# A Simulation System for Computational Cell Models Based on Object-Oriented Design Patterns

Kuanquan Wang (Corresponding author)
School of Computer Science and Technology, Harbin Institute of Technology
P.O. box 332, Harbin 150001, China
Tel: 86-451-8641-2871 E-mail: wangkq@hit.edu.cn

Yongfeng Yuan
School of Computer Science and Technology, Harbin Institute of Technology
P.O. box 332, Harbin 150001, China
Tel: 86-451-8641-2871 E-mail: yuanyongfeng@gmail.com

Jie Li
School of Computer Science and Technology, Harbin Institute of Technology
P.O. box 332, Harbin 150001, China
Tel: 86-451-8641-2871 E-mail: lj_ren@126.com

*The research is financed by the National Nature Science Foundation of China (No. 60571025 and No. 60872099) and the National High Technology Project of China (No. 2006AA01Z308).*

**Abstract**

Reconstruction of biophysically detailed computer models for simulating electrical activities of heart cells provides a powerful tool to systematically investigate the ionic mechanisms underlying the genesis and control of cardiac rhythms. However, the fact that there is no unified or standard architecture for computational cell models, which were built by different research groups with specific purposes, obstructed profound applications of these models. In this study, object-oriented design patterns were employed to redesign and reconstitute the cell models and provided a more flexible, portable, and expansible infrastructure for modeling computational cell models. Meanwhile, using the proposed methods, a simulation platform has been developed for electrical activities of cell models with aims to offer a user-friendly interface to study the electrical activities of cardiac cells. Both the proposed design methods and the developed system were validated effectively by testing several typical cell models.

**Keywords:** Object-oriented method, Design pattern, Computational cell model, Simulation platform

## 1. Introduction

Advances in cell biology and experimental technology have promoted the understanding of life behaviors of cells and generated massive amounts of biophysical data. As effective approaches, a great number of computational cell models based on different animal experimental data have been constructed and employed by cell biologists, bioengineers and other researchers to help them analyze and explain biological questions (Crampin, 2004; Noble, 2006). These computational models and related simulation systems have been playing a very important role in promoting the development of modern biology. However, the fact that there is no unified architecture or software system for these cell models, which were built by different research groups with various programming languages, obstructed profound applications of these models. Therefore it is crucial to develop a unified infrastructure or a user-friendly design approach for these cell models. In this study, object-oriented design patterns were used to reconstruct the architecture of these models, and to develop a modeling and simulation platform for computational cell models with aims to provide a user-friendly interface to study the electrical activities of cells. Some typical cell models have been employed for testing and validating the simulation system including the Luo-Rudy (LR) model (Luo & Rudy, 1994) for guinea pig ventricular cells, the Priebe-Beuckelman (PB) model (Priebe & Beuckelmann, 1998) and the ten Tusscher et al. (TNNP) model (ten

Tusscher et al., 2004) for human ventricular cells, and the Zhang et al. (Zhang) model (Zhang et al., 2000) for sinus node cells. As a result, the developed simulation system easily helped both cell biologists and mathematical modelers to build, validate, analyze and share their models.

The rest of this paper is organized as follows. Section 2 presents three important objected-oriented design patterns and their applications for cell models. Section 3 develops a framework and simulation system for computational cell models and analyses features of this system. At last, section 4 gives discussions and conclusions about the system.

## 2. Object-oriented design for cell models

In the last 50 years, a number of computational cell models had been constructed and employed to study the underlying electrophysiological mechanism of cells. These computational models mostly focused on their consistency with the experimental results, and they were usually expressed by lots of self-defined variables and many complex mathematical equations, and coded by different programming languages (Fortran, C, C++, Java, Matlab etc.). This made them very hard to be reused for other studies and extended to larger scale applications or to support unforeseeable functions. However, these models did share some common features in development, such as the structure of mathematical equations that was similar to the first identified mathematical cell model Hodgkin and Huxley model (Hodgkin and Huxley, 1952) and the methods to derive model parameters and their validation against experimental data. Developing a unified infrastructure for these models is pretty plausible, interesting and useful. The object-oriented design methods suit this task best. With the object-oriented methods, all cell models can be abstracted into the following four parts (Figure 1):

Structure: to define the geometric information of a single cell

Concentration: to specify the intra- and extra- cellular ionic concentrations

Ion channel: to describe the kinetics of ion channels

Conductance: to define the maximal conductance of ion channels.

An object-oriented software infrastructure, which is designed to reconstitute mathematical cell models, consists of three design patterns including *Abstract Factory pattern*, *Decorator pattern* and *Bridge pattern*, as shown in Figure 2. The detailed illustration is depicted as the following:

### *2.1 Abstract Factory pattern for constructing uniform infrastructure of cell models*

All biological users would expect that the users just need to understand functions of cell models and do not concern how to compute states, variables and equations of cell models. A cell model can be created as easily as making various shape cookies with templates and can be implemented anywhere for any applications with the standard interface. Figure 3 shows the *Abstract Factory pattern* for creating concrete objects for various cell models with the standard interface *createCell()*. Here, the *Cell* and *AbstractCellFactory* classes are equivalent to the *AbstractProduct* and the *AbstractFactory* in the *Abstract Factory pattern* respectively. The *LRdCell*, *PBCell*, *TenCell* and *ZhangCell* classes are substantiated subclasses of the *Cell* class and perform specific details of each cell model. The *LRdCellFactory*, *PBCellFactory*, *TenCellFactory* and *ZhangCellFactory* classes are responsible for implementing the uniform interface *createCell()* shared by the *AbstractCellFactory* and materializing cell instances for relative *Cell* subclasses respectively. The users can just access general properties of cell models such as *Structure*, *Concentration* and *Conductance*, but they do not know the details and processes to create and compute cell model objects It is the merit of using the *Abstract Factory pattern* that isolates cell models from their specific applications, maintains two parts independently and -makes it flexible and direct for adding new cell models into the existing applications through a uniform interface *ctreateCell()*.

