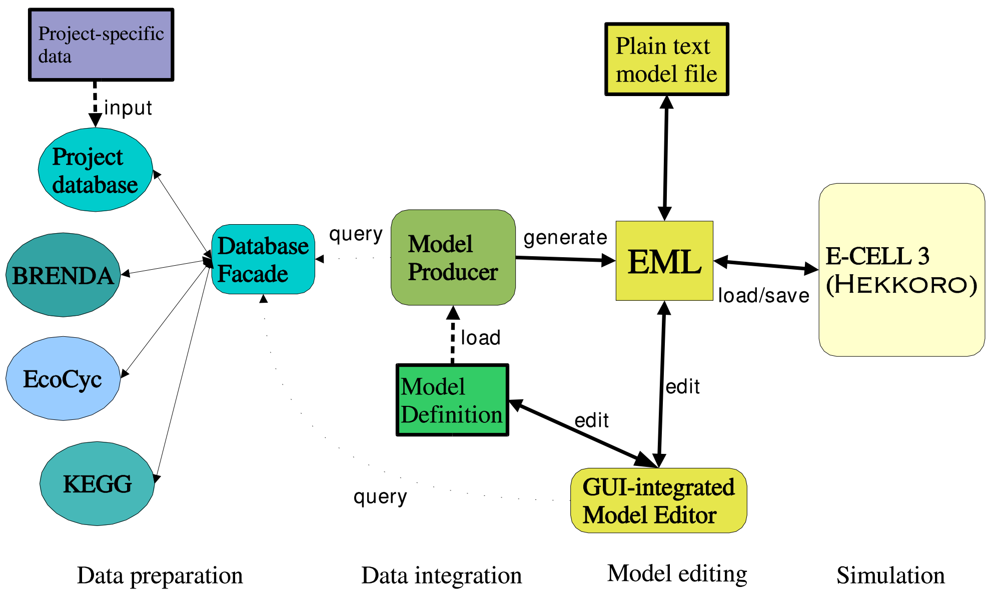


Figure 4.4: The E-Cell Modeling Environment

scripts to create many simulation sessions, dispatch them to remote computers, retrieve the simulation results and process the result data.

4.1.2 The E-Cell Modeling Environment

In whole and multiple cell scale simulation projects, writing models by hand is unlikely to work. A database-driven software environment for integrating biological data and creating simulation models is necessary. Although this is not part of the work that this thesis is mainly discussing, here we briefly describe the overall design of E-Cell Modeling Environment, or E-Cell ME, which is a software environment for large scale cell biology modeling projects.

We envision a four phase modeling scheme, each being supported by a seamlessly interoperating software suite (Figure 4.4).

- Data preparation phase

In this phase, biological data needed to develop simulation models is stored in public and private databases. The private database is used both for (1) collecting supplementary data not available on the public databases but can be retrieved from other sources such as literatures, and (2) collaboration between the modeling project and wet laboratories who supply the data.

The environment has a software component called the 'database facade' to provide other portions of the system a unified means of retrieving data from multiple databases.

- Data integration phase

  Automatic model generator software (the 'model producer') creates templates of the models in XML-based E-Cell Model description Language (EML) from the 'model definition' files written by users. A model definition file specifies information necessary to automate the model generation, such as a targeted organism, pathways, modeling conventions used, and simulation approaches (*e.g.* deterministic or stochastic) to be applied.

- Model editing phase

  EML models are edited by users with a multi-view, database-integrated model editor software. It also provides a plain-text modeling file format that can be edited by text editors.

- Simulation phase

  Constructed simulation models in EML are read and run on E-Cell SE. Simulation results are fed back to the simulation model and the databases.

E-Cell ME is currently under development.

4.2 Object-oriented implementation of the meta-algorithm

The multi-algorithm computation framework called the 'meta-algorithm' introduced in the previous chapter has been implemented in libecs core library of E-Cell System Version 3.

4.2.1 Fundamental classes

The 'meta-algorithm' has three basic object classes, *Variable*, *Process*, and *Stepper*. A set of Variable objects represent a state of the model system. Values of Variable objects are changed by Process objects, which encapsulate continuous and discrete transition functions. Timings of firings of Process objects are determined by Stepper objects. In addition to these three base classes, this implementation adds the *System* object class for structured modeling of large-scale systems. Figure 4.5 shows relationships between these fundamental classes. Variable, Process and System objects has a common base class, *Entity*. Entity and Stepper classes are two fundamental base class, and share a common root class called *PropertiedClass*. A System object contain Variables, Processes, and also other Systems (or sub-Systems), and represent physical and/or logical compartments.

Four basic base classes of Stepper are available: *DifferentialStepper*, *DiscreteTimeStepper*, *DiscreteEventStepper*, and *PassiveStepper* (Figure 4.6, 4.7). DifferentialStepper is a continuous Stepper, and all the other Steppers are discrete. DifferentialStepper is used to drive Processes that encodes differential equations. DiscreteEventStepper is used for discrete-event simulation algorithms such as the Gillespie methods. Discrete-TimeStepper is used for discrete-time simulation models, and does not change its step size during simulation ignoring incoming interruptions. Although it is not included in Zeigler's classification of simulation algorithms, another base class of Stepper named

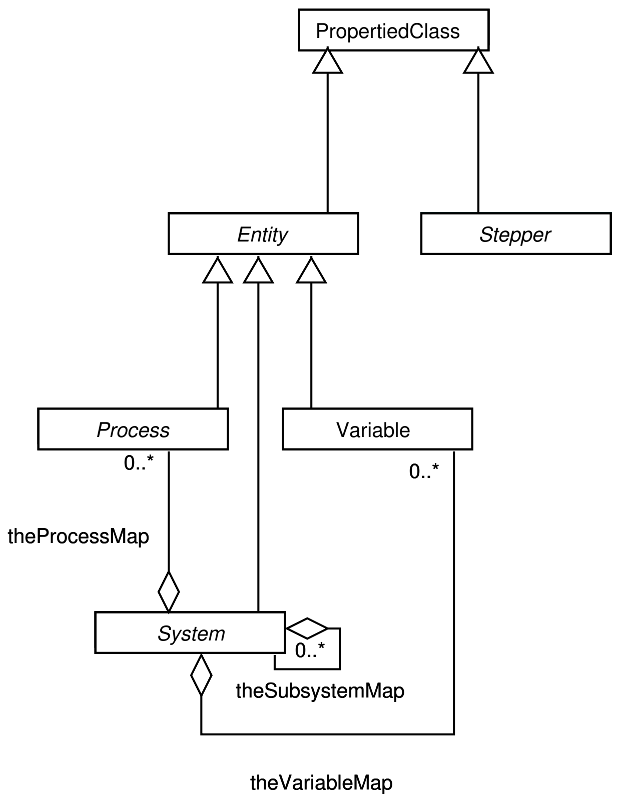


Figure 4.5: Fundamental class diagram

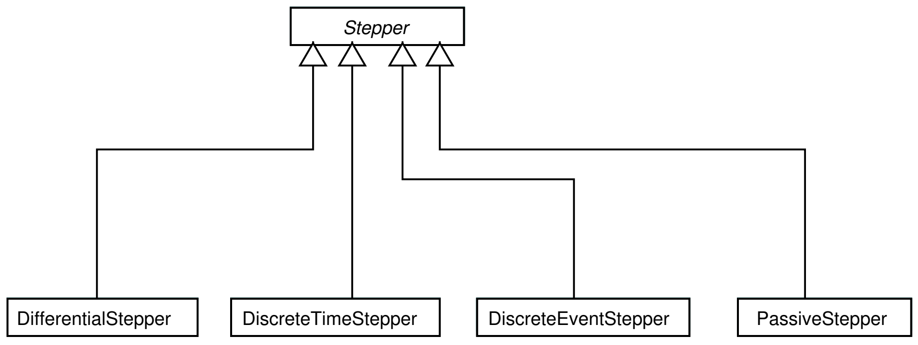


Figure 4.6: Stepper base classes

PassiveStepper is implemented. It never steps spontaneously (*i.e.* $\Delta t = \infty$), and fires its Process objects upon interruptions from other Steppers. It is useful for time-less computation procedures used in simulations, such as the calculation of flux-distribution coupled with a dynamic part.

4.2.2 Overall structure of the simulator core

A frontend module must instantiate a *Model* object when starting a simulation session. A Model is composed of three major parts, the *Scheduler*, the *root System* and the *LoggerBroker* member objects (Figure 4.8). The Scheduler has a set of all Steppers in the simulation model, and when the simulation is started, calls Steppers to proceed time. The root System is a plain System object that works as the root of all other Entity objects, and is automatically created when the simulator is started up. The LoggerBroker creates and maintains *Logger* objects. A Logger is created upon user's request, and stores a time-series of a property of an Entity object.

The Model object also owns class methods for basic operations to prepare and run simulations. For example, its step() method conduct a step of simulation calling the next Stepper in the Scheduler, and advance time. getCurrentTime() method returns

Class name	dt	Time scale	Type	Interruption	Interrupted
Differential	var.	== dt	continuous	always	recalculate interpolants
DiscreteTime	const.	== dt	discrete	always	ignore
DiscreteEvent	var.	var.	discrete	always	reset
Passive	inf.	--	discrete	never	step

Figure 4.7: Basic behaviors of Stepper base classes defined in E-CELL SE
dt: basic step sizing policy. ('var.' means variable step size, 'const.' means that the Stepper does not change step size by itself. 'inf.' indicates that the Stepper does not step spontaneously.) *Time scale*: the procedures to determine the time scale of the Stepper (θ in the algorithm chapter). *Type*: the type of the Stepper, either continuous or discrete. *Interruption*: the policy used to dispatch interruptions to dependent Steppers. ('always' means that it always dispatchs interruptions to all the dependent Steppers, and 'never' is that it does not make interruptions at all.) *Interrupted*: the policy used when the Stepper is interrupted. ('recalculate interpolants' means that it recalculates input parameters to its Interpolant objects. 'ignore' simply ignores incoming interruptions. 'reset' means that the Stepper resets the state of the system. 'step' indicates that it uses a procedure equivalent to usual simulation steps (*i.e.* calls fire() of Processes).)

the current time. getEntity(), getStepper() and getSystem() methods finds and returns a borrowed pointer to an Entity, a Stepper, or a System object getting an identifier. createStepper() and createEntity() methods create new Stepper or Entity objects in the simulation model.

