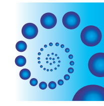
Punzalan *et al. Source Code for Biology and Medicine* 2012, **7**:11
http://www.scfbm.org/content/7/1/11



SOURCE CODE FOR
BIOLOGY AND MEDICINE

RESEARCH Open Access

A CellML simulation compiler and code generator using ODE solving schemes

Florencio Rusty Punzalan[1*], Yoshiharu Yamashita[2], Naoki Soejima[1], Masanari Kawabata[1], Takao Shimayoshi[3], Hiroaki Kuwabara[2], Yoshitoshi Kunieda[2] and Akira Amano[1]

Abstract

Models written in description languages such as CellML are becoming a popular solution to the handling of complex cellular physiological models in biological function simulations. However, in order to fully simulate a model, boundary conditions and ordinary differential equation (ODE) solving schemes have to be combined with it. Though boundary conditions can be described in CellML, it is difficult to explicitly specify ODE solving schemes using existing tools. In this study, we define an ODE solving scheme description language-based on XML and propose a code generation system for biological function simulations. In the proposed system, biological simulation programs using various ODE solving schemes can be easily generated. We designed a two-stage approach where the system generates the equation set associating the physiological model variable values at a certain time t with values at $t + \Delta t$ in the first stage. The second stage generates the simulation code for the model. This approach enables the flexible construction of code generation modules that can support complex sets of formulas. We evaluate the relationship between models and their calculation accuracies by simulating complex biological models using various ODE solving schemes. Using the FHN model simulation, results showed good qualitative and quantitative correspondence with the theoretical predictions. Results for the Luo-Rudy 1991 model showed that only first order precision was achieved. In addition, running the generated code in parallel on a GPU made it possible to speed up the calculation time by a factor of 50. The CellML Compiler source code is available for download at http://sourceforge.net/projects/cellmlcompiler.

Introduction

In recent years, the continued development in computer processing power paved the way for the increased use of biological function simulation. Computers have proven to be invaluable in analysing complex and nonintuitive biological models and biologists are turning to them to complement their experiments. Simulations enable the testing of experimentally unfeasible scenarios and can potentially reduce experimental costs. However, the number and complexity of physiological models has also grown with the increase in computing performance. This creates challenges in reproducing simulated behaviours of the published models and reuse of models by other researchers, hindering the dissemination of science and knowledge integration.

One way to address model complexity is to use markup language-based model descriptions. Some popular examples include CellML [1], SBML (Systems Biology Markup Language) [2] and insilicoML [3]. CellML is an open standard markup language capable of describing mathematical models of cellular functions. SBML is an open interchange machine-readable format for representing models of functions such as metabolism and cell signalling. Meanwhile, insilicoML describes mathematical models for biophysical objects and incorporates morphological information such as shape, angle and position. SED-ML (Simulation Experiment Description Language) [4] is another type of description language which can encode the information of simulation experiments. These markup languages allow researchers to take advantage of the vast amount of biological function models using a common set of easily readable and versatile description rules.

Biological and physiological function models are generally described by differential equations. A typical simulation of biological function models consists of three parts:

*Correspondence: floren@fc.ritsumei.ac.jp
[1]Graduate School of Life Sciences, Ritsumeikan University, Shiga, Japan
Full list of author information is available at the end of the article

BioMed Central

© 2012 Punzalan et al.; licensee BioMed Central Ltd. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

a model equation, a boundary condition, and an ordinary differential equation (ODE) solver. Model equations and boundary conditions can be described using CellML, while ODE numerical solutions like Euler and Runge-Kutta methods are typically built into the simulation software. However, it is necessary to be flexible in using ODE schemes in order to strike a balance between computational stability and speed. In addition, those using special hardware environments such as massively parallel computer systems require dedicated proprietary software to support their numerical solution needs. Thus, description languages like CellML and dedicated simulation software are not suitable or practical for flexibly incorporating different ODE solving schemes.

To address the need for more flexibility in creating simulation software, we created Time Evolution Calculation Markup Language (TecML), a machine-readable format for encoding ODE numerical solutions. TecML is a description language based on the extensible markup language (XML). This description language is designed to specify and store the numerical methods that can be used for solving the ODEs in biological models. It also allows the assignment of boundary conditions into the simulation experiments. The following sections describe TecML and how it is integrated into the proposed code generation system, which automatically generates codes for biological simulations.

The target of this study is limited to the use of different ODE numerical solutions and their application to models described in CellML. We propose an algorithm that allows users to change the ODE solution and boundary conditions of the model according to the computational needs of their simulation. To verify the effectiveness of the proposed system, we generate executable codes for several CellML models using a number of ODE numerical solutions. The system can generate code in several programming languages and code that runs in both sequential and parallel computing environments. Simulations on GPU (Graphics Processing Units) were undertaken to show the effect of using parallel computing on processing time.

Biological simulation code generation system

Summary of simulation code generation system

The proposed method is composed of two stages (Figure 1). In the first stage, the system represents the biological model by incorporating an ODE numerical solution method into the model's differential equations. This creates the equation sets that calculate the time evolution of the mathematical model. The second stage generates the simulation code for these sets of equations, allowing the user to run computer simulations of the model in machines with general-purpose compilers. This approach enables the flexible construction of code generation modules that can support complex sets of equations. By generating the code separately for each section of the mathematical model, parallel code execution can be easily integrated into the simulation.

To illustrate the capabilities of the proposed system, we used the FitzHugh-Nagumo (FHN) excitable media model [5] as an example. The model, proposed by R. FitzHugh in 1961 and which J. Nagumo et al. created an equivalent circuit, is a simplification of the Hodgkin-Huxley model and can be described by the following equations:

$$r = x^3, \tag{1}$$

$$dx/dt = x - r/3.0 - y + a, \tag{2}$$

$$dy/dt = b(x + c - d \cdot y), \tag{3}$$

where r, x and y are variables, t indicates time and a, b, c, d are constants. Removable algebraic expressions such as equation (1) are often used in biological models to improve the model's readability.