### *2.2 Decorator pattern for complex applications of cell models*

Inheritance is a general way to add new functionalities into cell models by creating new classes. When a mass of path-physiological conditions need to be considered in a simulation study, it will produce unnecessarily a large number of inherited subclasses that causes the exploded increase of inherited classes, enhances the complexity of a simulation system and limits the flexibility of cell model development and maintenance. An approach called *Decorator pattern* is employed here to solve this problem. The pattern attaches additional responsibilities to an object dynamically and provides a flexible alternative to subclassing for extending functionality (Gamma et al., 1994). The key of *Decorator pattern* maintains a reference of the original cell object and forwards requests to the object before or after implementing additional functions for simulating path-physiological conditions. Through the pattern, additional responsibilities were integrated to cell models like playing the toy bricks.

As shown in Figure 4, the designed *Decorator pattern* has four major elements (*Cell*, *ConcreteCell*, *CellDecorator* and *ConcreteDecorator*). The *Cell* class is the same class as mentioned above (see 2.1 section) and defines an extending interface *update ()*, which needs to be decorated. The *ConcreteCell* stands for the subclasses *LRdCell*, *PBCell*, *TenCell* and *ZhangCell*. The abstract class *CellDecorator* declares an interface *update ()* that conforms to the interface *update ()* of the *Cell* class and maintains a reference to a cell object. The *ConcreteDecorator* classes (*BlockerDecorator*, *MedicineDeractor*), which are subclasses of the *CellDecorator* class, are responsible for adding additional functions to the reference of a *Cell* object inherited from the parent class *CellDecorator*, forwarding requests to the *Cell* object and performing additional actions before or after forwarding.

Using the Decorator *pattern* has at least two major benefits:

1) to increase additional functions to *Cell* objects dynamically and transparently. The decorated *Cell* object can define and perform additional functions dynamically or independently in a new subclass of *ConcreteDecorator* for unforeseen extensions through the overriding and forwarding mechanism. The subclass *CellDecorator* only changes a *Cell* object from the outside and the *Cell* object doesn't have to know anything about the decorated objects. Thus, the decorated objects are transparent to the objects that are related to the original *Cell* object and no influence on any existing applications.

2) to simulate complex electrical activities of cells by simple composition of multiple decorator classes. Instead of designing complex inherited *Cell* subclasses to support all kinds of possible compositions of functions, the *Decorator pattern* offers a pay-as-you-go approach to add new specific functions or composite several existing functions to a *Cell* model. For example, supposing our aim to simulate the actions of antiarrthymic drugs on a morbid cell, this can be achieved by nesting two specific decorator objects (*BlockerDecorator* and *MedicineDecorator* objects) step by step to a original *Cell* object (as shown in Figure 5). To produce a morbid cell, the *BlockerDecorator* object reduces the conductance of a certain single ion channel before forwarding requests to the original cell object, and then the *MedicineDecorator* object nests medicine interference to treat the block mentioned at previous step by forwarding requests to the morbid object before calling the morbid object. Through the nest of different decorator objects, an unlimited number of functions can be allowed to append the simulation system and - to simulate more complex electrical activities of cells.

### *2.3 Bridge pattern for flexible implementations of multiple numerical solvers*

In general, the computational cell models are made of ordinary differential equations (ODEs) and partial differential equations (PDEs). There are many numerical methods to solve ODEs and PDEs such as the Euler method, the Runge-Kutta method (RK2 Method) and the Rush-Larsen method (RL Method) (Rush and Larsen, 1978). These methods are quite different in the sections of time cost and accuracy. In order to avoid repeatedly coding the numerical solvers for each specific application, it is considered that *Bridge pattern* which is a structural pattern used to decouple an abstraction from its implementation so that the two can vary independently (Gamma et al., 1994). It is suitable to deal with this problem that an abstraction (mathematical equations of ODEs and PDEs) and many implementations (many numerical solution methods).