4.2.3 Implementation of transition functions

The meta-algorithm defines the continuous transition function

$$F : (\check{R}^{*}, \Delta\tau, \tau) \mapsto (\delta\hat{R}) \tag{4.1}$$

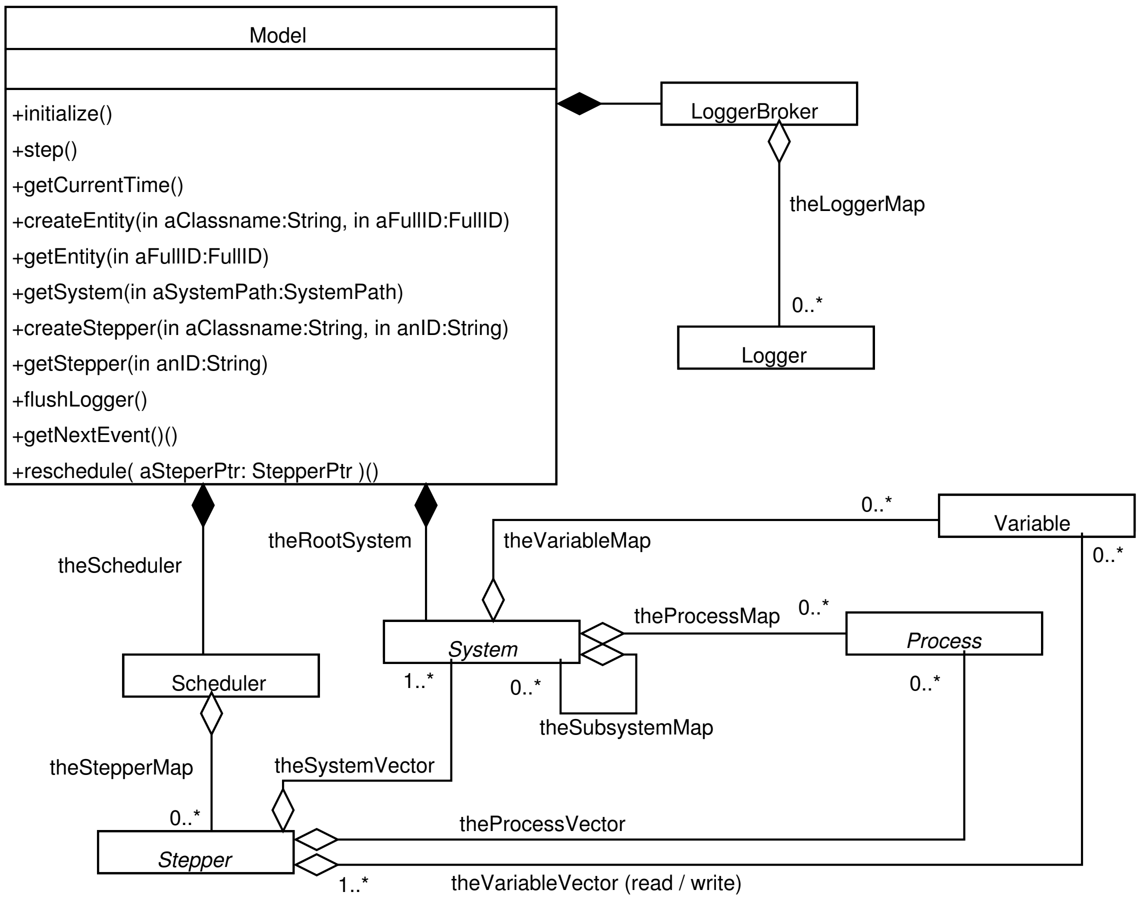


Figure 4.8: Simulator overview class diagram

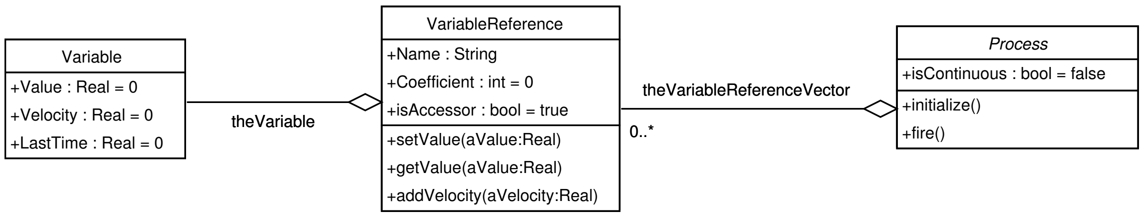


Figure 4.9: Classes implementing transition functions

and the discrete transition function

$$F : (\check{R}^{*}, \Delta\tau, \tau) \mapsto (\hat{R}^{*}). \tag{4.2}$$

In libecs, implementation of this framework of transition functions spreads over three object classes, Process, Variable, and VariableReference.

Figure 4.9 depicts how these three classes of objects are related. A Process object, $P_{i,j}$, has two main methods, Process::fire() and Process::initialize(). The Process::fire() method corresponds to the transition function $F_{i,j}$ of the Process $P_{i,j}$. The Process::initialize() method is called to initialize the Process object when the simulator is starting up or the structure of the model is changed. The Process object also has isContinuous boolean flag, which indicates if this Process's transition function (the fire() method) is continuous (true) or discrete (false). Variable references $R_{i,j}$ of the Process $P_{i,j}$ are stored as a vector of VariableReference objects (*theVariableReferenceVector* in the figure). Each VariableReference object has a name, a coefficient, an isAccessor flag, and a pointer to the Variable that this reference points to. A VariableReference ρ is a mutator ($\rho \in \hat{R}_{i,j}$) if the coefficient is not zero, and it is an accessor ($\rho \in \check{R}_{i,j}$) if the isAccessor flag is true (which is the default). Transition functions (fire() methods) of Process objects manipulate Variables using the VariableReference objects. When a VariableReference is an accessor, the Process is allowed to use its getValue() method

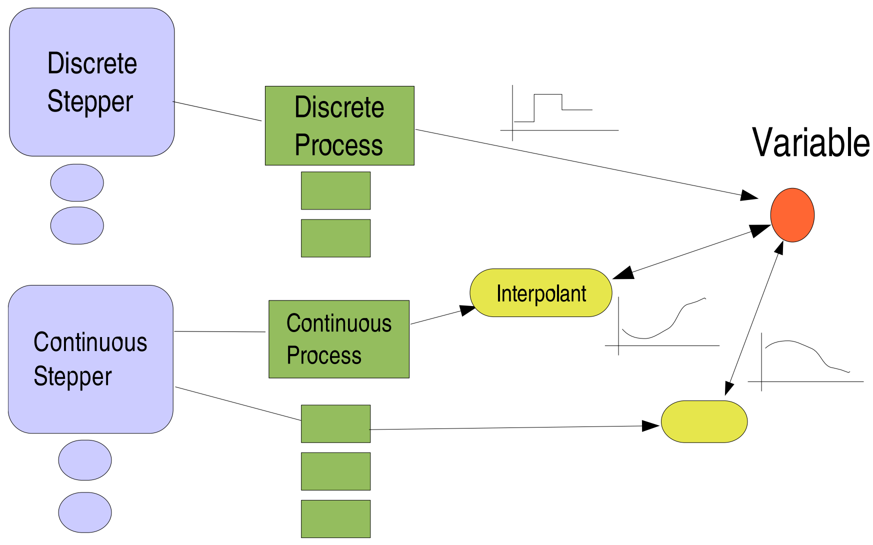


Figure 4.10: Relations between discrete/continuous Processes and a Variable. As shown in this diagram, a discrete Process belongs to a discrete Stepper, and can alter the value of the Variable directly. On the other hand, a continuous Process cannot directly access the Variable, and uses an Interpolant object to give it a continuous output. The continuous Stepper contains the continuous Process.

to get the *Value property* of the Variable. When the VariableReference is a mutator, and the Process is discrete, it can change the value of the Variable directly by calling setValue() method of the VariableReference. If the Process is continuous, it may call addVelocity() method of the VariableReference to add some value to the *Velocity property* of the Variable (which corresponds to $\delta\rho$ of the meta-algorithm framework). This Velocity is used by the Process's continuous Stepper to compute input parameters to its interpolants. Figure 4.10 is a schematic summary of relations between discrete/continuous Processes and a Variable.

4.2.4 Implementation of the integration algorithm

Object classes related to the implementation of the integration algorithm is shown in Figure 4.11. Each continuous Stepper class has its own Interpolant class, and registers an instance of the Interpolant to each of Variables in its mutator variable reference list ($\hat{R}_i^*$) at model initialization.

In procedure 5 (integrate phase) of the driver algorithm, the Stepper S_i calls integrate() method of all the Variables R_i^* with the current local time τ_i (*i.e.* integrate(τ_i)). In integrate() method, the Variable object corresponds to the variable x_j calls getDifference() method of each Interpolant object in its *InterpolantVector* with the current local time τ_i and the time interval Δt_j to calculate the interpolant difference $\Phi_{k,j}$ according to 3.18, where k is the index of the Stepper of the interpolant. Δt_j is calculated by the Variable object as $\tau_i - t_j$. t_j is stored in the *LastTime*LastTime property of the Variable object. The Variable then calculates the interpolant sum 3.17 by simply summing up the calculated interpolant differences. Lastly, the Variable updates the last updated time t_j, and the integration procedure finishes.

4.2.5 Simulation procedure

Figure 4.12 displays all the main object classes involved in simulation.

The procedure to conduct a step of simulation is as shown in Figure 4.13. The Model object calls step() method of the Scheduler. The Scheduler picks the next Stepper from its schedule priority queue, *theScheduleQueue* (procedure (3) of the Driver algorithm). The Scheduler then update the global time and the local time of the Stepper by calling setCurrentTime() method (procedure (4)). Next, the Scheduler calls the Stepper in four phases: *integrate*, *step*, *dispatchInterruptions*, and *log*. In the integrate phase, the Stepper calls integrate() method of all Variables related to the Stepper. As explained in the 'implementation of the integration algorithm' section above, each Variable uses its Interpolants' getDifference() method to update its value at the current time, and each Interpolant object uses interpolation parameters given by Steppers connected to

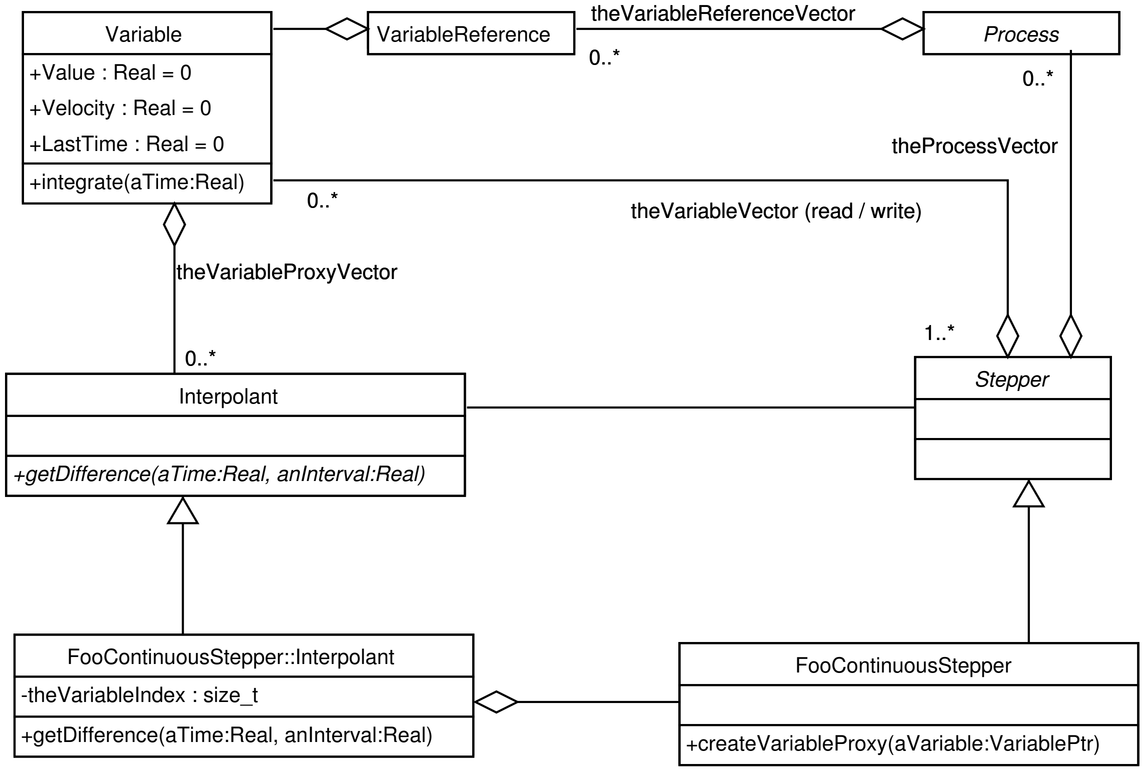

Variable
+Value : Real = 0
+Velocity : Real = 0
+LastTime : Real = 0
+integrate(aTime:Real)
VariableReference
theVariableReferenceVector
0..*
Process
0..*
theProcessVector
0..*
theVariableVector (read / write)
theVariableProxyVector
1..*
0..*
Interpolant
+getDifference(aTime:Real, anInterval:Real)
Stepper
FooContinuousStepper::Interpolant
-theVariableIndex : size_t
+getDifference(aTime:Real, anInterval:Real)
FooContinuousStepper
+createVariableProxy(aVariable:VariablePtr)

Figure 4.11: Integration mechanism class diagram

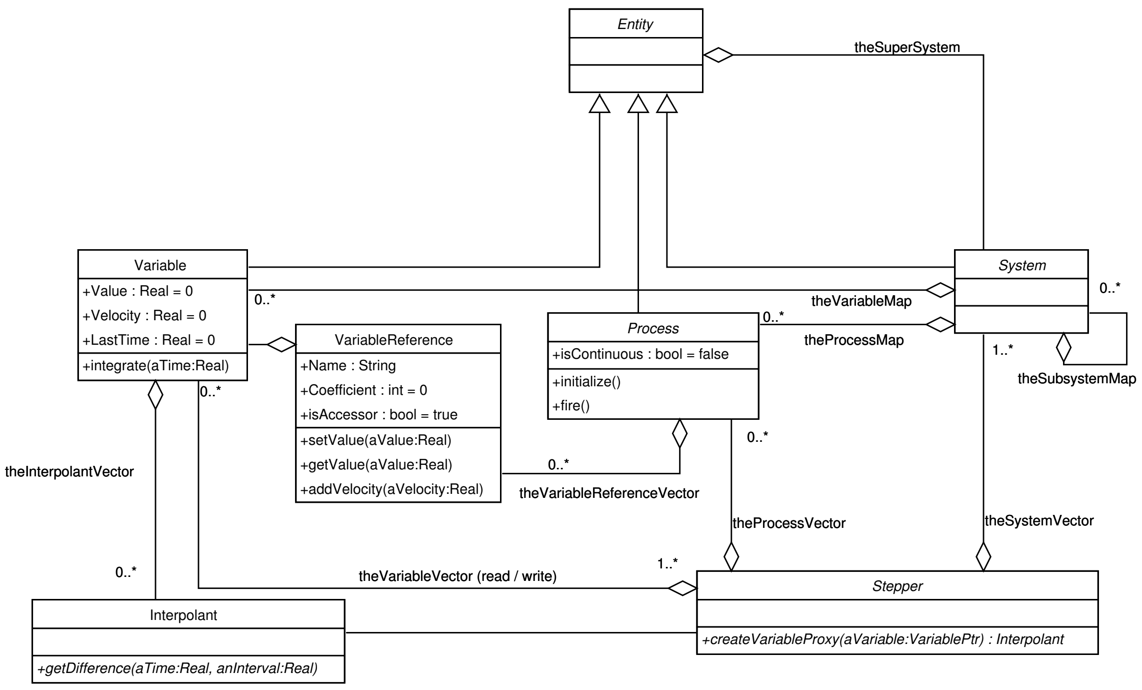


Figure 4.12: A diagram of classes involved in simulation

this Variable. In step phase, the Stepper calls fire() method of each Process object that the Stepper holds. The Stepper may also change the step size and other parameters (see the description of the driver algorithm in the previous chapter). Process objects execute their transition functions, which may alter values or give velocities to Variables via its VariableReferences. This procedure has been described in 'Implementation of transition functions' section. In log phase, log() methods of Loggers attached to Entity objects in the Stepper are called. The following 'Data logging mechanism' section will describe this procedure in detail. In the dispatchInterruptions phase, the Stepper calls interrupt() method of each Stepper dependent on it. Finally, the Scheduler reschedules the stepping Stepper on the schedule priority queue.