As for the ODE numerical solution, we used the Modified Euler method to solve the ODEs in (2) and (3). By combining a mathematical description of the model above with the ODE solution method and boundary conditions, we can derive the formula to calculate the time evolution of the independent variables x and y. This involves finding the solution to the variable vector $\boldsymbol{\xi}$ using its derivatives, temporal variable vector $\boldsymbol{\iota}$, and initial conditions at time t given by

$$d\boldsymbol{\xi}/dt = \boldsymbol{f}(t, \boldsymbol{\xi}, \boldsymbol{\iota}), \tag{4}$$

$$\boldsymbol{\iota} = \boldsymbol{g}(t, \boldsymbol{\xi}). \tag{5}$$

The rest of this section shows how the proposed system incorporates the Modified Euler method into the model equations to calculate for $\boldsymbol{\xi}$. The following set of equations represent the numerical solution with boundary conditions:

$$\boldsymbol{\xi_0} = \boldsymbol{\xi_t}, \tag{6}$$

$$t_0 = t, \tag{7}$$

$$\boldsymbol{\iota_0} = \boldsymbol{g}(t_0, \boldsymbol{\xi}_0), \tag{8}$$

$$\boldsymbol{\kappa_1} = \boldsymbol{f}(t_0, \boldsymbol{\xi}_0, \iota_0), \tag{9}$$

$$\boldsymbol{\xi}_1 = \boldsymbol{\xi_0} + \boldsymbol{\kappa}_1 \cdot \delta, \tag{10}$$

$$t_1 = t_0 + \delta, \tag{11}$$

$$\boldsymbol{\iota_1} = \boldsymbol{g}(t_1, \boldsymbol{\xi}_1), \tag{12}$$

$$\boldsymbol{\kappa_2} = \boldsymbol{f}(t_1, \boldsymbol{\xi}_1, \iota_1), \tag{13}$$

$$\boldsymbol{\xi}_2 = \boldsymbol{\xi}_0 + \frac{1}{2}(\boldsymbol{\kappa}_1 + \boldsymbol{\kappa}_2) \cdot \delta, \tag{14}$$

$$\boldsymbol{\xi}_{t+\delta} = \boldsymbol{\xi}_2, \tag{15}$$

where $\boldsymbol{\xi}$, $\boldsymbol{\kappa}$ and $\boldsymbol{\iota}$ are the vectors representing the variables and differential equations in the biological model.

By applying the ODE solver described above, we can arrive at the set of equations detailing the numerical solution for the initial value problem of the FHN model.

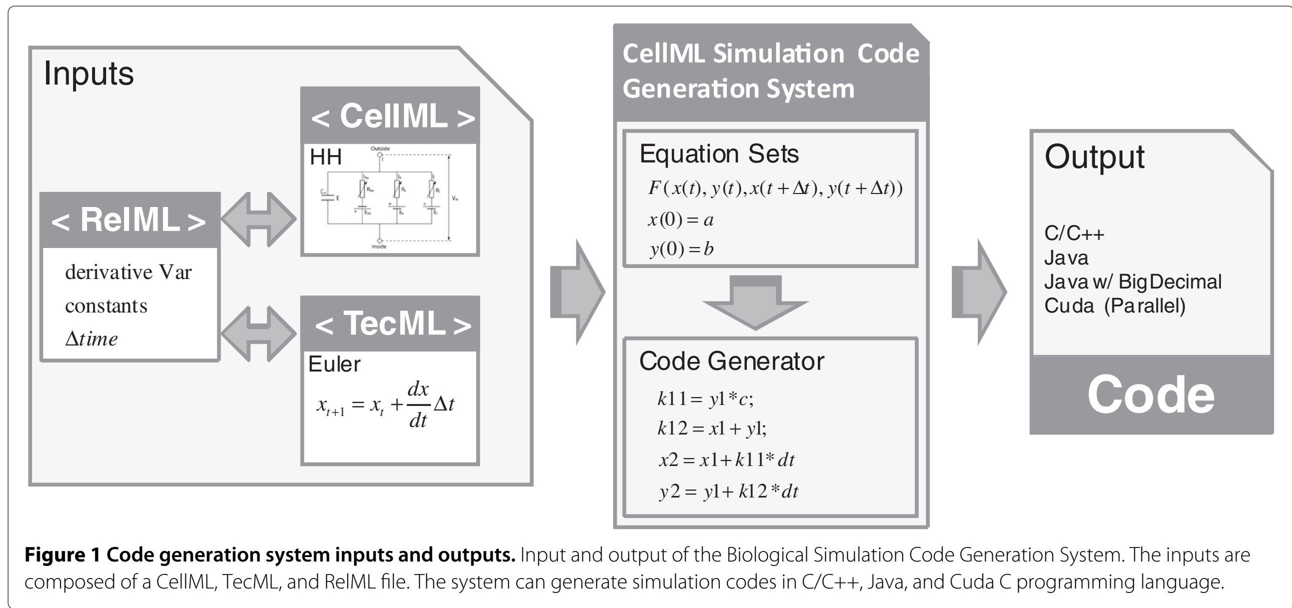


Figure 1 Code generation system inputs and outputs. Input and output of the Biological Simulation Code Generation System. The inputs are composed of a CellML, TecML, and RelML file. The system can generate simulation codes in C/C++, Java, and Cuda C programming language.

First, from equations (6) and (7), the current values of the differential variables at t are assigned as the initial values with

$$x_0 = x_t, \tag{16}$$
$$y_0 = y_t, \tag{17}$$
$$t_0 = t. \tag{18}$$

Then with equations (8) and (9), the differential and nondifferential equations are expanded and evaluated using these initial values to obtain

$$r_0 = g_1(t_0, [x_0\ y_0]^T) = x_0^3, \tag{19}$$
$$\kappa_{1,x} = f_1(t_0, [x_0\ y_0]^T, r_0) = x_0 - r_0/3.0 - y_0 + a, \tag{20}$$
$$\kappa_{1,y} = f_2(t_0, [x_0\ y_0]^T, r_0) = b(x_0 + c - d \cdot y_0). \tag{21}$$

Next, the new values of x and y are computed as functions of $\boldsymbol{\kappa}_1$ and δ (equations (10) and (11)) and given by

$$x_1 = x_0 + \kappa_{1,x} \cdot \delta, \tag{22}$$
$$y_1 = y_0 + \kappa_{1,y} \cdot \delta, \tag{23}$$
$$t_1 = t_0 + \delta. \tag{24}$$