The *Bridge pattern* decouples an abstraction of mathematical formula from their numerical solvers as depicted in Figure 6. Referring to the relationship of cars and engines, the abstract class *Cell* is like the car and while the abstract class *AbstractMethod* is like the engine and their subclasses of the *Cell* or *AbstractMethod* classes are specific brands for cars or engines. The class *AbstractMethod* defines a public operation *solution()* which is like the uniform interface that the car fires the engine, that is, a car can use various brand engines whilst an engine can be equipped into different brand cars. The class *Cell* just declares the description of static mathematical equations of all cell models and maintains a reference to an instance of a certain subclass of the *AbstractMethod*, but do not specify any numerical solver. The class *AbstractMethod* defines a uniform solution procedure *solution ()* for all special numerical solution algorithms which are completed by concrete subclasses such as *Euler Method*, *RK2 Method* and *RL Method*. The reference is like a bridge connecting cell model objects to their implementation of a certain numerical solution and cell model objects by calling the *solution()* method in the reference to choose a specific solver. Except to maintain cell models and their implementations independently, the *Bridge pattern* has another advantage that cell models can switch various numerical methods at run-time. The cell action potential (AP) has different profiles from the Phase 0 to the Phase 4. The profiles of the Phase 0, 1 and 3 are steep, thus needs to use numerical methods with higher computing precision (e.g. the RK2 method). But the profiles of the Phase 2 and 4 of AP are oppositely steady, it is suitable to use the method with lower computing precision and less computing time (i.e. the Euler method). Because expression and implementation of ODEs and PDEs are separated in the pattern, it allows that a cell model to reselect or switch the instance of numerical solution methods according to different computing *precision* requirements at run-time. In turn, it can keep computational precision and meanwhile reduce computing time cost.

## 3. The developed simulation system for computational cell models

The framework of simulation system consists of six layers as shown in Figure 7. Every layer is briefly illustrated from bottom to top as below.

(1) Extensible applications layer

The layer is composed of plug-in programs for expanding functions of the simulation system. For example, the myocardiacal cell modeling language editor (*MCML Editor*) and *Convertor Engine* are plug-in programs that were developed by another project of our group (Wang et al., 2007). The *MCML Editor* module is a program for developing and editing cell models with XML language and the *Convertor Engine* module is responsible for automatically translating cell models built by *MCML Editor* into the .class Java file which can be directly executed in the simulation system. With two plug-in programs, novel cell models can be imported into the simulation system automatically and conveniently.

(2) Plug-in Layer

The Eclipse rich client platform (RCP) is an open-source platform for building and deploying rich client applications. The plug-in architecture of RCP provides a highly effective method for developing pluggable and dynamically extensible systems. With the architecture, all expandable applications (i.e. *MCML Editor* and *Convertor Engine*) can be easily integrated into the main system as plug-in programs.

(3) Simulating and computing layer

The *Simulating and Computing Layer* includes principle functions of the system. It is made of several important functional modules *Algorithms Supporting*, *Cell Models Management*, *Parameters Controlling* and *Protocols Controlling*. The *Algorithms Supporting* module offers all numeral solution methods for the simulation. The *Cell Models Managemen*t module organizes and views all cell models by a tree structure. The *Parameters Controlling* module adds, deletes and updates the properties of cell models under normal or path-physiological conditions. The last module *Protocol Controlling* defines stimulating protocols on cell models in order to simulate all kinds of electrical activities of cells.

(4) Simulation scheduler layer

The *Simulation Scheduler Layer* is the core of the whole simulation system and is responsible for interacting with customers, calling logic functional modules, controlling the whole simulation process, and responding the users' requests.

(5) Data container layer

The *Data Container Layer* is responsible for storing, cataloging, searching and organizing all result data. There are a lot of result data such as membrane voltage, ion currents, ion concentrations and ion channel conductance. The users can self-define their interested result data to view. The *Data Container Layer* provides many data containers as clones of original result data that the customers can reorganize and select their interested data to investigate and display.

(6) Data expression layer

The *Data Expression Layer* includes two parts *Data Analyzing* and *Data Plotting*. The *Data Analyzing* component has some basic statistic functions (i.e. means and standard deviation). The *Data Plotting* component takes charge of plotting data depended on the special view self-defined by users, showing the value at the appointed position, zooming in or out the simulated data and comparing computed data with experimental data.

The simulation system distinguishes itself with the following significant features.

(1) User-friendly interface (UI)

As shown in Figure 8, it provides a user-friendly interface and shows all input and output properties of cell models such as gated variables, individual ion current, and membrane action potentials. Simulated results can be exhibited as graphs and tables in the middle of the window. Customers can organize the plotting curves by themselves based on their interested simulated results, directly get the value of a certain point or the distance between two appointed positions with cursors and zoom in or out their interested field through mouse to resize and repaint the graphs.

(2) Virtual patch-clamp experiments

It performs a virtual patch-clamp experimental environment. The patch-clamp is a very important technique to investigate ion channel properties of cells (Molleman, 2003). The system can simulate the kinetics of ion currents cross cell membrane under the standard patch protocol and automatically plot the current – voltage relationship (I-V) curve that represents the electrical properties of ion channels (Fig. 9A). The simulation system also permits users to freely define their patch-clamp protocol with sine/cosine or stochastic signals and enriches the experimental means that is hard to perform in real biological environment.

(3) Action potential duration (APD) restitution curve

It is known that the steepness of APD restitution curve plays a critical role in analyzing stability of cell models. Therefore, there is a built-in algorithm to compute APD restitution curve, as depicted in Figure 9B. Usual protocols such as the S1-S2 protocol and the dynamic protocol have been programmed in the software to investigate the relationship between APDs and diastole intervals. In the simulation system, customers operate the mouse to capture the measured APDs, and then the APD restitution curve can be characterized automatically.

(4) Hypothesized path-physiological study of cells

The underlying path-physiological mechanism of cells is electrical and mechanical dysfunctions of ion channels. The simulation system can model alternations of ion channels by blocking ion channels thoroughly or partially, changing intercellular or extracellular ion concentrations and remodeling activation or inactivation kinetics of ion channels. The customers can do some drug hypothesized experiments regarding the changes of ion channels under physiological and pathological conditions.