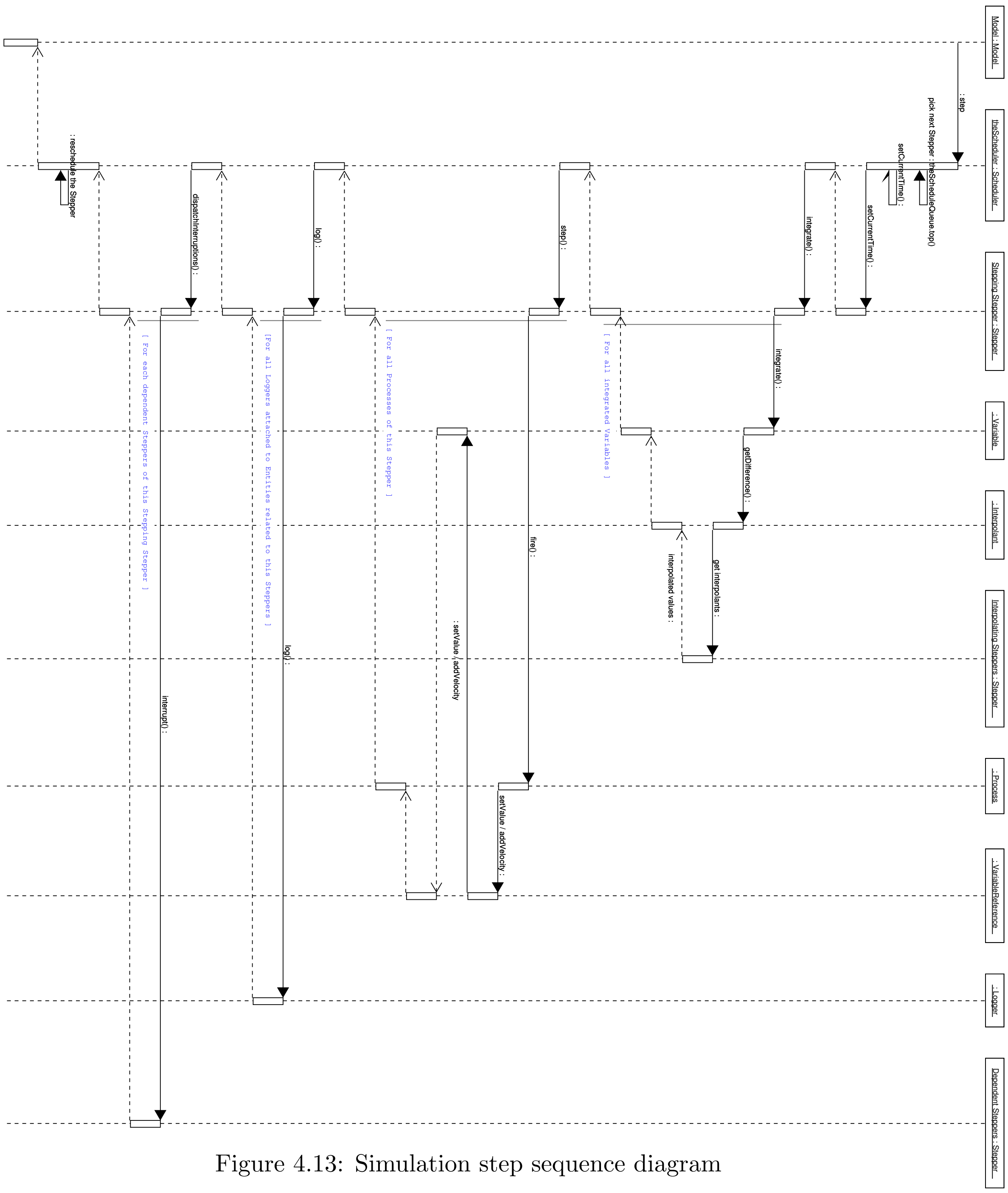

Model : Model
theScheduler : Scheduler
Stepping Stepper : Stepper
: Variable
: Interpolant
Interpolating Steppers : Stepper
: Process
: VariableReference
: Logger
Dependent Steppers : Stepper
: step
pick next Stepper : theScheduleQueue.top()
setCurrentTime() :
setCurrentTime() :
integrate() :
step() :
log() :
dispatchInterruptions() :
: reschedule the Stepper
[For each dependent Steppers of this Stepping Stepper]
[For all Loggers attached to Entities related to this Steppers]
[For all Processes of this Stepper]
[For all integrated Variables]
integrate() :
getDifference() :
get interpolants :
interpolated values :
fire() :
: setValue / addVelocity
setValue / addVelocity :
log() :
interrupt() :

Figure 4.13: Simulation step sequence diagram

4.2.6 Dynamic method invocation using PropertySlot

PropertiedClass gives objects an ability of dynamic method invocation (Figure 4.14) via object property. Each object property has a pair of methods to *set* and *get* the property. The simplest behaviors of these methods are to simply store and return the value of a member variable. In E-Cell SE, a property has a type of one of Real (floating-point real number), Integer, String, or list (called *PolymorphVector* in the implementation). A PropertiedClass (*e.g.* a subclass of Entity or Stepper) object has a static member object, *thePropertyInterface*, which associates *PropertySlot* objects with property names. PropertySlot is a template base class of *ConcretePropertySlot* class, which holds pointers to 'get method' and 'set method'. There are two ways for a client code to access a property of a PropertiedClass. The first way is to use getProperty() and setProperty() member function of the PropertiedClass itself. These methods take and return a *Polymorph* object, which is a type of object that can be created from and converted to any of the property value types. Although simple and easy to use, this property invocation scheme has some performance overhead (the property table lookup and construction of a Polymorph object). The second way is to create a *PropertySlot-Proxy* object first, and make accesses to the property via the proxy object. In this case, the overhead of the creation of the PropertySlotProxy object is roughly the same as the first way of property access, but from the second time it becomes considerably faster. PropertySlotProxy is used in a form of its template child class, *ConcretePropertySlot-Proxy<T>*, where T is the sub-class of the PropertiedClass to which the PropertySlot belongs.

This mechanism of dynamically bound property access is used mainly in communication between frontend modules and objects in the simulation model (Entity and Stepper). For example, frontend software obtains or changes values of parameters of objects during simulations using this PropertySlot facilities. In simulation, direct invocation

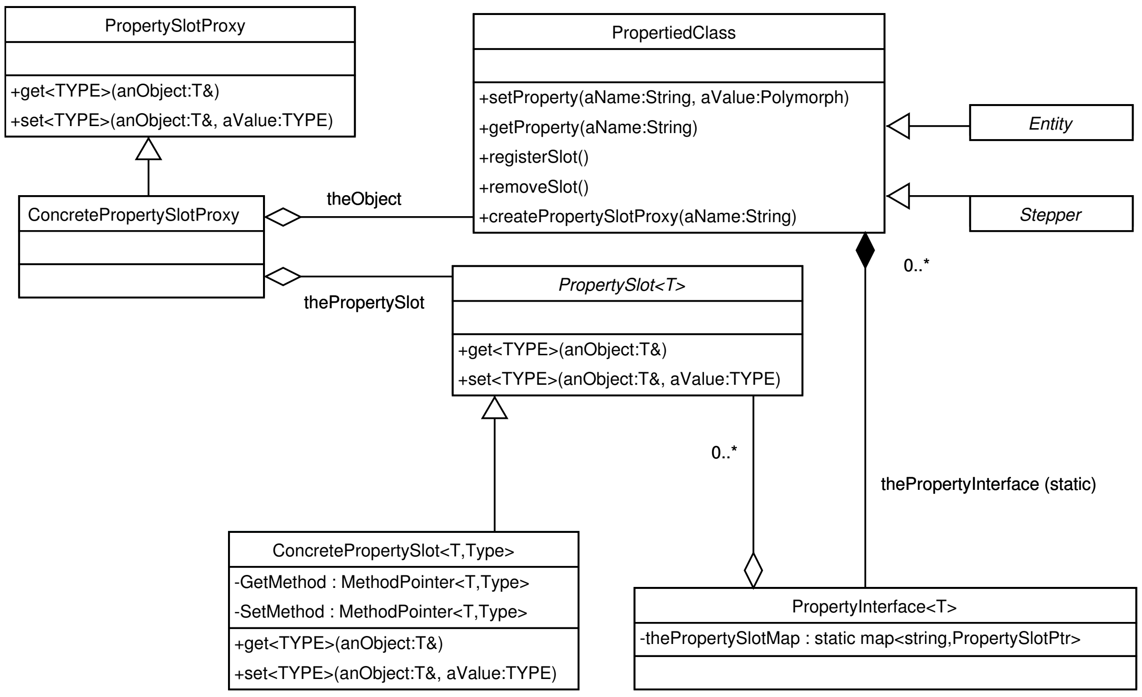

PropertySlotProxy
+get<TYPE>(anObject:T&)
+set<TYPE>(anObject:T&, aValue:TYPE)
PropertiedClass
+setProperty(aName:String, aValue:Polymorph)
+getProperty(aName:String)
+registerSlot()
+removeSlot()
+createPropertySlotProxy(aName:String)
Entity
Stepper
ConcretePropertySlotProxy
theObject
thePropertySlot
PropertySlot<T>
+get<TYPE>(anObject:T&)
+set<TYPE>(anObject:T&, aValue:TYPE)
0..*
0..*
thePropertyInterface (static)
ConcretePropertySlot<T,Type>
-GetMethod : MethodPointer<T,Type>
-SetMethod : MethodPointer<T,Type>
+get<TYPE>(anObject:T&)
+set<TYPE>(anObject:T&, aValue:TYPE)
PropertyInterface<T>
-thePropertySlotMap : static map<string,PropertySlotPtr>

Figure 4.14: PropertiedClass class diagram

of C++ methods are also used to maximize the performance.

4.2.7 Data logging mechanism

In E-Cell SE Version 3, Logger objects record the simulation results. This architecture, where neither the Entity objects nor the frontends store the entire data, has the following advantages.

- It keeps the structure of the system simple.
- There is no need to hold overlapping data when there are more than two view objects in the frontend.
- Any frontend software can use a common data logging system via a simple API,

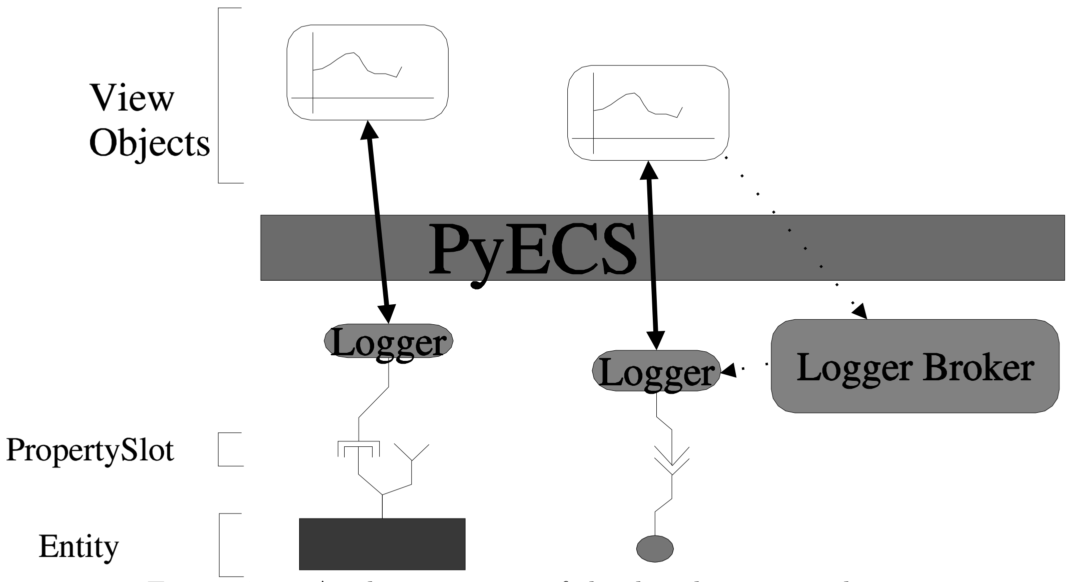


Figure 4.15: A schematic view of the data logging mechanism

instead of accessing implementation details of libecs. Use of virtual memory mechanism enables management of large amounts of data without limitation from available RAM.