This process is repeated for r_1 and $\boldsymbol{\kappa}_2$ as shown in equations (12) and (13);

$$r_1 = g_1(t_1, [x_1\ y_1]^T) = x_1^3, \tag{25}$$
$$\kappa_{2,x} = f_1(t_1, [x_1\ y_1]^T, r_1) = x_1 - r_1/3.0 - y_1 + a, \tag{26}$$
$$\kappa_{2,y} = f_2(t_1, [x_1\ y_1]^t, r_1) = b(x_1 + c - d \cdot y_1). \tag{27}$$

Finally, the value of the differential variables are obtained by advancing the solution from time t to $t + \delta$ (equations (14) and (15)) with

$$x_2 = x_0 + \frac{1}{2}(\kappa_{1,x} + \kappa_{2,x}) \cdot \delta, \tag{28}$$
$$y_2 = y_0 + \frac{1}{2}(\kappa_{1,y} + \kappa_{2,y}) \cdot \delta, \tag{29}$$
$$x_{t+\delta} = x_2, \tag{30}$$
$$y_{t+\delta} = y_2. \tag{31}$$

The variable r of the equation set corresponds to $\boldsymbol{\iota}$ while x and y are the variables in the vector $\boldsymbol{\xi}$. The derivatives dx/dt and dy/dt are expressed by $\boldsymbol{\kappa}$ in equations (20), (21), (26) and (27).

In order to automatically incorporate the ODE numerical solution into the CellML model, we introduced an ODE solving scheme description language called Time Evolution Calculation Markup Language (TecML). TecML is an XML-based description language that can be used to configure ODE solutions and implement the algorithm described in equations (6) – (15). Another input of the system is the Relation Markup Language (RelML) file. The basic role of a RelML file is to relate the variables and variable types of the CellML file into their equivalent in the TecML file.

The simulation code generation system uses these inputs to generate the set of equations describing the time evolution of the variables in biological models. This step creates equations (16) – (31) in the FHN example. Once the model equations are applied with the ODE numerical solution method, the next stage involves the generation of the executable code that will do the actual calculations.

Description language

The code generation system requires three inputs, namely, the CellML file, TecML file and RelML file. This section gives a short introduction of each and how it is used in the algorithm and consequently, in the FHN model example.

CellML model encoding standard

The CellML [1] language is an open standard based on XML for describing mathematical models. It was designed to allow scientists to share models even if they are using different model-building software. The majority of the models stored in its repository are cell representations and these include information about the cell structure, equations for underlying processes and in some cases, boundary conditions. Figure 2 shows how a CellML model describes the mathematical equations of the FHN cell model. The file lists the input and output variables as well as their initial values and underlying equations. Note that compact syntax representation is used to show its contents and is based on Alan Garny's notation for CellML in the Cellular Open Resource (COR) software [6].

TecML: an ODE solving scheme description language

TecML (Time Evolution Calculation Markup Language) is an XML-based description language designed to describe ODE numerical solutions that can be used in biosimulations. This proposed standard allows the mathematical description of solving schemes like the Euler and Runge-Kutta methods. It allows integration of numerical methods with other description languages like CellML. TecML categorizes variables into six different types (Table 1) namely, *diffvar*, *derivativevar*, *arithvar*, *constvar*, *timevar*, and *deltatimevar*. The variables determined by a rate of change with respect to time ($\boldsymbol{\xi}$) are referred to as differential variables (*diffvar*) while their derivatives ($\boldsymbol{\kappa}$) are called derivative variables (*derivativevar*). The removable variables ($\boldsymbol{\iota}$) are the arithmetic variables (*arithvar*) and variables that do not change in value ($\boldsymbol{\zeta}$) are the constants (*constvar*). In addition, the time (t) and time increment (δ) are referred to as *timevar* and *deltatimevar*, respectively. TecML also divides the mathematical equations into two types; namely, differential ($\boldsymbol{f}()$) and non-differential ($\boldsymbol{g}()$) equations. Equations of type (*diffequ*) are the derivatives of a function while (*nondiffequ*) are the arithmetic functions. The information and example of a TecML file for the Modified Euler method are shown in Figure 3 and Figure 4, respectively.

Table 1 Variable and function types in TecML

Type	Definition
diffvar	Differential variable (variable is a function of time)
arithvar	Temporal variable (can be substituted with math equations)
derivativevar	Derivative of diffvar
constvar	Constants
timevar	Time variable
deltatimevar	Variable denoting the change of time per step
diffequ	Differential equation
nondiffequ	Non-differential equation

```
def model fhn as
   def comp Main as
      var t: {init: 0};
      var r: {init: 0};
      var x: {init: -1.501250563778375};
      var y: {init: -0.376213677498469};
      var a: {init: 0};
      var b: {init: 0.03};
      var c: {init: 1.2};
      var d: {init: 0.3};

         r = x^3;
      dx/dt = x - r/3.0 - y + a;
      dy/dt = b(x + c - dy);
      enddef;
enddef;
```

Figure 2 Information in the CellML file of the FHN model (represented in Alan Garny's COR notation).

```
def method ModifiedEuler as
   def variables odesolvar as
      vartype input: {ξ_i};
      vartype output: {ξ_o};
      vartype diffvar: {ξ_0, ξ_1, ξ_2};
      vartype derivativevar: {κ_1, κ_2};
      vartype arithvar: {ι_0, ι_1};
      vartype constvar: {ζ};
      vartype timevar: {τ};
   enddef;
   def equations odesolequ as
      equtype diffequ: {φ(ξ, ι, τ, ζ)};
      equtype nondiffequ: {γ(ξ, ι, τ, ζ)};
   enddef;

   ξ_0 = ξ_i;
   ι_0 = γ(ξ_0, ι_0, τ, ζ);
   κ_1 = φ(ξ_0, ι_0, τ, ζ);
   ξ_1 = ξ_0 + κ_1 * δ;
   ι_1 = γ(ξ_1, ι_1, τ + δ, ζ);
   κ_2 = φ(ξ_1, ι_1, τ + δ, ζ);
   ξ_2 = ξ_0 + (κ_1 + κ_2) * δ;
   ξ_o = ξ_2;
enddef;
```