(5) Extensible applications

It provides a useful function of editing or updating cell models by an extensible application program *MCML Editor*. Cell biologists and computational modelers do not need write any code and just do some simple operations by mouses and keyboards to build their cell models through the *MCML Editor,* as illustrated in Figure 10. There is another extensible application program the *Convertor Engine* modules, which automatically translates cell models into executed Java codes.

## 4. Discussion and conclusions

In this study, the simulation system has been validated effectively by models of different cell types that include the LR model, the PB model, the TNNP model and the Zhang model. Most of cell models were developed by a combination of independent ion channels and each ion channel was modeled by some dependent mathematical components which are very similar to the first mathematical cell model Hodgkin and Huxley model. The cell models share some common features in the structure of mathematical equations, and the methods to derive model parameters from experimental data. (in Table 1).Therefore, the simulation system is designed for cell biologists and mathematical modelers and it is able to construct computational cell models in a user-friendly manner and validate their models with simulated or experimental data. The simulation system is programmed by Java and is compatible to operation systems of both Windows series (WindowsXP, Vista and Windows 7) and Linux series (Redhat and Ubuntu). E-cell (Tomita et al., 1997) and Virtual Cell (Loew & Schaff, 2001; Slepchenko et al., 2003) are two popular computer software environments for cell modeling and simulating. Compared with them, our platform supports some characterized functions for analyzing electrical activities of computational cells such as automatically plotting the I-V curve and semi-automatically drawing the APD restitution curve. Various cell models can be expressed in a unified pattern, which enables an easy and convenient way to incorporate new cell models into the system. The system also enables appending of new functionalities or numerical methods dynamically and unlimitedly, and reselecting or switching numerical solvers at run-time casually without changing original programs and recompiling. In conclusion, the simulation system provides a powerful tool to study the electrical activities of cells and the ionic mechanism(s) underlying the genesis of cardiac action potentials.

## References

Crampin, E.J., Halstead, M., Hunter, P., Nielsen, P., Noble, D., Smith, N., & Tawhai, M. (2004). *Computational physiology and the Physiome Project. Exp Physiol,* 89, pp. 1-26.

Gamma. E., Helm. R., Johnson.R. , Vlissides. J.M. (1994). *Design Patterns: Elements of Reusable Object-Orient Software.,* (1st ed.). Addison Wesley. (Chapter 3 and 4).

HODGKIN, A.L. & HUXLEY, A.F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol,* 117, pp. 500-544.

Loew, L.M. & Schaff, J.C. (2001). The Virtual Cell: a software environment for computational cell biology. *Trends Biotechnol,* 19, pp. 401-406.

Luo, C.H. & Rudy, Y. (1994). A dynamic model of the cardiac ventricular action potential. I. Simulations of ionic currents and concentration changes. Circ Res, 74, pp. 1071-1096.

Moraru, I.I., Schaff, J.C., Slepchenko, B.M., & Loew, L.M. (2002). The virtual cell: an integrated modeling environment for experimental and computational cell biology. Ann N Y Acad Sci, 971, pp. 595-596.

Noble, D. (2006). *Systems biology and the heart. Biosystems,* 83, pp. 75-80.

Priebe, L. & Beuckelmann, D.J. (1998). Simulation study of cellular electric properties in heart failure. *Circ Res,* 82, pp. 1206-1223.

Rush, S. & Larsen, H. (1978). A practical algorithm for solving dynamic membrane equations. *IEEE Trans Biomed Eng,* 25, pp. 389-392.

Slepchenko, B.M., Schaff, J.C., Macara, I., & Loew, L.M. (2003). Quantitative cell biology with the Virtual Cell. *Trends Cell Biol,* 13, pp. 570-576.

ten Tusscher, K.H., Noble, D., Noble, P.J., & Panfilov, A.V. (2004). A model for human ventricular tissue. *Am J Physiol Heart Circ Physiol,* 286, pp. H1573-89.

Tomita, M., Hashimoto, K., Takahashi, K., Shimizu, T., Matsuzaki, Y., Miyoshi, F., Saito, K., Tanida, S., Yugi, K., Venter, J.C., & Hutchison, C.A. (1997). E-CELL: Software Environment for Whole Cell Simulation. *Genome Inform Ser Workshop Genome Inform,* 8, pp. 147-155.

Wang KQ, Yang GS, Yuan YF, Zhang HZ. (2007). MCML: an XML-based Modeling Language for Myocardial Cell Electrophysiology. Proceedings of the International Conference on Life System Modeling and Simulation, pp. 392-395.

Zhang, H., Holden, A.V., Kodama, I., Honjo, H., Lei, M., Varghese, T., & Boyett, M.R. (2000). Mathematical models of action potentials in the periphery and center of the rabbit sinoatrial node. *Am J Physiol Heart Circ Physiol,* 279, pp. H397-421.

Molleman A. (2003). Patch clamping: an introductory guide to patch clamp electrophysiology. West Sussex: John Wiley & Sons Ltd.