Figure 4.15 shows a schematic view of the data logging mechanism. Accepting a request from the frontend, LoggerBroker either creates and returns a new Logger object connected with the requested property slot of the requested Entity, or simply gives a reference to an existing Logger if the requested property in the Entity has already being recorded by the Logger object.

Figure 4.16 is a UML class diagram depicting the classes related to data logging. The Logger class has a PhysicalLogger object, which implements a physical data storage mechanism. In this case, it is implemented using the *vvector* class, which is an implementation of a virtual memory system. A data source is connected to the Logger via a *LoggerAdapter* object. Each subclass of LoggerAdapter is prepared for a type of data source. When the original data source is a PropertiedClass object, such as an Entity

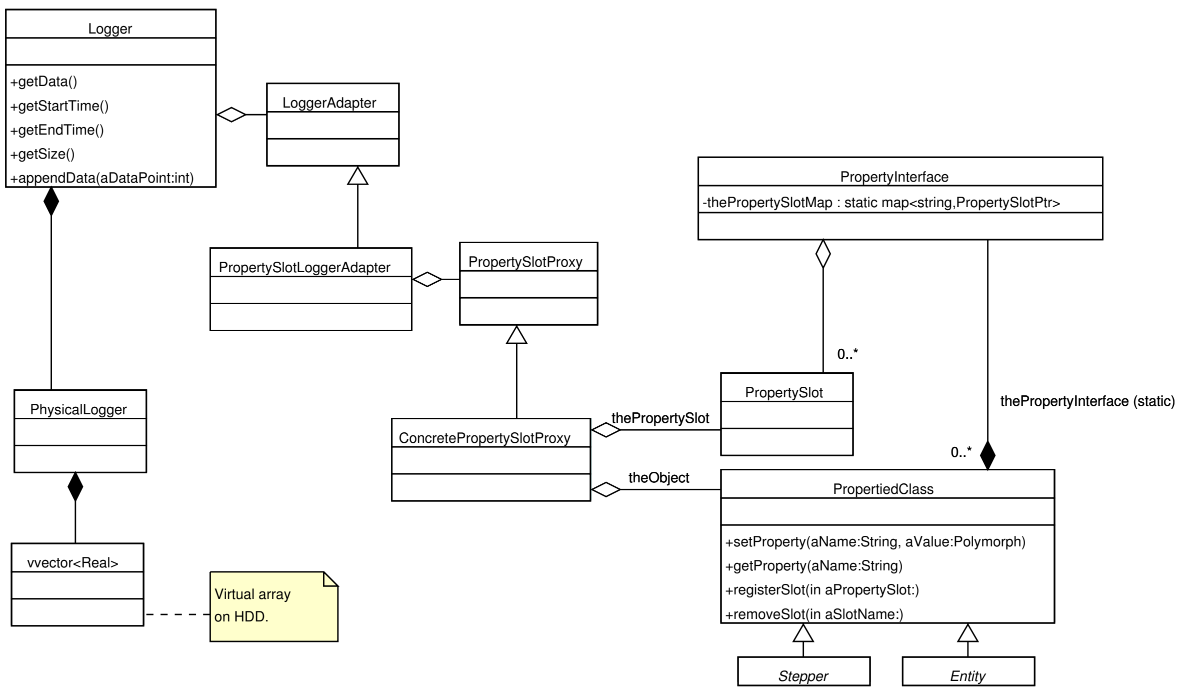

Logger
+getData()
+getStartTime()
+getEndTime()
+getSize()
+appendData(aDataPoint:int)
LoggerAdapter
PropertySlotLoggerAdapter
PropertySlotProxy
PropertyInterface
-thePropertySlotMap : static map<string,PropertySlotPtr>
0..*
PropertySlot
thePropertyInterface (static)
0..*
PhysicalLogger
thePropertySlot
ConcretePropertySlotProxy
theObject
PropertiedClass
+setProperty(aName:String, aValue:Polymorph)
+getProperty(aName:String)
+registerSlot(in aPropertySlot:)
+removeSlot(in aSlotName:)
vvector<Real>
Virtual array
on HDD.
Stepper
Entity

Figure 4.16: Logging class diagram

and a Stepper, the data is provided via a proxy object, ConcretePropertySlotProxy object, and a subclass of LoggerAdapter, *PropertySlotLoggerAdapter*, which is designed to get the data from a PropertySlotProxy object is necessary. Each PropertiedClass has a static member PropertyInterface object that holds a table of PropertySlot objects. Therefore, to access a PropertySlot of a PropertiedClass, the ConcretePropertySlot-Proxy object must have two pointer member variables, *theObject* member variable that points to the PropertiedClass object, and *thePropertySlot* member variable which specifies what PropertySlot the proxy is attached to.

When a Stepper steps in the simulation, it calls Logger objects attached to its Entities (Variables, Processes and Systems). Figure 4.17 shows the sequence of data logging. The Stepper calls each Logger's log() method, and it eventually retrieves the value of the property via PropertySlotLoggerAdapter and PropertySlotProxy objects. Then it

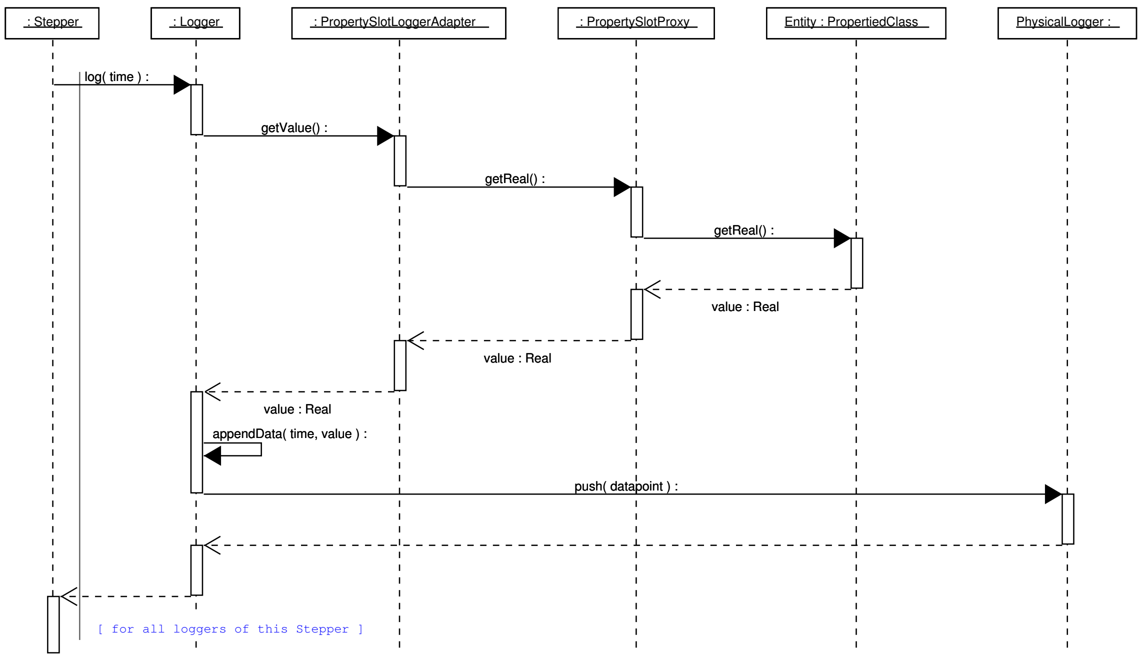


Figure 4.17: Logging sequence diagram

pushes the value into its PhysicalLogger object that implements the virtual-memory data storage (vvector class in the Figure 4.16).

At first sight, this design, which involves more than ten classes, might seems to be overly complicated considering the simple task that Logger is designed for. However, the key idea is to implement the whole data logging mechanism in a generic way so that the Logger class can be used with little modification when the system's design is changed or new classes of objects other than Entity and Stepper are made targets of the data logging. Considering this gain in flexibility, the impact on performance is negligible compared to the amount of operations that must be done to push a data tuple into the virtual memory; at most two or three extra pointer de-referenciations per a logging event.

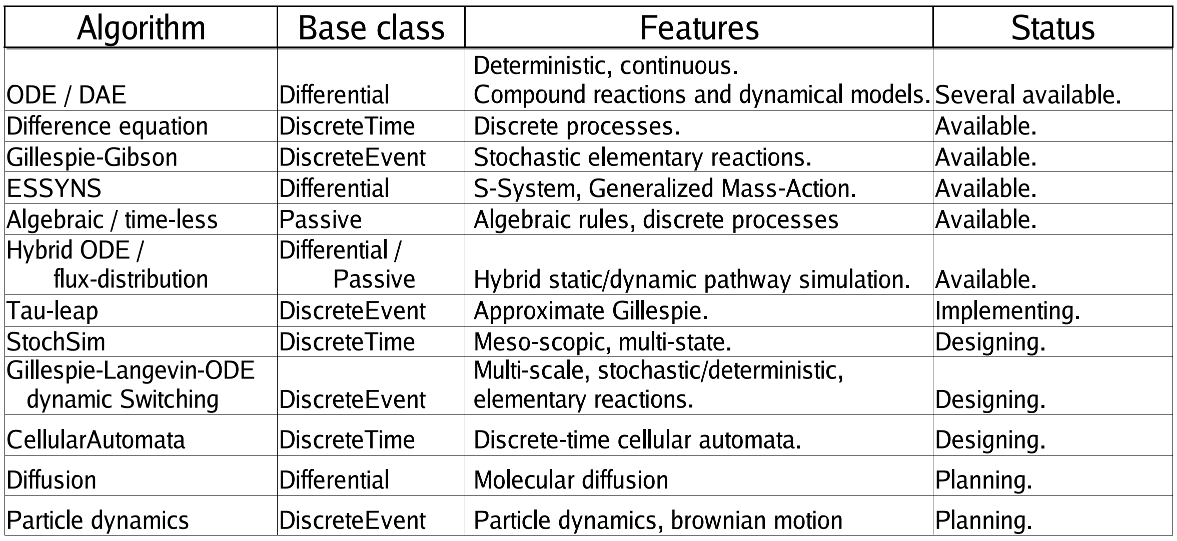
4.2.8 Implemented simulation algorithms

Virtually any time-driven simulation algorithms can be implemented as plug-ins of E-Cell SE. Current version of the software is shipped with modules for simulation algorithms including: (1) several variations of ODE and DAE solvers, (2) Gillespie-Gibson stochastic simulation algorithm (Gibson and Bruck, 2000), (3) ESSYNS GMA/S-System power-law canonical differential equations solver (Irvine and Savageau, 1990), (4) discrete-time simulator, (5) a hybrid dynamic/static pathway simulation method (Yugi *et al.,*, 2002). Some more algorithm modules, including the tau leaping method (Gillespie, 2001), a cellular automata module, a diffusion-reaction module, a StochSim module (Le Novere and Shimizu, 2001), and adaptive switching between Gillespie and ODE methods are currently under development. Figure 4.18 summarizes development status of algorithm modules as of E-Cell Simulation Environment Version 3.1.100, released in January 2004.

4.3 Discussion

This thesis includes some selected chapters from the users' manual of the software in Appendix A. This stripped-down version of the manual should give enough information to grasp the architecture of the system more in detail, and to know how the system can be extended and customized for large-scale cell simulation projects. Readers interested in more practical utility of the system are directed to the full version of the manual (Takahashi, 2004).

E-Cell 3 is currently widely used for modeling, simulation and analysis of cell systems and parts of cell systems. The latest version of E-Cell System as of January 2004 is version 3.1.100. The first stable version, version 3.2 is going to be released in 2004.



Algorithm	Base class	Features	Status
ODE / DAE	Differential	Deterministic, continuous. Compound reactions and dynamical models.	Several available.
Difference equation	DiscreteTime	Discrete processes.	Available.
Gillespie-Gibson	DiscreteEvent	Stochastic elementary reactions.	Available.
ESSYNS	Differential	S-System, Generalized Mass-Action.	Available.
Algebraic / time-less	Passive	Algebraic rules, discrete processes	Available.
Hybrid ODE / flux-distribution	Differential / Passive	Hybrid static/dynamic pathway simulation.	Available.
Tau-leap	DiscreteEvent	Approximate Gillespie.	Implementing.
StochSim	DiscreteTime	Meso-scopic, multi-state.	Designing.
Gillespie-Langevin-ODE dynamic Switching	DiscreteEvent	Multi-scale, stochastic/deterministic, elementary reactions.	Designing.
CellularAutomata	DiscreteTime	Discrete-time cellular automata.	Designing.
Diffusion	Differential	Molecular diffusion	Planning.
Particle dynamics	DiscreteEvent	Particle dynamics, brownian motion	Planning.