Figure 3 Information in the TecML file of the Modified Euler method where ξ is the *diffvar*, κ is the *derivativevar*, ζ is the *constvar*, and variable type ι is the *arithvar*.

```
<tecml>
  <inputvar name="xi" type="diffvar" />
  <outputvar name="xo" type="diffvar" />
  <variable name="y1" type="arithvar" />
  <variable name="k1" type="derivativevar" />
  <function name="f" type="diffequ">
  ...
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <apply>
      x0 = xi (in MathML code)
    </apply>
    ...
  </math>
</tecml>
```

Figure 4 **TecML file example.**

Relation Markup Language (RelML)

RelML is a language for describing the correspondence between the variables in the CellML model file and the variable types in an ODE numerical solution scheme described in a TecML file. Figure 5 shows the correspondence of variables described in the FHN CellML file (Figure 2) and the variable types in the TecML file (Figure 3). For the FHN model in Figure 2, variables x and y are defined as *diffvar* and their respective derivatives dx/dt and dy/dt as *derivativevar*. In addition, a, b, c, and d are *constvar* type while r is considered a temporal variable or *arithvar* type. Equations (2) and (3) are the formulas for calculating the *diffvar* so the functions are categorized as *diffequ* while the arithmetic equation for r in equation (1) is a *nondiffequ*.

An example RelML file where the Modified Euler method is applied to the FHN model is shown in Figure 6. The first two statements after the header indicate the filename and location of the corresponding CellML and TecML file. The succeeding lines enumerate all the variables used in the FHN model and their corresponding types. The complete RelML and TecML files used in this paper have been published on the Web [7].

```
def connection fhn_ModifiedEuler as
   def cellml fhn as
      filename "fhn.cellml";
   enddef;
   def tecml ModifiedEuler as
      filename "ModifiedEuler.tecml";
   enddef;
   def variables cellmlvar as
      vartype diffvar: {x, y};
      vartype arithvar: {r};
      vartype constvar: {a, b, c, d};
   enddef;
enddef;
```

Figure 5 **Information contained in the RelML file for the FHN model and Modified Euler method.**

```
<relml>
  <cellml filename="model/cellml/fhn.cellml"/>
  <tecml filename="model/tecml/ModifiedEuler.tecml"/>

  <variable name="time" type="timevar" />
  <variable name="x" type="diffvar" />
  <variable name="y" type="diffvar" />
  <variable name="r" type="arithvar" />
  <variable name="a" type="constvar" />
  <variable name="b" type="constvar" />
  <variable name="c" type="constvar" />
  <variable name="d" type="constvar" />
</relml>
```

Figure 6 **RelML file example for FHN model and Modified Euler method.**

Executable code generation for equation sets

The system implementation of the code generator allows the creation of codes in a number of programming languages. Table 2 lists the tested inputs and outputs of the system. For a desktop CPU environment, the generator can produce source codes in C, Java, and Java with BigDecimal library, which can handle higher precision computing. Codes can be generated for both single cell and cell array simulation. The cell array code generator can produce both 1D and 2D excitation propagation codes. The generated codes for 2D simulation describes the excitation propagation in a rectangular array with $N \times M$ number of cells. Aside from generating single CPU codes, it can also create codes suited for parallel computing that runs on a GPU machine.

Methods: simulation code generation algorithm

Cell model description

The cell model is a collection of variables and equation and can be written as

$$\begin{cases} \boldsymbol{k} = d\boldsymbol{x}/dt = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}, t, \boldsymbol{z}) \\ \boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{y}, t, \boldsymbol{z}) \end{cases}, \tag{32}$$

where $\boldsymbol{x}$ is a differential variables vector, $\boldsymbol{k}$ is a derivatives vector, $\boldsymbol{y}$ is a temporal variables vector, and $\boldsymbol{z}$ is a constants vector while t is the scalar variable for time. These variable vectors can be expressed as sets with

$$\boldsymbol{x} = [x_1, x_2, \cdots, x_{N_x}]^T, \tag{33}$$

$$\boldsymbol{k} = [k_1, k_2, \cdots, k_{N_x}]^T, \tag{34}$$

$$\boldsymbol{y} = [y_1, y_2, \cdots, y_{N_y}]^T, \tag{35}$$

$$\boldsymbol{z} = [z_1, z_2, \cdots, z_{N_z}]^T, \tag{36}$$

Table 2 Input and output code types in the system

Cell Physiology	FitzHugh-Nagumo model [5]
Model	LuoRudy 1991 model [8], Kyoto 2003 Model [9]
ODE Solution	Euler Method, Modified Euler Method
Scheme	1^{st}-, 2^{nd}-, 4^{th}-order Runge-Kutta Method
Generated codes	C code, Java, Java with BigDecimal
	Cuda code (GPU)

where N_x, N_y, N_z are the variable count for $\boldsymbol{k}$, $\boldsymbol{y}$, and $\boldsymbol{z}$, respectively. Furthermore, $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z})$ and $\boldsymbol{g}(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z})$ are function vectors with $k_i = f_i(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z})$ and $y_i = g_i(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z})$. These function vectors are defined by

$$\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z}) = [f_1(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z}),\cdots,f_{N_x}(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z})]^T, \quad (37)$$

$$\boldsymbol{g}(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z}) = [g_1(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z}),\cdots,g_{N_y}(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z})]^T. \quad (38)$$