Table 1. An example of unified description about several typical cell models with the Object-Oriented method in the simulation system

| Cell model | LR model | PB model | TNNP model | Zhang model |
|---|---|---|---|---|
| ◆ Current | $I_{Na}$ | $I_{Na}$ | $I_{Na}$ | $I_{Na}$ |
| ▭ $[Na]_o$ | 135 mMol/L | 135 mMol/L | 140 mMol/L | 140 mMol/L |
| ▭ $[Na]_i$ | 15 mMol/L | 15 mMol/L | variable | 8 mMol/L |
| ♦ $E_{Na}$ | 2.1972 | 2.1972 | variable | 2.8622 |
| ▲ $G_{Na}$ | 16 nS/pF | 16 nS/pF | 14.838 nS/pF | 1.85 nS/pF |
| ⧗ m | Yes | Yes | Yes | Yes |
| ⧗ h | Yes | Yes | Yes | Yes |
| ⧗ j | Yes | Yes | Yes | No |
| ☷ MathML | $G_{Na}*m^3*h*j*(V-E_{Na})$ | $G_{Na}*m^3*h*j*(V-E_{Na})$ | $G_{Na}*m^3*h*j*(V-E_{Na})$ | $G_{Na}*m^3*h*[Na]_o*V*[(e^{(V-E_{Na})*F/RT}-1)/(e^{VF/RT}-1)]$ |

◆, Gated current; ▭, Concentration; ♦, Electrical potential; ▲, Conductance; ⧗, Gated variable; ☷, MathML equation.

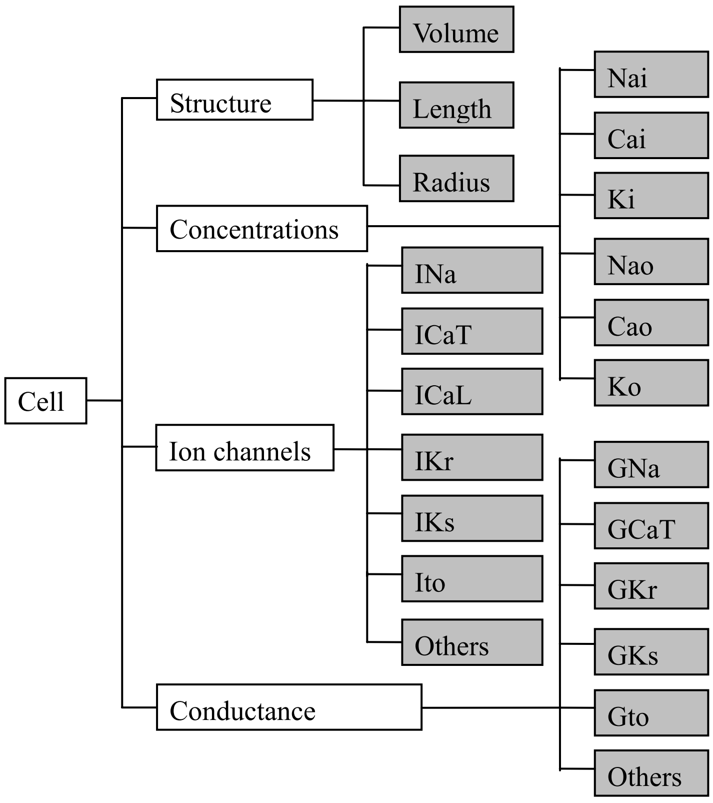


Figure 1. Abstract Architecture of Cell Models

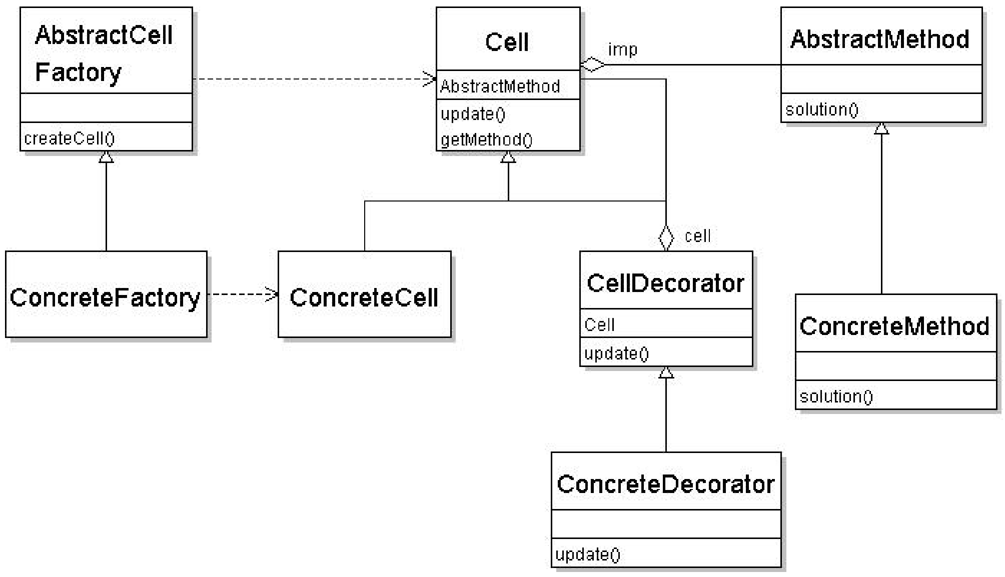


Figure 2. The Software Architecture of Cell Models Based on Object-Oriented Design Patterns