Figure 4.18: Algorithm modules being developed for E-Cell SE
Some algorithm modules available and being developed for E-Cell SE are listed. Algorithm: the name of the simulation algorithm. Base class: the class name of the Stepper base class. Features: some major features of the algorithm. Status: development status of the algorithm module.

Chapter 5

Conclusions

5.1 Summary

This thesis discussed numerical simulations in computational cell biology in three parts. First we overviewed the area of research from the viewpoint of simulation studies, and commonly used modeling schemes and numerical methods have been reviewed. We performed a domain requirements analysis, and identified seven 'requirements' desirable for simulation platforms used in this field of research.

Secondly, a novel computational framework for cell simulation that can drive many simulation sub-models with different driving algorithms and accompanying timescales has been developed. This object-oriented framework based on a discrete event scheduler and Hermite polynomial interpolation, the 'meta-algorithm', defines discrete and continuous 'transition functions', and can incorporate virtually any, discrete/continuous and deterministic/stochastic, time-driven simulation algorithms in a modular way. Implemented simulation algorithm modules can be used in any combination. It was shown that 'meta-algorithm' can efficiently handle two relatively simple multi-algorithm and multi-timescale models, (1) the ODE/Gillespie composite *E. coli* heat shock response model and (2) the model of multi-timescale coupled harmonic oscillators. It was suggested that this framework will likely be scaled up to more realistic, complex and large-scale computational cell biology problems.

Lastly, the 'meta-algorithm' is implemented on E-Cell System, a generic software environment for computational cell biology. It has been verified that, owing to its object-oriented design, the meta-algorithm can be implemented on a real software platform using popular C++ language in an intuitive and efficient way.

5.2 Future directions

Let us review the list of seven identified requirements yet again.

1. **Multi-algorithm simulation*
2. **Multi-timescale simulation*
3. **Object-oriented modeling*
4. **Object-oriented simulation*
5. **Runtime user interaction*
6. Dynamic model structure
7. Spatial modeling and simulation

Marked with * (asterisk) are items we have given solutions in this thesis. The 'meta-algorithm' computational framework described in chapter 3 has been designed to meet first two of the requirements, (1) multi-algorithm simulation and (2) multi-timescale simulation. The framework itself is object-oriented, and our software, E-Cell SE Version 3, is implemented in a fully object-oriented way. Therefore, we believe that we have also been able to meet requirements (3) OO modeling and (4) OO simulation in this work. Requirement (5), realtime UI, in addition to (3) and (4) are in fact some of those items we could be able to fulfill in the previous version of the software, E-Cell System Version 1 (Takahashi *et al.,*, 2003; Tomita *et al.,*, 1999).

Remaining two items are the requirement (6) Dynamic model structure, and (7) Spatial modeling and simulation, which we will tackle in a future work to extend the 'meta-algorithm' that will form a basis of a succeeding version of the software.

5.2.1 Dynamic model structure

The necessity of dynamic structure modeling arises from the nature of the target system, the cell. That is, as commonly observed in living systems, the cell changes its physical, chemical and inter-cellular system structures constantly. There can be two fundamentally different approaches to treat system structures as variables. The first possibility is to use a special formalism that treats the structure as parameters. Power-law formalisms such as S-System and GMA introduced in chapter 2 are some good examples. These canonical differential systems treat dynamics parameters (*i.e.* rate constants) and system structure indiscriminately as forms of coefficient matrices called S-System or GMA matrix. Many are actively exploring this approach for network structure prediction problems of bio-pathways such as genetic networks and signaling cascades. For instance, a recently developed method called PEACE (Kikuchi *et al.,*, 2003), that can estimate network structure and kinetic parameters at the same time by coupling a variant of Genetic Algorithms and S-System formalism is being implemented on E-Cell Simulation Environment.

Although very useful for applications in biochemical inverse problems, these methodologies rely on special types of formalisms, and thus have limited utility when how to support dynamic structure in the generic simulation framework for computational cell biology is to be considered. Thus the support must be at the level of the fundamental framework, the meta-algorithm. Necessary improvements to the framework on this line may include these points. We need:

1. to support dynamic changes in connections between objects, for instance Processes and Variables (VariableReferences), Processes and Steppers, Systems and other Entities.
2. a mechanism to notify related objects of creations and deletions of other objects and changes in the connections between objects, and a programming framework (*e.g.* APIs) to adequately handle these dynamic events.

5.2.2 Spatial modeling and simulation

There have been a number of works in spatial modeling of the cellular systems (for example, see Howard *et al.,* (2001); Loew and Schaff (2001)). However, most previous works are developed in a conventional a-method-per-problem fashion (review the requirements 1 and 2). Rather, what we need here is a single generic framework that can treat all the phenomena in the cell with spatial extent, such as molecular diffusion in cytosol, vesicular trafficking, cytoplasmic streaming, cytoskeletal movement at the same time. As discussed for the requirement 7 in the chapter 2, the framework should be able to:

1. support development of algorithm modules for various spatial representations used in, at least, reaction-diffusion systems, particle dynamics (such as Brownian

dynamics), fluid dynamics, and dynamic morphological change of the cell.

2. allow interactions between these spatial sub-models.

There are at least two possible approaches to this problem. The first possibility is the development of a new spatial framework in which all of often used spatial modeling methodologies such as structured/unstructured meshings, voxel modeling, cellular automata *etc.* can be implemented. The other way is to look for an existing formalism that is generic enough to handle those common spatial simulation methods, and can be incorporated into the meta-algorithm framework easily. One of such existing formalisms is Cellular Automata introduced in chapter 2. From a purely requirements-based viewpoint, the former option is certainly beneficial because it is more generic and includes CA as a special case. However, the difficulty of developing such a platform and how efficient it can be is still highly unclear at this point of development. Therefore the use of CA as an alternative looks somewhat attractive. Unlike the case of the design options for the support for the dynamic model structure, both possibilities are equally beneficial, and we need further investigations comparing any of these lines of design options. Perhaps we would develop the CA framework first, and when it is found to be not enough, proceed to the development of the generic support.

5.2.3 Other possible improvements

There is prior art in use of Hermite polynomial for multi-timescale simulations. In Makino and Aarseth (1992), a Hermite integration scheme for gravitational many-body problems is described. In this algorithm, each particle has an individual time step size, and its position and velocity are integrated by using Hermite interpolation. Owing to its single-formalism (the law of gravity) nature, higher order derivatives are analytically available. Inspired by this related work is to integrate Hermite polynomial with the meta-algorithm framework more radically. Currently the calculations and uses of the Hermite interpolants are implemented in each differential simulation algorithm modules. By having the support for Hermite interpolants in the meta-algorithm itself, we would be able to get more chances to even clean up the design and optimize the implementation.

5.3 Conclusion

Design and implementation of a generic computational framework and scientific software demand a way of thinking that is completely different from ordinary virtues in many other scientific disciplines. Well defined and concrete problems often characterize good problem-driven scientific projects, whilst abstraction and generalization of the problems are the driving forces for the algorithm and software developers. Although at the time of writing this it is somewhat unclear that what extent of application domains are to be given to the algorithmic, computational and software frameworks developed in this work successfully, we strongly believe that, at least, necessity of novel approaches to the computational problems those characterize the new field of computational cell biology, including multi-formalism and multi-scale simulations, poses genuine challenges. We hope that the ideas described in this work stimulates further discussions and developments.

Appendix A

Selected Manual Chapters

To aid readers in understanding E-Cell System's object-model and overall architecture, two chapters from E-Cell Simulation Environment Users' manual have been selected and presented. The first selected chapter, 'Modeling with E-Cell' describes basics and some practical aspects of the model construction in the software system. The other 'Scripting A Simulation Session' chapter is presented here to show how the system allows frontend modules including user-level scripting of simulation sessions, mathematical analysis modules and graphical user interfaces. This appendix chapter is not intended to be a complete users' manual which describes everything necessary to get started to use E-Cell System. Needless to say, those who want to actually try the software are directed to the software manual itself (Takahashi, 2004).

A.1 Modeling With E-Cell

By reading this chapter, you can get information about:

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

Objects In The Model

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

Types of the Objects

A simulation model of E-Cell Simulation Environment consists of the following types of objects.

- Usually more than one `Entity` objects
- One or more `Stepper` object(s)

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

Entity objects

The `Entity` class has three subclasses:

- `Variable`

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

- `Process`

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

- `System`

  This class of objects define overall structure of the model. A `System` object can contain sets of these three types of `Entity`, `Variable`, `Process`, and `System` objects. A `System` can contain other `Systems`, and can form a tree-like structure.

Stepper objects

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

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

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

Object Identifiers

E-Cell Simulation Environment uses several types of identifier strings to specify the objects, such as the `Entity` and `Stepper` objects, in a simulation model.

ID (*Entity ID* and *Stepper ID*)

Every `Entity` and `Stepper` object has an *ID*. ID is a character string of arbitrary length starting from an alphabet or '_' with succeeding alphabet, '_', and numeric characters. E-Cell Simulation Environment treats IDs in a case-sensitive way.

If the ID is used to indicate a `Stepper` object, it is called a *Stepper ID*. The ID points to an `Entity` object is refered to as *Entity ID*, or just *ID*.

(need EBNF here) Examples: `_P3`, `ATP`, `GlucoKinase`

SystemPath;

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

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

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

FullID

A *FullID* (FULLy qualified IDentifier) identifies a unique `Entity` object in a simulation model. A *FullID* comprises three parts, (1) a *EntityType*, (2) a *SystemPath*, and (3) an *Entity ID*, joined by a character '`:`' (colon).

```
EntityType:SystemPath:ID
```

The *EntityType* is one of the following class names:

- `System`
- `Process`
- `Variable`

For example, the following *FullID* points to a `Process` object of which *Entity ID* is '`P`', in the `System` '`CELL`' immediately under the root system (`/`).

```
Process:/CELL:P
```

FullPN

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

```
FullID:property_name
```

or,

```
EntityType:SystemPath:ID:property_name
```

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

```
Variable:/CELL:S:Value
```

Object Properties

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

Types of object properties

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

- Real number

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

- Integer number

  (ex. `3`, `100`)

- String

  String has two forms: quoted and not quoted. A quoted String can contain any ASCII characters except the quotation characters (' or "). Quotations can be omitted if the string has a form of a valid object identifier (*Entity ID*, *Stepper ID*, *SystemPath*, *FullID*, or *FullPN*).

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

  (ex. `_C10_A`, `Process:/A/B:P1`, `"It can include spaces if double-quoted."`, `'single-quote is available too, if you want to use "double-quotes" inside.'`)

- List

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

  (ex. `[ A 10 [ 1.0 "a string" 1e+10 ] ]`)

Dynamic type adaptation of property values

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

The conversion is done in the following manner.

How property value type adaptation is conducted

- From a numeric value (Real or Integer)
  - To a String
    The number is simply converted to a character string. For example, a number 12.3 is converted to a String `'12.3'`.
  - To a list
    A numeric value can be converted to a length-1 list which has that number as the first item. For example, 12.3 is equivalent to '[12.3]'.
- From a String
  - To a numeric value (Real or Integer)
    The initial portion of the String is converted to a numeric value. The number can be represented either in a decimal form or a hexadecimal form. Leading white space characters are ignored. 'INF' and 'NAN' (case-insensitive) are converted to an infinity and a NaN (not-a-number), respectively. If the initial portion of the String cannot be converted to a numeric value, it is interpreted as a zero (0.0 or 0). This conversion procedure is equivalent to C functions `strtol` and `strtod`, according to the destined type.
  - To a list
    A String can be converted to a length-1 list which has that String as the first item. For example, 'string' is equivalent to '['string']'.
- From a list
  - To a numeric or a String value
    It simply takes the first item of the list. If necessary the taken value is further converted to the destined types.

Overflow and underflow when converting a property value

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

E-Cell Model (EM) File Basics

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

What is EM?

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

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

Why and when use EM?

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

EM files can be viewed as EML generator scripts.

EM at a Glance

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

This example is a model of a mass-action differential equation. In this example, the model has a `Stepper ODE_1` of class `ODE45Stepper`, which is a generic ordinary differential equation solver. The model also has the root system (`/`). The root sytem has the StepperID property, and four `Entity` objects, `Variables SIZE`, `S` and `P`, and the `Process E`. `SIZE` is a special name of the `Variable`, that determines the size of the compartment. If the compartment is three-dimensional, it means the volume of the compartment in [L] (liter). That value is used to calculate concentrations of other `Variables`. These `Entity` objects have their property values of several different types. For example, `StepperID` of the root system is the string without quotes (`ODE_1`). The initial value given to Value property of the `Variable S` is an integer number `10000` (and this is automatically converted to a real number `10000.0` when the `Variable` gets it because the type of the Value property is Real). Name property of the `Process E` is the quoted string `"A mass action from S to P"`, and 'k' of it is the real number `1.0`. VariableReferenceList property of `E` is the list of two lists, which contain strings (such as `S0`), and numbers (such as `-1`). The list contain relative *FullID*s (such as `:.:S`) without quotes.