ODE solving scheme

Table 3 shows the elements inside a TecML file. Note that all the TecML variables are denoted with Greek letters. The differential equations for the cell model are represented by the terms $d\boldsymbol{\xi}/dt = \boldsymbol{\Phi}(\boldsymbol{\xi},\boldsymbol{\iota},t,\boldsymbol{\zeta})$ and $\boldsymbol{\iota} = \boldsymbol{\Gamma}(\boldsymbol{\xi},\boldsymbol{\iota},t,\boldsymbol{\zeta})$. Inside a TecML file, the dependence between the differential variables $\boldsymbol{\xi_0}$ (time t) and $\boldsymbol{\xi}_{N_\xi}$ (time $t+\delta$) is given by

$$\boldsymbol{\xi}_i = \boldsymbol{\sigma}_i(\boldsymbol{\Xi},\boldsymbol{\Pi},\delta) \quad (0 \le i \le N_\xi), \quad (39)$$

$$\boldsymbol{\kappa_{i+1}} = \boldsymbol{\Phi}(\boldsymbol{\xi}_i,\boldsymbol{\iota}_i,\tau_i,\boldsymbol{\zeta}) \quad (0 \le i \le N_\xi), \quad (40)$$

$$\boldsymbol{\iota}_i = \boldsymbol{\Gamma}(\boldsymbol{\xi}_i,\boldsymbol{\iota}_i,\tau_i,\boldsymbol{\zeta}) \quad (0 \le i \le N_\xi), \quad (41)$$

$$\tau_i = T_i(t,\delta) \quad (0 \le i \le N_\xi), \quad (42)$$

where the differential variable vector $\boldsymbol{\Xi}$ and derivative variable vector $\boldsymbol{\Pi}$ are given by

$$\boldsymbol{\Xi} = [\boldsymbol{\xi}_0,\boldsymbol{\xi}_1,\cdots,\boldsymbol{\xi}_{N_\xi}]^T, \quad (43)$$

$$\boldsymbol{\Pi} = [\boldsymbol{\kappa}_1,\boldsymbol{\kappa}_2,\cdots,\boldsymbol{\kappa}_{N_\xi}]^T. \quad (44)$$

CellML and TecML integration

The integration of the CellML model and TecML solving scheme involves the mapping of corresponding variables.

Table 3 Information written in a TecML file

Notation	Definition
$\boldsymbol{\xi}_0 = [\xi_{0,1},\xi_{0,2},\cdots,\xi_{0,N_x}]^T$	Input differential variable vector
$\boldsymbol{\xi}_{N_\xi} = [\xi_{N_\xi,1},\xi_{N_\xi,2},\cdots,\xi_{N_\xi,N_x}]^T$	Output differential variable vector
$\boldsymbol{\xi}_i (1 \le i \le N_\xi)$	Time differential variable vector
$\boldsymbol{\Xi} = [\boldsymbol{\xi}_0,\boldsymbol{\xi}_1,\cdots,\boldsymbol{\xi}_{N_\xi}]^T$	Differential variable vector of $\boldsymbol{\xi}$
t	Current time value
δ	Time step
$\boldsymbol{\zeta} = [\zeta_0,\zeta_1,\cdots,\zeta_{N_z}]^T$	Constants vector
$\tau = T_i(t,\delta)$	Calculation of variable T_i
$\boldsymbol{\Phi}(\boldsymbol{\xi},\boldsymbol{\iota},\tau,\boldsymbol{\zeta}) = [\phi_1(\boldsymbol{\xi},\boldsymbol{\iota},\tau,\boldsymbol{\zeta}), \cdots,\phi_{N_x}(\boldsymbol{\xi},\boldsymbol{\iota},t,\boldsymbol{\zeta})]^T$	Differential equation vector
$\boldsymbol{\Gamma}(\boldsymbol{\gamma},\boldsymbol{\iota},\tau,\boldsymbol{\zeta}) = [\gamma_1(\boldsymbol{\xi},\boldsymbol{\iota},\tau,\boldsymbol{\zeta}), \cdots,\gamma_{N_y}(\boldsymbol{\xi},\boldsymbol{\iota},t,\boldsymbol{\zeta})]^T$	Temporal function vector
$\boldsymbol{\kappa}_{i+1} = \boldsymbol{\Phi}(\boldsymbol{\xi},\boldsymbol{\iota},t,\boldsymbol{\zeta})$	Derivative variable vector
$\boldsymbol{\iota}_i = \boldsymbol{\Gamma}(\boldsymbol{\xi},\boldsymbol{\iota},t,\boldsymbol{\zeta})$	Temporal variable derived from $\boldsymbol{\xi}$
$\boldsymbol{\Pi} = [\boldsymbol{\kappa}_1,\boldsymbol{\kappa}_2,\cdots,\boldsymbol{\kappa}_{N_\xi}]^T$	Derivative vector of $\boldsymbol{\kappa}$
$\xi_i = \sigma(\Xi,K,\delta)$	Relation between ξ_i and Ξ, K, δ

The mapping shows how each physiological model variable written in CellML is replaced with its corresponding TecML variable. The differential variable vector $\boldsymbol{\xi}$ of TecML corresponds to the differential variable vector $\boldsymbol{x}$ of CellML and reads as

$$\boldsymbol{\xi}_k \leftarrow \boldsymbol{x}_k = [x_{k,1},x_{k,2}\cdots x_{k,N_x}]^T. \quad (45)$$

TecML's derivative variable $\boldsymbol{\kappa}$ equates to the CellML's derivative variable $d\boldsymbol{x}/dt$ and the temporal variable $\boldsymbol{\iota}$ corresponds to $\boldsymbol{y}$ and given by

$$\boldsymbol{\kappa}_{k+1} \leftarrow d\boldsymbol{x}_k/dt = \left[\frac{d\boldsymbol{x}_{k,1}}{dt},\frac{d\boldsymbol{x}_{k,2}}{dt},\cdots,\frac{d\boldsymbol{x}_{k,N_x}}{dt}\right]^T, \quad (46)$$

$$\boldsymbol{\iota}_k \leftarrow \boldsymbol{y}_k = [y_{k,1},y_{k,2},\cdots,y_{k,N_y}]^T. \quad (47)$$

Furthermore, the TecML equations $\boldsymbol{\kappa} = \boldsymbol{\Phi}(\boldsymbol{\xi},\boldsymbol{\iota},t,\boldsymbol{\zeta})$ and $\boldsymbol{\iota} = \boldsymbol{\Gamma}(\boldsymbol{\xi},\boldsymbol{\iota},t,\boldsymbol{\zeta})$ correspond to the CellML equations $\boldsymbol{f}$ and $\boldsymbol{g}$, respectively, as shown by

$$\boldsymbol{\Phi}(\boldsymbol{\xi},\boldsymbol{\iota},\tau,\boldsymbol{\zeta}) \leftarrow \boldsymbol{f}(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z}), \quad (48)$$

$$\boldsymbol{\Gamma}(\boldsymbol{\xi},\boldsymbol{\iota},\tau,\boldsymbol{\zeta}) \leftarrow \boldsymbol{g}(\boldsymbol{x},\boldsymbol{y},t,\boldsymbol{z}). \quad (49)$$