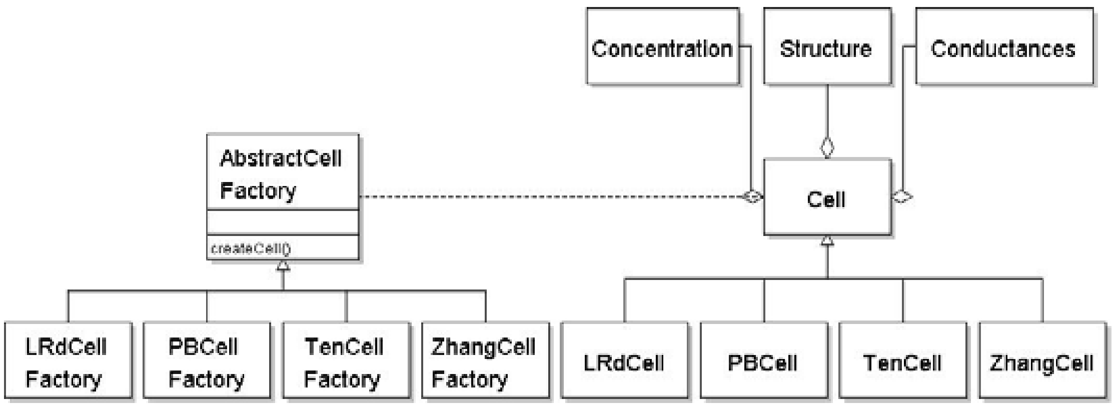


Figure 3. Creating Concrete Instances for Different Cell Models with the *Abstract Factory Pattern*

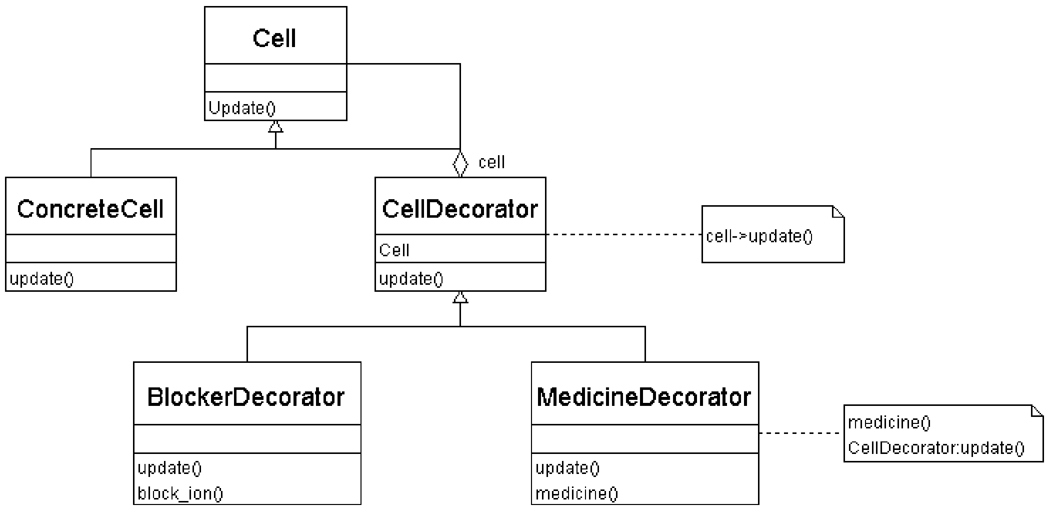


Figure 4. Using the *Decorator Pattern* to Extend Additional Functions of *Cell* Objects

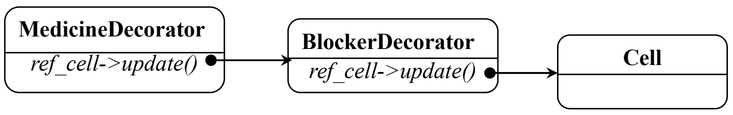


Figure 5. Nesting Two Decorator Objects to the Original Cell Object

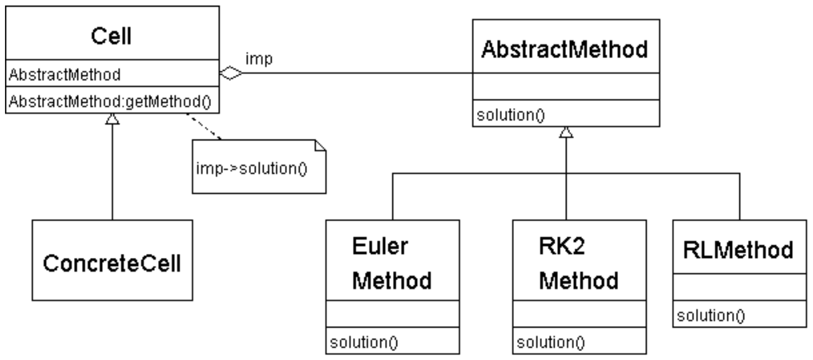


Figure 6. The *Bridge Pattern* to Support Multiple Numerical Method Implementations