General Syntax of EM

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

General form of object instantiation statements

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

```
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 ];
        }

}
```

Example A.1.1: A tiny EM example

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

where:

- TYPE

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

  - `Stepper`
  - `Variable`
  - `Process`
  - `System`

- ID

  This is a *StepperID* if the object type is `Stepper`. If it is `System`, put a *SystemPath* here. Fill in an *Entity ID* if it is a `Variable` or a `Process`.

- CLASSNAME

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

- INFO

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

- PROPERTY

  An object definition has zero or more properties.

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

  ```
  Concentration 10.0;
  ```

  Real, Integer, String, and List are allowed as property value types (See the Object Properties section above).

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

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

into a property slot Foo, write a line in the object definition like this:

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

Why the outermost brackets can be ommited?

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

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

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

is interpreted by the system as a length-1 List

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

of which the first item is a list

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

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

Macros and Preprocessing

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

By using empy, you can embed any Python expressions and statements after '@' in an EM file. Put a Python expression inside '@(python expression)', and the macro will be replated with an evaluation of the expression. If the expression is very simple, '()' can be ommited. Use '@{ pytyon statements }' to embed Python statements. For example, the following code:

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

is expanded to:

```
10
```

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

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

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

is expanded to

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

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

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

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

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

Comments

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

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

Structure Of The Model

Top Level Elements

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

```
STEPPER_0
STEPPER_1
...
STEPPER_n

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

`STEPPER_?` is a `Stepper` statement and `SYSTEM_?` is a `System` statement.

Systems

The root system

The model must have a `System` with a *SystemPath* '/'. This `System` is called the *root system* of the model.

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

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

Constructing the system tree

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

```
System System( / )
{
}

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

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

```
System System( / )
{
}

System System( /CELL0 )
{
}

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

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

```
System System( / )
{
}

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

Planned support for model composition

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

Sizes of the Systems

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

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

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

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

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

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

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

SIZE MUST BE A POSITIVE REAL NUMBER

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

Unit of the size

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

Variables and Processes

A `System` statement has zero, one or more `Variable` and `Process` statements in addition to its properties.

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

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

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

Do not put a `System` statement inside `System`.

Connecting Steppers with Entity Objects

Any `Process` and `Variable` object in the model must be connected with a `Stepper` by setting its StepperID property. If the StepperID of a `Process` is omitted, it defaults to that of its supersystm (the `System` the `Process` belongs to). StepperID of `System` cannot be omitted.

In the following example, the root sytem is connected to the `Stepper STEPPER0`, and the `Process P0` and `P1` belong to `Steppers STEPPER0` and `STEPPER1`, respectively.

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

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

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

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

Connections between `Steppers` and `Variables` are automatically determined by the system, and cannot be specified manually. See the next section.

Connecting Variable Objects with Processes

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

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

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

The last two fields can be omitted:

```
[ reference_name FullID coefficient ]
```

or,

```
[ reference_name FullID ]
```

These elements have the following meanings.

1. Reference name

   This field gives a local name inside the `Process` to this *VariableReference*. Some `Process` classes use this name to identify particular instances of *VariableReference*.

   Currently, this reference name must be set for all *VariableReference*s, even if the `Process` does not use the name at all.

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

2. *FullID*

   This *FullID* specifies the `Variable` that this *VariableReference* points to.

   The *SystemPath* of this *FullID* can be relative. Also, *EntityType* can be omitted. That is, writing like this is allowed:

   ```
   :.:S0
   ```

   instead of

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

   , if the `Process` exists in the `System /CELL`.

3. Coefficient (*optional*)

   This coefficient is an integer value that defines weight of the connection between the `Process` and the `Variable` that this *VariableReference* points to.

   If this value is a non-zero integer, then this *VariableReference* is said to be a *mutator* VariableReference, and the `Process` can change the value of the `Variable`. If the value is zero, this *VariableReference* is not a mutator, and the `Process` should not change the value of the `Variable`.

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

   If omitted, *this field defaults to zero.*

4. *isAccessor* flag (*optional*)

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

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

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

HOW ISACCESSOR FLAG IS USED IN THE SIMULATION

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

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

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

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

    }
}
```

Modeling Schemes

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

Discrete or Continuous ?

E-Cell can model both discrete and continuous processes, and these can be mixed in simulation. The system models discrete and continuous systems by discriminating two different types of `Process` and `Stepper` objects: discrete `Process` / `Stepper` and continuous `Process` / `Stepper`.

VARIABLE IS DISCRETE AND CONTINOUS

`Variable` and `System` do not have special discrete and continuous classes. The base `Variable` class supports both discrete and continous operations, because it can be connected to any types of `Process` and `Stepper` objects. `System` objects do not do any computation that needs to discriminate discrete and continuos.

Discrete classes

A `Process` object that models discrete changes of one or more `Variable` objects is called a *discrete* ***`Process`***, and it must be used in conjunction with a *discrete* ***`Stepper`***.

A discrete `Process` directly changes the *values* of related `Variable` objects when its `Stepper` requests to do so.

Continuous classes

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

Types of discrete object classes

There are two types of discrete `Process` / `Stepper` classes: discrete and discrete event.

- Discrete

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

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

- Discrete event

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

Followings are some examples of discrete types of object classes that the current version of E-Cell provides by default.

DiscreteTimeStepper

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

PassiveStepper

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

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

GillespieProcess and NRStepper (Gillespie-Gibson pair)

An example of discrete-event classes provided by E-Cell is a pair of `GillespieProcess` and `NRStepper`. `GillespieProcess`, which is a subclass of `DiscreteEventProcess`, calculates a time of the next occurence of the reaction using Gillespie's reaction probability equation and a random number. When this class is used with a `Stepper` (by setting a value of StepperID property) that implements the Gibson's Next Reaction

Method, `NRStepper`, E-Cell conducts a Gillespie-Gibson stochastic simulation of elementary chemical reactions. In fact, `NRStepper` is an alias to `DiscreteEventStepper`, because Gillespie-Gibson method is actually a typical discrete-event simulation. (See the next tutorial for detailed usage of `GillespieProcess` and `NRStepper`.)

Types of continuous object classes

Classes for ordinary differential equations

Continuous `Process` and `Stepper` objects are usually used to model differential systems: a set of differential equations.

Modeling Convensions

Units

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

- Time

  s (second)

- Volume

  L (liter)

- Concentration

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

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

A.2 Scripting A Simulation Session

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

What is E-Cell Session Script (ESS).
How to run ESS in scripting mode.
How to use ESS in GUI mode.
How to automate a simulation run by writing an ESS file.
How to write frontend software components for E-Cell in Python.

Session Scripting

An E-Cell Session Script (ESS) is a Python script which is loaded by a E-Cell `Session` object. A `Session` instance represents a single run of a simulation.

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

1. Loading a model file.

   Usually an EML file is loaded.

2. Pre-simulation setup of the simulator.

   Simulator and model parameters, such as initial values of `Variable` objects and property values of `Process` objects, are set and/or altered. Also, data `Logger`s may be created in this phase.

3. Running the simulation.

   The simulation is run for a certain length of time.

4. Post-simulation data processing.

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

5. Data saving.

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

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

Running E-Cell Session Script

There are three ways to execute ESS;

- Execute the script from the operating system's command line (the shell prompt).
- Load the script from frontend software such as Osogo Session Monitor.
- Use `SessionManager` to automate the invokation of the simulation sessions itself. This is usually used to write mathematical analysis scripts, such as parameter tuning, which involves multiple runs of the simulator.

Running ESS in command line mode

An ESS can be run by using **ecell3-session** command either in *batch mode* or in *interactive mode.*

Batch mode

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

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

ecell3-session command creates a simulation `Session` object and executes the ESS file `ess.py` on it. The option [-e] can be omitted. Optionally, if [-f model.eml] is given, the EML file `model.eml` is loaded immediately before executing the ESS.

Interactive mode

To run the **ecell3-session** in interactive mode, invoke the command without an ESS file.

```
$ ecell3-session -f model.eml
ecell3-session [ for E-Cell SE Version 3, on Python Version 2.2.1 ]
Copyright (C) 1996-2002 Keio University.
Send feedback to Kouichi Takahashi shafi@e-cell.org
ecell3-session>>>
```

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

Giving parameters to the script

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

To give the ESS parameters from the **ecell3-session** command, use either `-D` or `--parameters=` option.

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

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

Loading ESS from Osogo Session Monitor

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

gecell command accepts `-e` and `-f` options in the same way as the **ecell3-session** command.

Writing E-Cell Session Script

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

Using Session methods

General rules

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

For example, to run the simulation for 10 seconds, use `run` method of the `Session` object.

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

where `self.` points to the current `Session` object given by the system. Alternatively, you can use `theSession` in place of the `self`.

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

Unlike usual Python script, you can omit the object on which the method is called if the method is for the current `Session`.

```
run( 10 )
```

Loading a model

Let's try this in the interactive mode of the **ecell3-session** command. On the prompt of the command, load an EML file by using `loadModel`() method.

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

Then the prompt changes from `ecell3-session>>>` to `model_name, t=current time>>>`

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

Running the simulation

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

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

NOTE

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

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

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

Getting current time

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

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

Printing messages

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

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

An example of using `Session` methods

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

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

Example A.2.1: A simple ESS example.

Getting Session Parameters.

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

```
loadModel( MODELFILE )
```

To check what parameters are given to ESS, use `dir`() or `globals`() built-in functions. Session parameters are listed as well as other available methods and variables. To check if a certain ESS parameter or a global variable is given, write an if statement like this:

```
if 'MODELFILE' in globals():
    # MODELFILE is given
else:
    # not given
```

NOTE

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

Observing and Manipulating the Model with `ObjectStub`s

What is `ObjectStub`?

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

There are three types of `ObjectStub`:

- `EntityStub`
- `StepperStub`
- `LoggerStub`

each correspond to `Entity`, `Stepper`, and `Logger` classes in the simulator, respectively.

Why `ObjectStub` is needed

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

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

Creating an `ObjectStub` by ID

To get an `ObjectStub` object, `createEntityStub`(), `createStepperStub`(), and `createLoggerStub`() methods of `Session` class are used.

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

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

Similarly, a `StepperStub` object and a `LoggerStub` object can be retrieved with a *StepperID* and a *FullPN*, respectively.

```
aStepperStub = createStepperStub( 'STEPPER_01' )
```

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

Creating and checking existence of the backend object

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

To check if a corresponding object to an `ObjectStub` exists in the simulator, use `isExist()` method. For example, the following if statement checks if a Stepper whose ID is `STEPPER_01` exists:

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

To create the backend object, just call `create()` method.

```
aStepperStub.create()# Stepper 'STEPPER_01' is created here
```

Getting the name and a class name from an `ObjectStub`

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

To get the class name of an `EntityStub` or a `StepperStub`, call `getClassName()` method. This operation is not applicable to `LoggerStub`.

Setting and getting properties

As described in the previous chapters, `Entity` and `Stepper` objects has *properties*. This section describes how to access the object properties via `ObjectStub`s. This section is not applicable to `LoggerStub`s.

To get a property value from a backend object by using an `EntityStub` or a `StepperStub`, invoke `getProperty`() method or access an object attribute with a property name:

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

or equivalently,

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

To set a new property value to an `Entity` or a `Stepper`, call `setProperty`() method or mutate an object attribute with a property name and the new value:

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

or equivalently,

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

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

```
aStub.getPropertyList()
```

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

```
aStub.getPropertyAttribute( 'Activity' )[0] # true if setable
aStub.getPropertyAttribute( 'Activity' )[1] # true if getable
```

Getting `Logger` data

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

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

- getData()

  Get the whole data.
- getData(starttime [, endtime])

  Get a slice of the data from starttime to endtime. If endtime is omitted, the slice includes the tail of the data.
- getData(starttime, endtime, interval)

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

`getData`() method returns a rank-2 (matrix) array object of Numeric Python module. The array has either one of the following forms:

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

or

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

The first five-tuple data format has five values in a single datapoint:

- time

  The time of the data point.
- value

  The value at the time point.

- average

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

- min

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

- max

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

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

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

Getting and changing logging interval

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

An example usage of an `EntityStub`

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

Handling Data Files

About ECD file

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

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

```
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() )
```

Example A.2.2: An ESS to check ATP level every 10 seconds

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

Importing `ECDDataFile` class

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

```
import ecell
```

or import `ecell.ECDDataFile` module selectively.