Replacement algorithm

Once the variables are mapped, the differential and arithmetic equations are transformed and expanded according to the numerical method described in the TecML file. First, the variables in the CellML model equations are replaced with their corresponding TecML variables. Each model equation is searched for any *diffvar*, *derivativevar*, *arithvar*, *constvar*, *timevar*, or *deltatimevar* and that variable is replaced with the corresponding TecML variable name. This replacement procedure is represented in Strachey brackets and expressed by the following function:

$$\boldsymbol{\mathcal{R}_v}[\![equ]\!] = \boldsymbol{\mathcal{R}_x}[\![\boldsymbol{\mathcal{R}_k}[\![\boldsymbol{\mathcal{R}_y}[\![\boldsymbol{\mathcal{R}_t}[\![\boldsymbol{\mathcal{R}_z}[\![\boldsymbol{\mathcal{R}_d}[\![equ]\!]]\!]]\!]]\!]]\!]]\!], \quad (50)$$

where $[\![equ]\!]$ is the TecML equation and, for all $i = 1,2,...,N_\xi$, the replacement functions are

$$\boldsymbol{\mathcal{R}_x}[\![\boldsymbol{\xi}_i]\!] = [\![\boldsymbol{x}_i]\!], \quad (51)$$

$$\boldsymbol{\mathcal{R}_k}[\![\boldsymbol{\kappa}_i]\!] = [\![\boldsymbol{k}_i]\!], \quad (52)$$

$$\boldsymbol{\mathcal{R}_y}[\![\boldsymbol{\iota}_i]\!] = [\![\boldsymbol{y}_i]\!], \quad (53)$$

$$\boldsymbol{\mathcal{R}_t}[\![\tau_i]\!] = [\![t_i]\!], \quad (54)$$

$$\boldsymbol{\mathcal{R}_z}[\![\boldsymbol{\zeta}]\!] = [\![\boldsymbol{z}]\!], \quad (55)$$

$$\boldsymbol{\mathcal{R}_d}[\![\delta]\!] = [\![d]\!]. \quad (56)$$

Note that the Strachey brackets in $\boldsymbol{\mathcal{R}_x}[\![\boldsymbol{\xi}_i]\!] = [\![\boldsymbol{x}_i]\!]$ means that all the TecML *diffvar* (*i.e.* for $i = 1 \ldots N_\xi$) in the argument is replaced with the corresponding CellML *diffvar* $\boldsymbol{x}_i$. This function is true for all the other variable types in TecML.

The generated set of equations advances the solution of the differential variables from time t to $t+\delta$.

Figure 7 shows how the algorithm generates the simulation program from the input files. Note that the subroutine `replace_v()` in the algorithm replaces variables in the CellML model equation with the TecML variables as expressed in equation (50). The function `replace_sj()` generates one scalar equation from a vector equation and appends index `j` to the variables. In the equations which do not contain functions f or g, `replace_d()` generates multiple scalar equations from a vector equation. Finally, `replace_f()` and `replace_g()` unfold functions f and g, respectively. These subroutines can be expressed in the following transformations:

$$\mathcal{R}_{s,j}[\![equ]\!] = \texttt{makeEquation}[\![\mathcal{S}_{y,j}[\![\mathcal{S}_{k,j}[\![\texttt{getLHS}[\![equ]\!]]\!]]\!], \mathcal{S}_{g,j}[\![\mathcal{S}_{f,j}[\![\texttt{getRHS}[\![equ]\!]]\!]]\!]]\!], \tag{57}$$

$$\mathcal{R}_{d,j}[\![equ]\!] = \mathcal{S}_{k,j}[\![\mathcal{S}_{x,j}[\![equ]\!]]\!], \tag{58}$$

$$\mathcal{R}_{f}[\![\phi_j(\boldsymbol{\xi}, \boldsymbol{\iota}, \tau, \boldsymbol{\zeta})]\!] = \mathcal{T}_{y,l}[\![\mathcal{T}_{x,k}[\![f_j(\boldsymbol{x}, \boldsymbol{y}, t, \boldsymbol{z})]\!]]\!], \tag{59}$$

$$\mathcal{R}_{g}[\![\gamma_j(\boldsymbol{\xi}, \boldsymbol{\iota}, \tau, \boldsymbol{\zeta})]\!] = \mathcal{T}_{y,l}[\![\mathcal{T}_{x,k}[\![g_j(\boldsymbol{x}, \boldsymbol{y}, t, \boldsymbol{z})]\!]]\!], \tag{60}$$

where the index-appending functions $\mathcal{S}$ and unfolding functions $\mathcal{T}$ are defined as