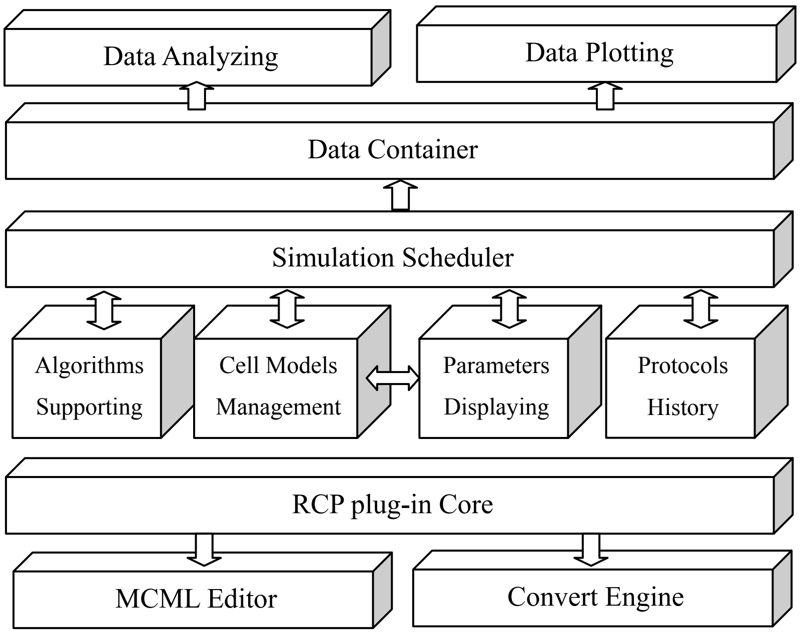
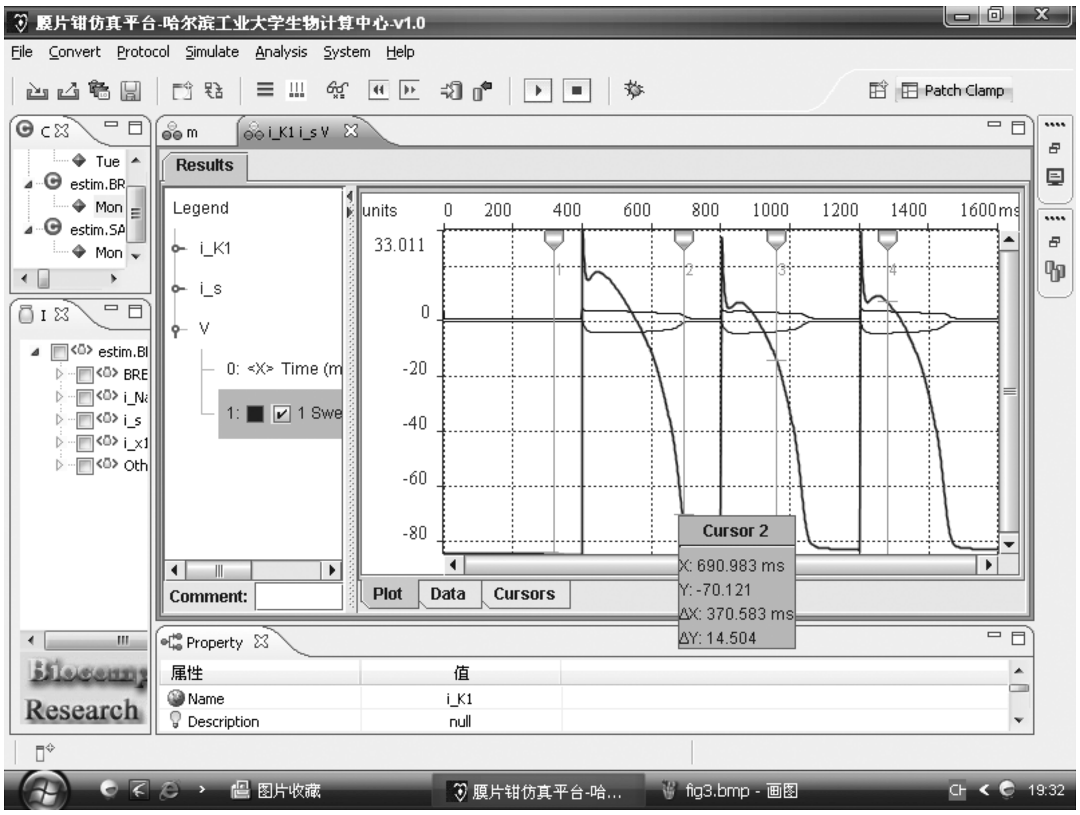
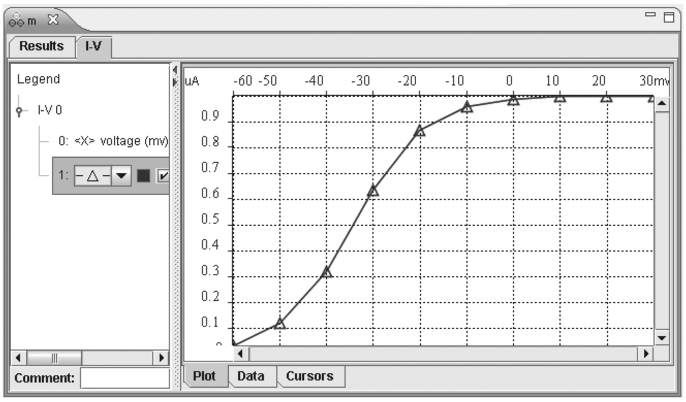