```
import ecell.ECDDataFile
```

Saving and loading data

To save data to an ECD file, say, `datafile.ecd`, instantiate an `ECDDataFile` object and use `save()` method.

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

here `DATA` is a rank-2 array of Numeric Python or an equivalent object. The data can also be set by using `setData`() method after the instantiation. If the data is already set, it is replaced.

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

Loading the ECD file is also straightforward.

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

The `getData`() method extracts the data from the `ECDDataFile` object as an array.

ECD header information

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

- Data name

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

- Label

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

- Note

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

The header information is stored in the file like this.

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

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

Using ECD outside E-Cell SE

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

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

GNU gnuplot is a scientific presentation-quality plotting software with a sophisticated interactive command system. To plot an ECD file from gnuplot, just use **plot** command. For example, this draws a time-value 2D-graph:

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

Use **using** modifier to specify which column to use for the plotting. The following example makes a time-average 2D-plot.

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

Another OpenSource software useful for data processing is GNU Octave. Loading an ECD from Octave is also simplest.

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

Now the data is stored in a matrix variable with the same name as the file without the extension (`datafile`).

```
octave:2> mean(datafile)
ans =

   5.0663  51.7158  51.7158  51.2396  52.2386
```

Binary format

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

Manipulating Model Files

This section describes how to create, modify, and read EML files with the EML module of the E-Cell Python library.

Importing EML module

To import the EML module, just import `ecell` module.

```
import ecell
```

And `ecell.Eml` class is made available.

Other Methods

Getting version numbers

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

```
ecell3-session>>> import ecell
ecell3-session>>> ecell.ecs.getLibECSVersion()
'3.2.0'
ecell3-session>>> ecell.ecs.getLibECSVersionInfo()
(3, 2, 0)
```

DM loading-related methods

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

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

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

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

Advanced Topics

How ecell3-session runs

ecell3-session command runs on **ecell3-python** interpreter command. **ecell3-python**

command is a thin wrapper to the Python interpreter. **ecell3-python** command simply invokes a Python interpreter command specified at compile time. Before executing Python, **ecell3-python** sets some environment variables to ensure that it can find necessary E-Cell Python extension modules and the Standard DM Library. After processing the commandline options, **ecell3-session** command creates an `ecell.ecs.Simulator` object, and then instantiate a `ecell.Session` object for the simulator object.

Thus basically **ecell3-python** is just a Python interpreter, and frontend components of E-Cell SE run on this command. To use the E-Cell Python Library from **ecell3-python** command, use

```
import ecell
```

statement from the prompt:

```
$ ecell3-python
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ecell
>>>
```

or, (on UNIX-like systems) execute a file starting with:

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

Getting information about execution environment

To get the current configuration of **ecell3-python** command, invoke **ecell3-python** command with a -h option. This will print values of some variables as well as usage of the command.

```
$ ecell3-python -h
[...]

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

[...]
```

The 'PYTHON =' line gives the path of the Python interpreter to be used.

Debugging

To invoke **ecell3-python** command in debugging mode, set ECELL_DEBUG environment variable. This runs the command on a debugger software. If found, GNU gdb is used as the debugger. ECELL_DEBUG can be used for any commands which run on **ecell3-python**, including **ecell3-session** and **gecell**. For example, to run **ecell3-session** in debug mode on the shell prompt:

```
$ ECELL_DEBUG=1 ecell3-session -f foo.eml
gdb --command=/tmp/ecell3.0mlQyE /usr/bin/python
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
[New Thread 1074178112 (LWP 7327)]
ecell3-session [ E-Cell SE Version 3.2.0, on Python Version 2.2.2 ]
Copyright (C) 1996-2003 Keio University.
Send feedback to Kouichi Takahashi <shafi@e-cell.org>
<foo.eml, t=0>>> CtrlC
Program received signal SIGINT, Interrupt.
[Switching to Thread 1074178112 (LWP 7327)]
0xffffe002 in ?? ()
(gdb)
```

It automatically runs the program with the commandline options with '–command=' option of gdb. The gdb prompt appears when the program crashes or interrupted by the user by pressing **Ctrl+C**.

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

Profiling

It is possible to run **ecell3-python** command in profiling mode, if the operating system has GNU sprof command, and its C library supports `LD_PROFILE` environmental variable. Currently it only supports per-shared object profiling. (See GNU C Library Reference Manual)

To run **ecell3-python** in profiling mode, set `ECELL_PROFILE` environment variable to *SONAME* of the shared object. SONAME of a shared object file can be found by using objdump command, with, for example, -p option.

For example, the following commandline takes a performance profile of Libecs:

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

After running, it creates a profiling data file with a filename `SONAME.profile` in the *current directory*. In this case, it is `libecs.so.2.profile`. The binary profiling data can be converted to a text format by using **sprof** command. For example:

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

Appendix B

History of E-CELL System

In the spring of 1996, a group project on molecular biology of *Mycoplasma genitalium* driven mainly by undergraduate students, initially called ’the Mycoplasma project’, had started in Laboratory for Bioinformatics, Keio University SFC (Shonan-Fujisawa campus). In the October of that year, the head of the laboratory, Dr. Masaru Tomita, recruited several undergraduate and graduate students in his lab, including Takahashi, a junior at that time, for the development of software that can graphically represent dynamic changes in activities of genes. Those students worked in a competitive way with completely different approaches, and on November 15th, Takahashi gave a presentation in which he proposed (1) the project to focus on the kinetic dynamics of metabolic pathways and the control of enzyme productions through the expressions of genes, and (2) development of a generic software system based on object-orientation in C++ language. The software was initially named ECL (Electronic Cell Laboratory), and was later given the current name ’E-CELL System’, or ’Electronic Cell System’, by Kenta Hashimoto, a senior in the lab. In December 29th, Takahashi made an internal release of an alpha version (version 0.0-pre), of the software to lab members. The first user of E-CELL System, who had been testing the software from the ’pre-alpha’ days, was

Riow Matsushima, a junior at the time. This 3,500-line C++ piece of software had a minimum set of functionalities necessary to model and simulate a system of enzymatic reactions in a fully object-oriented way.

The object-model of this version defined two fundamental classes called primitives, 'Substance' and 'Reactor' for representations of molecular species and reactions, respectively, and was called the 'Substance-Reactor' model. By the end of the March, 1997, this object-model had given another primitive class called 'System', to model physical and logical (or functional) compartments in the cell. Hashimoto and Thomas S. Shimizu, a master student, contributed the discussion about this 'Structured Substance-Reactor' object model. Hashimoto also created a class of Reactor which calculates a chemical equilibrium. By the summer of 1997, Hashimoto participated in the development of a set of classes for simulation of gene expressions, and Yasuhiro Asakawa, a master student, and Katsuyuki Yugi, a junior in that year, developed a GUI component called 'GeneMapWindow' to graphically represent activities and amounts of products (mRNA molecules) of many genes.

In August and September of 1997, the software had been used in a collaborative project between Keio and TIGR (The Institute for Genomic Research), which aimed at *in silico* reconstruction of a virtual hypothetical cell with 127 genes based on *Mycoplasma genitalium*, which can self-sustain by producing energy from glucose with enzymes created from those genes. In addition to Dr. Tomita, Hashimoto, Shimizu, Takahashi, and Yugi, four more students, Yuri Matsuzaki, Fumihiko Miyoshi, Kanako Saito, and Sakura Tanida participated in this 'camp' held at an apartment in Baltimore, MD, USA, and Dr. C. Hutchson and Dr. J. C. Ventor of TIGR co-authored papers published later.

The version of the software used in this camp was further developed to become the version '1.0beta' of the software with helps from many contributors, and released under E-Cell Beta-testing License in March 2nd, 2000. On October 13th, 2000, E-Cell 1 had been accepted by the Bioinformatics.org as an OpenSource project under the terms of GPL (GNU General Public License).

In 2000, development projects of two other versions of E-Cell System, E-Cell 2 and E-Cell 3, started. E-Cell 2, developed by Naota Ishikawa, is a Windows port of E-Cell 1 which runs only on Linux operating system. E-Cell 3, initially started its development on Bioinformatics.org and later moved to Sourceforge.net in early 2003, is a complete reconstruction of E-Cell 1. E-Cell 3 can be viewed as an object-oriented computation platform on which any types of simulation algorithms can be used in any combination. In E-Cell 3, the 'Substance-Reactor' model of E-Cell 1 has been renamed to 'Variable-Process' model for further flexibility in modeling. Unlike E-Cell 1, it is possible to create 'wrappers', or interface layers, of its C++ API. A wrapper for the Python language has been developed, allowing development of frontend software in the productive scripting language. In addition to a GUI simulation session monitor component and a model editor, a distributed computation framework called E-Cell SessionManager is under development.

Bibliography

Arjunan, S., Takahashi, K. and Tomita, M. (2003). Shared-memory multi processing of gillespie's stochastic simulation algorithm on E-Cell 3. In *Proceedings of the Fourth International Conference on Systems Biology.*

Barshop, B. A. Wrenn, R. F. and Frieden, C. (1983) Analysis of numerical methods for computer simulation of kinetic processes: Development of kinsim–a flexible, portable system. *Anal. Biochem.* **130**(1), 134–145.

Bartol, T. M. J., Stiles, J. R., Salpeter, M. M., Salpeter, E. E. and Sejnowski, T. J. (1996) Mcell: generalized monte carlo computer simulation of synaptic transmission and chemical signaling. *Soc. Neurosci. Abstr.* **22**, 1742.

Bray, D., Bourret, R. B. and Simon, M. I. (1993) Computer simulation of the phosphorylation cascade controlling bacterial chemotaxis. *Mol. Biol. Cell.* **4**(5), 469–482.

Dawson, S. P., Chen, S. and Doolen, G. D. (1993) Lattice boltzmann computations for reaction-diffusion equations. *J. Chem. Phys.* **98**(2), 1514–1523.

Dormand, J. R. and Prince, P. J. (1980) A family of embedded runge-kutta formulae. *J. Comp. Appl. Math.* **6**, 19–26.

Ermentrout, G. B. and Edelstein-Keshet, L. (1993) Cellular automata approaches to biological modelling. *J. Theor. Biol.* **160**(1), 97–133.

Fehlberg, E. (1969). Low-order classical runge-kutta formulas with stepsize control and their application to some heat transfer problems. Technical Report NASA-TR-R-315, NASA.

Gamer, J., Multhaup, G., Tomoyasu, T., McCarty, J. S., Rudiger, S., Schonfeld, H. J., Schirra, C., Bujard, H. and Bukau, B. (1996) A cycle of binding and release of the dnak, dnaj and grpe chaperones regulates activity of the *Escherichia coli* heat shock transcription factor sigma32. *EMBO J.* **15**(3), 607–617.

Gear, C. W. (1971). *Numerical initial value problems in ordinary differential equations.* Prentice-hall series in automatic computation. Englewood Cliffs, N.J.: Prentice-Hall.

Gibbons, F., Chauwin, J. F., Desposito, M. and Jose, J. V. (2001) A dynamical model of kinesin-microtubule motility assays. *Biophys. J.* **80**(6), 2515–2526.

Gibson, M. A. and Bruck, J. (2000) Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A* **104**(9), 1876–1889.

Gillespie, D. T. (1976) A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *J.Comp. Phys.* **22**, 403–434.

Gillespie, D. T. (1977) Concerning the validity of the stochastic approach of chemical kinetics. *J. Stat. Phys.* **16**, 311–319.

Gillespie, D. T. (1992) A rigorous derivation of the chemical master equation. *Physica A* **188**, 404–425.

Gillespie, D. T. (2001) Approximate accelerated stochastic simulation of chemically reacting systems. *J. Chem. Phys.* **115**(4), 1716–1733.

Gillespie, D. T. and Petzold, L. R. (2003) Improved leap-size selection for accelerated stochastic simulation. *J. Chem. Phys.* **119**(16), 8229.

Ginkel, M., Kremling, A., Nutsch, T., Rehner, R. and Gilles, E. D. (2003) Modular modeling of cellular systems with ProMoT/Diva. *Bioinformatics* **19**(9), 1169–76.

Goryanin, I., Hodgman, T. C. and Selkov, E. (1999) Mathematical simulation and analysis of cellular metabolism and regulation. *Bioinformatics* **15**(9), 749–758.

Gross, C. A. (1996). Function and regulation of the heat shock proteins. In F. C. Neidhardt and R. Curtiss (Eds.), *Escherichia coli and Salmonella : Cellular and molecular biology* (2nd ed.)., pp. 1382–1399. Washington, D.C.: ASM Press.

Grossman, A. D., Straus, D. B., Walter, W. A. and Gross, C. A. (1987) Sigma 32 synthesis can regulate the synthesis of heat shock proteins in *Escherichia coli. Genes Dev.* **1**(2), 179–184.

Hairer, E. and Wanner, G. (1996). *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems* (2nd ed.), Volume 14 of *Comput. Math.* Berlin: Springer.

Haseltine, E. L. and Rawlings, J. B. (2002) Approximate simulation of coupled fast and slow reactions for stochastic chemical kinetics. *J. Chem. Phys.* **117**(15), 6959–6969.

Hashimoto, K., Miyoshi, F., Seno, A. and Tomita, M. (1999). Generic gene expression system for modeling complex gene regulation network using E-Cell system. In *Genome Informatics*, Tokyo, pp. 356–357. Universal Academy Press Inc.

Hasty, J., McMillen, D., Isaacs, F. and Collins, J. J. (2001) Computational studies of gene regulatory networks: In numero molecular biology. *Nat. Rev. Genet.* **2**(4), 268–279.

Headley, W. and Nelson, M. (2001). Cellml specification.

Hitz, M. and Werthner, H. (1997). Earning benefits of the object-oriented paradigm in dynamic system. In *IEEE Hawaii International Conference on System Sciences*, Hawaii.

Howard, M., Rutenberg, A. and de Vet, S. (2001) Dynamic compartmentalization of bacteria: Accurate division in *E. Coli. Phys. Rev. Lett.* **87**, 278102.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. and Kitano, H. (2002). The erato systems biology workbench: Enabling interaction and exchange between software tools for computational biology. In *Pacific Symposium on Biocomputing*, Volume 7, pp. 450–461.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H. Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A., Cuellar, A. A., Dronov, S., Gilles, E. D., Ginkel, M., Gor, V., Goryanin, I. I., Hedley, W. J., Hodgman, T. C., Hofmeyr, J., Hunter, P. J., Juty, N. S., Kasberger, J. L., Kremling, A., Kummer, U., Le Novere, N., Loew, L. M., Lucio, D., Mendes, P., Minch, E., Mjolsness, E. D., Nakayama, Y., Nelson, M. R., Nielsen, P. F., Sakurada, T., Schaff, J. C., Shapiro, B. E., Shimizu, T. S., Spence, H. D., Stelling, J., Takahashi, K., Tomita, M., Wagner, J. and Wang, J. (2003) The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models. *Bioinformatics* **19**(4), 524–531.

Ichikawa, K. (2001) A-cell: Graphical user interface for the construction of biochemical reaction models. *Bioinformatics* **17**(5), 483–484.

Irvine, D. H. and Savageau, M. A. (1990) Efficient solution of nonlinear ordinary differential-equations expressed in s-system canonical form. *SIAM J. Num. Anal.* **27**(3), 704–735.

Kiehl, T. R., Mattheyses, R. M. and Simmons, M. K. (2004) Hybrid simulation of cellular behavior. *Bioinformatics (to appear).*

Kikuchi, S., Tominaga, D., Arita, M., Takahashi, K. and Tomita, M. (2003) Dynamic modeling of genetic networks using genetic algorithm and s-system. *Bioinformatics* **19**(5), 643–650.

King, E. L. and Altman, C. (1956) A schematic method of deriving the rate laws for enzyme-catalyzed reactions. *J. Phys. Chem.* **60**, 1375–1378.

Kootsey, J. M., Kohn, M. C., Feezor, M. D., Mitchell, G. R. and Fletcher, P. R. (1986) Scop: An interactive simulation control program for micro- and minicomputers. *Bull. Math. Biol.* **48**(3-4), 427–441.

Le Novere, N. and Shimizu, T. S. (2001) StochSim: Modelling of stochastic biomolecular processes. *Bioinformatics* **17**(6), 575–576.

Loew, L. M. and Schaff, J. C. (2001) The virtual cell: A software environment for computational cell biology. *Trends Biotechnol.* **19**(10), 401–406.

Makino, J. and Aarseth, S. J. (1992) On a hermite integrator with ahmad-cohen scheme for gravitational many-body problems. *Publ. Astron. Soc. Jpn.* **44**(2), 141–151.

Marion, G., Renshaw, E. and Gibson, G. (1998) Stochastic effects in a model of nematode infection in ruminants. *IMA J. Math. Appl. Med. Biol.* **15**(2), 97–116.

Matsumoto, M. and Nishimura, T. (1998) Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. Mod. Comput. Sim.* **8**, 3–30.

McAdams, H. H. and Arkin, A. (1998) Simulation of prokaryotic genetic circuits. *Annu. Rev. Biophys. Biomol. Struct.* **27**, 199–224.

McAdams, H. H. and Shapiro, L. (1995) Circuit simulation of genetic networks. *Science* **269**(5224), 650–656.

McQuarrie, D. A. (1967) Stochastic approach to chemical kinetics. *J. Appl. Prob.* **4**, 413–478.

Mendes, P. (1993) Gepasi: A software package for modelling the dynamics, steady states and control of biochemical and other systems. *Comput. Appl. Biosci.* **9**(5), 563–571.

Michaelis, L. and Menten, M. L. (1913) Die kinetik der invertinwirkung. *Biochem. Z* **49**, 333–369.

Morton-Firth, C. J. and Bray, D. (1998) Predicting temporal fluctuations in an intracellular signalling pathway. *J. Theor. Biol.* **192**(1), 117–128.

Morton-Firth, C. J., Shimizu, T. S. and Bray, D. (1999) A free-energy-based stochastic simulation of the tar receptor complex. *J. Mol. Biol.* **286**(4), 1059–1074.

Novitski, C. E. and Bajer, A. S. (1978) Interaction of microtubules and the mechanism of chromosome movement (zipper hypothesis). 3 theoretical analysis of energy requirements and computer simulation of chromosome movement. *Cytobios* **18**(71-72), 173–182.

Patel, A. A., Gawlinski, E. T., Lemieux, S. K. and Gatenby, R. A. (2001) A cellular automaton model of early tumor growth and invasion. *J. Theor. Biol.* **213**(3), 315–331.

Phair, R. D. and Misteli, T. (2001) Kinetic modelling approaches to *in vivo* imaging. *Nat. Rev. Mol. Cell. Biol.* **2**(12), 898–907.

Press, W. H. (1988). *Numerical recipes in c : The art of scientific computing.* Cambridge Cambridgeshire ; New York: Cambridge University Press.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (2002). *Numerical* *recipes in C++ : The art of scientific computing* (2nd ed.). Cambridge ; New York: Cambridge University Press.

Rao, C. V., Wolf, D. M. and Arkin, A. P. (2002) Control, exploitation and tolerance of intracellular noise. *Nature* **420**(6912), 231–237.

Rathinam, M., Petzold, L. R., Cao, Y. and Gillespie, D. T. (2003) Stiffness in stochastic chemically reacting systems: The implicit tau-leaping method. *J. Chem. Phys.* **119**(24), 12784.

Sauro, H. M. (1993) Scamp: A general-purpose simulator and metabolic control analysis program. *Comput. Appl. Biosci.* **9**(4), 441–450.

Savageau, M. A. (1976). *Biochemical systems analysis : A study of function and design in molecular biology.* Reading, Mass.: Addison-Wesley Pub. Co. Advanced Book Program.

Shampine, L. F. (1986) Some practical runge-kutta formulas. *Math. Comput.* **46**(173), 135–150.

Shimizu, T. S., Aksenov, S. V. and Bray, D. (2003) A spatially extended stochastic model of the bacterial chemotaxis signalling pathway. *J. Mol. Biol.* **329**(2), 291–309.

Singer, K. (1953) Application of the theory of stochastic processes to the study of irreproducible chemical reactions and nucleation processes. *J. R. Stat. Soc. B* **15**, 92–106.

Srivastava, R., Peterson, M. S. and Bentley, W. E. (2001) Stochastic kinetic analysis of the *Escherichia coli* stress circuit using sigma(32)-targeted antisense. *Biotechnol. Bioeng.* **75**(1), 120–129.

Srivastava, R., You, L., Summers, J. and Yin, J. (2002) Stochastic vs. deterministic modeling of intracellular viral kinetics. *J. Theor. Biol.* **218**(3), 309–321.

Takahashi, K. (2004). *E-Cell Simulation Environment Users' Manual.* E-Cell Project.

Takahashi, K., Ishikawa, N., Sadamoto, Y., Sasamoto, H., Ohta, S., Shiozawa, A., Miyoshi, F., Naito, Y., Nakayama, Y. and M., T. (2003) E-Cell 2: Multi-platform E-Cell simulation system. *Bioinformatics* **19**(13), 1727–1729.

Takahashi, K., Kaizu, K., Hu, B. and Tomita, M. (2004) A multi-algorithm, multi-timescale method for cell simulation. *Bioinformatics (to appear).*

Takahashi, K., Yugi, K., Hashimoto, K., Yamada, Y., Pickett, C. J. F. and Tomita, M. (2002) Computational challenges in cell simulation: A software engineering approach. *IEEE Intell. Syst.* **17**(5), 64–71.

Tomita, M., Hashimoto, K., Takahashi, K., Shimizu, T. S., Matsuzaki, Y., Miyoshi, F., Saito, K., Tanida, S., Yugi, K., Venter, J. C. and Hutchison, C. A., r. (1999) E-Cell: Software environment for whole-cell simulation. *Bioinformatics* **15**(1), 72–84.

Tyson, J. J., Chen, K. and Novak, B. (2001) Network dynamics and cell physiology. *Nat. Rev. Mol. Cell. Biol.* **2**(12), 908–916.

Vangheluwe, H. (2000). Devs as a common denominator for multi-formalism hybrid systems modelling. In A. Varga (Ed.), *IEEE International Symposium on Computer-Aided Control System Design*, Anchorage, Alaska, pp. 129–34. IEEE Computer Society Press.

Vasudeva, K. and Bhalla, U. S. (2004) Adaptive stochastic-deterministic chemical kinetic simulations. *Bioinformatics* **20**(1), 78–84.

Weimar, J. R. (2002). Cellular automata approaches to enzymatic reaction networks. In S. Bandini, B. Chopard, and M. Tomassini (Eds.), *5th International Conference on Cellular Automata for Research and Industry ACRI*, Lecture Notes in Computer Science, Switzerland, pp. 294–303. Springer.

Yugi, K., Nakayama, Y. and Tomita, M. (2002). A hybrid static/dynamic simulation algorithm: Towards large-scale pathway simulation. In E. Aurell, J. Elf, and J. Jeppsson (Eds.), *The 3rd. International Conference on Systems Biology*, pp. 235.

Zajicek, G. (1968) A computer model simulating the behavior of adult red blood cells. red cell model. *J. Theor. Biol.* **19**(1), 51–66.

Zak, D., Gonye, G. E., Schwaber, J. S. and Doyle, F. J. I. (2003) Importance of input perturbations and stochastic gene expression in the reverse engineering of genetic regulatory networks: Insights from the identifiability analysis of an *in silico* network. *Genome Res.* **13**(11), 2396–2405.

Zeigler, B., Moon, Y., Kim, D. and Ball, G. (1997) The devs environment for highperformance modeling and simulation. *IEEE Computational Science and Engineering* **4**, 61–71.

Zeigler, B. P., Kim, T. G. and Praehofer, H. (2000). *Theory of modeling and simulation : Integrating discrete event and continuous complex dynamic systems* (2nd ed.). San Diego, London: Academic.

Index

A-stable, 36

B-convergence, 36
B-stable, 36
Butcher array, 30

Chemical Langevin equation, 51
chemical master equation, 41
Co-simulation, 70
ConcretePropertySlot, 121
consistency, 27
continuous Stepper, 75
continuous transition function, 74, 113
convergence, 27, 28

DAE, 17
database facade, 109
DifferentialStepper, 80, 110
discrete Stepper, 75
discrete transition function, 74, 115
DiscreteEventStepper, 80, 110
DiscreteTimeStepper, 110
dispatchInterruptions phase, 76, 117

E-Cell model description language, 109
E-CELL SessionManager, 107
embedded Runge-Kutta, 33
Entity class, 110
explicit Euler method, 27, 35, 51

First Reaction method, 44
Formalism combining, 71, 99
Formalism embedding, 71, 99
Formalism Transformation, 70

get method, 121
global truncation error, 28

implicit tau leap method, 50
initial value problem, 26
integrate phase, 76, 77, 117
interpolant difference, 78, 117
Interpolant object, 117
isAccessor flag, 115
isContinuous flag, 115

Jacobian, 34

King-Altman method, 17

L-stable, 37

Langevin method, 51
LastTime, 117
log phase, 76, 117
Logger, 112, 122
LoggerAdapter, 123
LoggerBroker, 112

macroscopically infinitesimal, 47
meta-algorithm, 15, 67, 70–72, 85, 98, 99, 103, 104, 110
Meta-formalism, 69
Metabolism, 17
Michaelis-Menten equations, 17
midpoint method, 31
Model, 72
Model class, 112

next reaction density function, 42
Next Reaction method, 45

ODE, 17, 18

PassiveStepper, 110, 112
Poisson random variable, 47
Process class, 110, 115
Process migration, 100
propensity function, 40
PropertiedClass class, 110, 121
PropertySlot, 121
root System, 112
Runge-Kutta coefficients, 30

Scheduler, 112
set method, 121
Signal transduction, 18
stability, 34
state change vector, 41
step phase, 76, 117
Stepper class, 110
Stepper Dependency, 75
stiff, 33
stiffly accurate, 36
stiffness, 49
System class, 110

tau leaping, 47, 99, 126
thermodynamical limit, 51

updated-time vector, 75

Value property, 116
Variable class, 110, 115
VariableReference class, 115
Velocity property, 116