$$\mathcal{S}_{x,j}[\![\boldsymbol{x}_i]\!] = [\![x_{i,j}]\!], \tag{61}$$

$$\mathcal{S}_{y,j}[\![\boldsymbol{y}_i]\!] = [\![y_{i,j}]\!], \tag{62}$$

$$\mathcal{S}_{f,j}[\![\boldsymbol{\Phi}]\!] = [\![\phi_j]\!], \tag{63}$$

$$\mathcal{S}_{g,j}[\![\boldsymbol{\Gamma}]\!] = [\![\gamma_j]\!], \tag{64}$$

$$\mathcal{T}_{x,k}[\![\boldsymbol{x}]\!] = [\![\boldsymbol{x}_k]\!], \tag{65}$$

$$\mathcal{T}_{y,k}[\![\boldsymbol{y}]\!] = [\![\boldsymbol{y}_k]\!]. \tag{66}$$

Experiments and results

Code generation for the cell model simulation

The FHN cell model [5] was used to evaluate the proposed system. The system was also used to generate simulation codes for a number of cell models (LuoRudy1991 [8], LuoRudy1994 [10] and KyotoModel2003 [9]) and with other more accurate ODE numerical methods (e.g. 4th-order Runge-Kutta method). All the generated code used in the simulation experiments can be downloaded from the files section of the program generator site [7].

```
generate_equations(){
  Equation equation_set[] ← all equations in TecML;
  Equation equation_set2[] ← null;
  int equ2_count ← 0;
  for i ← 1 to length[equation_set] do
      equation_set[i] ← replace_v(equation_set[i]);
      if equation_set[i] includes Φ then
          for j ← 1 to N_x do
              equation_set2[equ2_count++] ←
                          replace_sj(equation_set[i], j);
          end
      else if equation_set[i] includes Γ then
          for j ← 1 to N_y do
              equation_set2[equ2_count++] ←
                          replace_sj(equation_set[i], j);
          end
      else
          for j ← 1 to N_x do
              equation_set2[equ2_count++] ←
                          replace_d(equation_set[i], j);
          end
  end
  for i ← 1 to length[equation_set2] do
      if equation_set2[i] includes f then
          for j ← 1 to N_x do
              equation_set2[i] ←
                          replace_f(equation_set2[i], j);
          end
      else if equation_set2[i] includes g then
          for j ← 1 to N_y do
              equation_set2[i] ←
                          replace_g(equation_set2[i], j);
          end
  end
  return equation_set2;
}
replace_v(Equation equ) {
  for i ← 0 to N_ξ do
      replace all ξ_i with x_i in equ;
      replace all κ_i with k_i in equ;
      replace all ι_i with y_i in equ;
      replace all τ_i with t_i in equ;
  end
  replace all ζ with z in equ;
  replace all δ with d in equ;
  return equ;
}
```

```
replace_sj(Equation equ, j) {
  lhs ← getLHS(equ);
  rhs ← getRHS(equ);
  replace all k_i with k_{i,j} in lhs;
  replace all y_i with y_{i,j} in lhs;
  replace all Φ with φ_j in rhs;
  replace all Γ with γ_j in rhs;
  equ ← makeEquation(lhs, rhs);
  return equ;
}
replace_d(Equation equ, j) {
  replace all x_i with x_{i,j} in equ;
  replace all k_i with k_{i,j} in equ;
  return equ;
}
replace_f(Equation equ, j) {
  lhs ← getLHS(equ);
  rhs ← getRHS(equ);
  replace φ_j(x_k, y_l, t_m, z) with f_j(x, y, t_m, z) in rhs;
  for i ← 1 to N_x do
      replace all x_i with x_{k,i} in rhs;
  end
  for i ← 1 to N_y do
      replace all y_i with y_{l,i} in rhs;
  end
  equ ← makeEquation(lhs, rhs);
  return equ;
}
replace_g(Equation equ, j) {
  lhs ← getLHS(equ);
  rhs ← getRHS(equ);
  replace γ_j(x_k, y_l, t_m, z) with g_j(x, y, t_m, z) in rhs;
  for i ← 1 to N_x do
      replace all x_i with x_{k,i} in rhs;
  end
  for i ← 1 to N_y do
      replace all y_i with y_{l,i} in rhs;
  end
  equ ← makeEquation(lhs, rhs);
  return equ;
}
```

Figure 7 Algorithm for generating the set of ODE numerical solution equations.

The traditional approach to creating simulation experiment codes also offers little flexibility once the software is created. Changing the numerical ODE solution method means making major revisions in the simulation code. The proposed method allows the flexibility of using different simulation models, boundary conditions, and numerical methods for a simulation experiment. Since the simulation codes are generated automatically, users can choose or change their desired ODE solver without making changes in the simulation codes themselves. Also, this can give clear information on what is calculated to generate the simulation results.

The different cell models and ODE numerical methods produced varying simulation code sizes. Table 4 lists the number of execution steps generated for the cell models using different ODE solutions. The table shows that the more complex the model becomes and the more equations it has, the larger the number of steps to compute the model equations in the code.

An issue in approximating ODE solutions is the accuracy of the numerical method used. A good approximation to the underlying differential equation needs to be achieved in order to arrive at accurate simulation results. We tested a number of commonly used ODE numerical methods to determine how the use of different solutions affect the accuracy of the calculations. The three methods used were Euler, Modified Euler and 4th-order Runge-Kutta. Each of these methods were used to generate an FHN model simulation code that runs in a single CPU. The generated code can run in different compilers and does not require third party software.

In order to test the accuracy of the ODE numerical methods, simulation codes were generated in Java using the BigDecimal class and numeric formatting. The Java BigDecimal can represent a large number of decimal places and help avoid rounding errors. It can offer higher precisions than the 16 decimal digits offered by floating point *double*. In the simulation codes, we used BigDecimal to represent the numbers in a 32-digit decimal point precision format for all the ODE solving methods. Different time steps were also used in testing the accuracy of these ODE methods, ranging from 10^{-1} ms to 10^{-5} ms. The simulation using a time step of 10^{-6} and Runge-Kutta as ODE solving scheme was used as the basis to compute for the root-mean-square error (RMSE) and evaluate the accuracy of the other calculations. The RMSE was computed for all the calculations using different ODE numerical methods and in varying time steps (Figure 8). Each level in the RMSE indicates a 1/10 less accuracy in the simulation results compared to the calculations using Runge-Kutta and 10^{-6} ms time step.

Table 4 The number of execution steps for the generated codes for different cell models and ODE numerical methods

Cell Model	Euler	Modified Euler	Runge-Kutta
FHN	11	16	26
LuoRudy1991	70	117	211
LuoRudy1994	123	211	387
Kyoto Model	335	560	1200

The first-order Euler method gave the largest error (log error $\approx 10^{-3}$) while the Modified Euler resulted to a smaller error. The fourth-order Runge-Kutta method resulted with the best accuracy (log error $\approx 10^{-10}$). However, the Runge-Kutta error is almost constant from the time step 10^{-3} ms, . This can be attributed to the rounding error of digits over the used 32-digit precision.