Figure 7. The Framework of the Simulation System

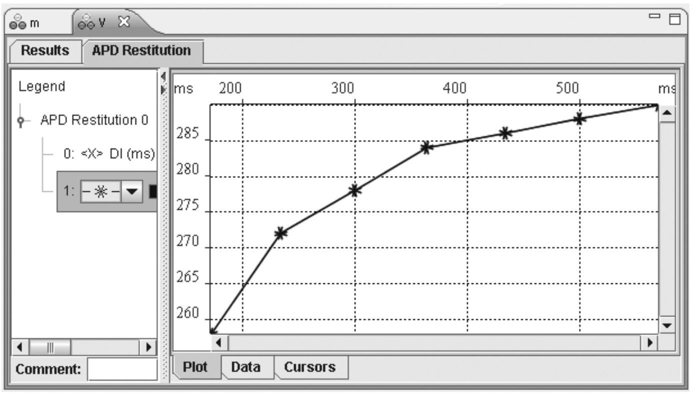


Cursor 2
X: 690.983 ms
Y: -70.121
ΔX: 370.583 ms
ΔY: 14.504

Figure 8. A Snapshot of the Simulation System

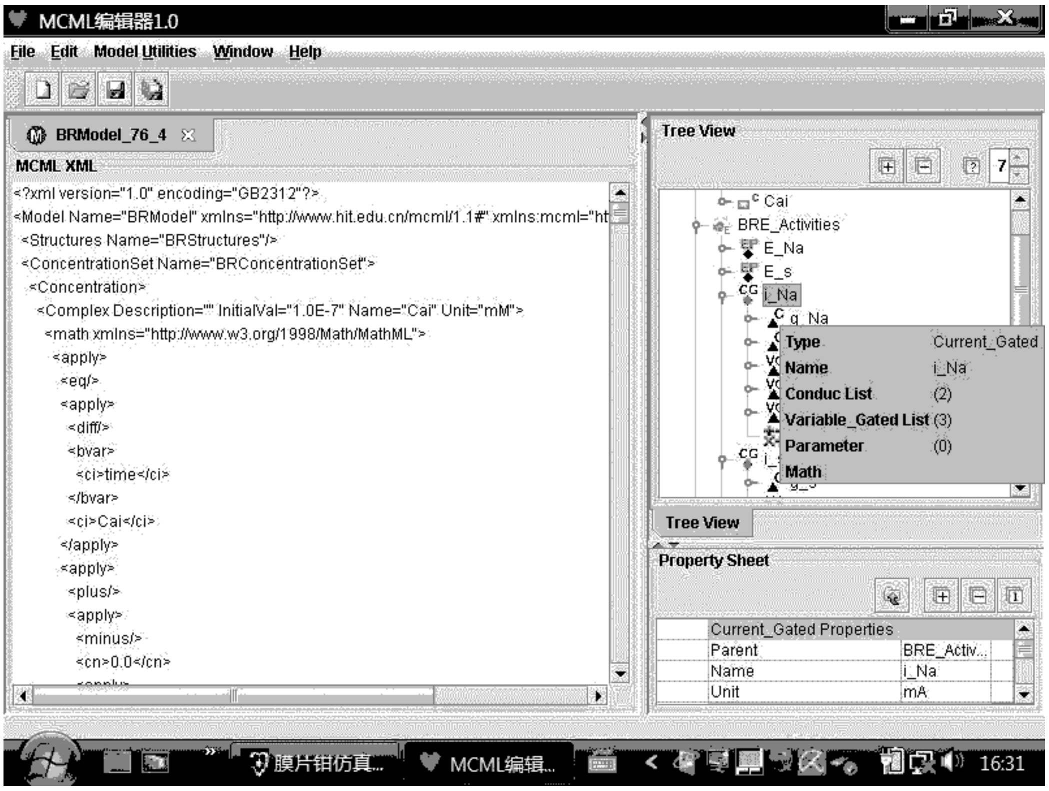


a) I-V relationship curve



b) APD restitution curve

Figure 9. Simulated I-V Relationship Curve and APD Restitution Curve for Cell Model Study



```
<?xml version="1.0" encoding="GB2312"?>
<Model Name="BRModel" xmlns="http://www.hit.edu.cn/mcml/1.1#" xmlns:mcml="ht
 <Structures Name="BRStructures"/>
 <ConcentrationSet Name="BRConcentrationSet">
  <Concentration>
   <Complex Description="" InitialVal="1.0E-7" Name="Cai" Unit="mM">
    <math xmlns="http://www.w3.org/1998/Math/MathML">
     <apply>
      <eq/>
      <apply>
       <diff/>
       <bvar>
        <ci>time</ci>
       </bvar>
       <ci>Cai</ci>
      </apply>
      <apply>
       <plus/>
       <apply>
        <minus/>
        <cn>0.0</cn>
```

```
Cai
BRE_Activities
  E_Na
  E_s
  i_Na
    g_Na
```

| | |
|---|---|
| Type | Current_Gated |
| Name | i_Na |
| Conduc List | (2) |
| Variable_Gated List | (3) |
| Parameter | (0) |
| Math | |

| Current_Gated Properties | |
|---|---|
| Parent | BRE_Activ... |
| Name | i_Na |
| Unit | mA |

Figure 10. A Snapshot of the MCML Editor