Simulation codes using different ODE numerical methods were also generated for models more complicated than the FHN or Hodgkin-Huxley model. For this, we used the model introduced by C. Luo and Y. Rudy in 1991 [8]. It is a simple cell model of cardiac action potential that uses Hodgkin-Huxley type equations to calculate ionic currents. The RMSE computations undertaken for the FHN model were also applied for the Luo-Rudy 1991 simulations. Codes were generated for the three ODE solutions with time steps ranging from 10^{-1} to 10^{-4} ms. Meanwhile, the simulation using the Euler method with a time step of 10^{-5} ms was used as the reference for RMSE calculations (Figure 9).

Excitation propagation simulations

One possible application of 1- or 2-dimensional excitation propagation simulation is cardiac arrhythmia research. Computational models have provided new insights into the underlying mechanisms of re-entry like the role of

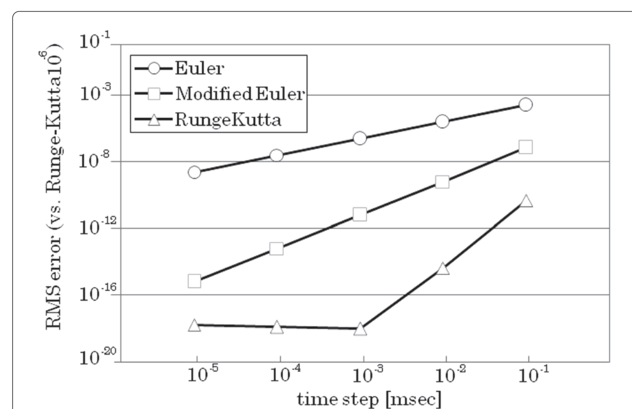


Figure 8 Relationship between the time step and calculation error for each ODE numerical method used in generating the FHN model. The results and root-mean-square errors (RMSE) are computed using Java BigDecimal. The result of the Runge-Kutta method with 10^{-6} ms time step is used as reference for the RMSE computations.

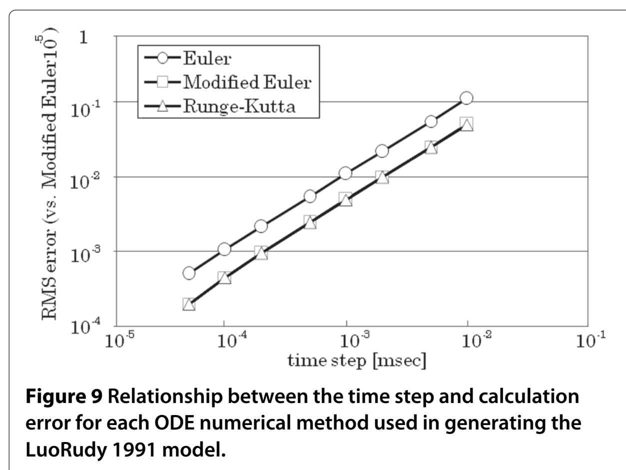


Figure 9 Relationship between the time step and calculation error for each ODE numerical method used in generating the LuoRudy 1991 model.

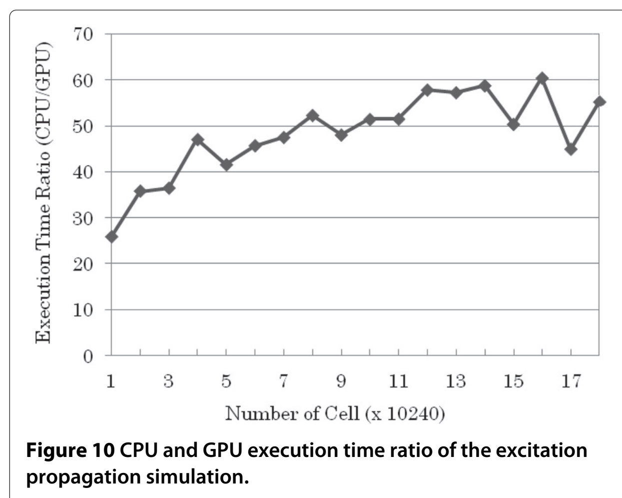


Figure 10 CPU and GPU execution time ratio of the excitation propagation simulation.

ionic currents and ion channel mutations [11]. In addition, designs of defibrillation treatments can be optimized using simulations of excitation propagation to achieve good clinical results for patients.

Simulation codes for the Luo-Rudy 1991 model were generated to simulate cell excitation propagation on a 2D homogeneous sheet. The two-dimensional cell action potential propagation simulations were run for both CPU and GPU to compare the speed of calculations. The CPU simulation was performed with an intel Core i7 880 processor with 8 GB of memory running Windows 7. The GPU has a single Nvidia Tesla C2050 processor with 448 CUDA cores, and 3 GB memory in a CentOS 5.5 system. The programs are written in C for the CPU simulations and Cuda C for the GPU.

In the experiments, the size of the homogeneous sheet was varied from 10240 (10240 $\times$ 1) to 1433600 (10240 $\times$ 14) cells with 10240-cell increments. The computation time was measured for each increment. Figure 10 shows the CPU and GPU computational time at different cell array sizes. Results showed that by using the GPU, the computational time can be accelerated 50 times as compared to using a standard CPU.

Discussion

Traditionally, biological function simulation programs have to be manually created from scratch. This is manageable with small models or simulation experiments but it becomes less practical once model complexity increases. The more complex the model becomes the larger the program becomes and the harder it is to create and maintain such programs. The Doty model [12] showed that the cost of software development is proportional to the square exponential of the number of computational steps. As seen from Table 4, the number of computational steps increases with the complexity of the model. By using this system, the creation of a simulation program from biological models becomes simpler. The system takes input from experts in biological functions through CellML models and mathematical experts through ODE numerical solutions and automatically generates the simulation code. The automatic code generation allows it to deal with large and complex models without the proportional increase in software development cost. This also keeps the software maintenance cost low since the programs are created with the same structure, regardless of the biological model under study.

The system's support for multiple programming languages makes it easier for users to test the model in different programming environments and keep the cost of switching from one programming language to a low. Based on the codes generated for the purpose of this paper, science and engineering students typically create a Java simulation code from the C version within four days.

The system requires the formatted RelML file to do the first stage of the algorithm and the RelML information can also be used to automatically determine the boundary conditions. The RelML information can be extracted automatically from the CellML model file but boundary conditions are not always available. We need a mathematical representation of boundary conditions which are compatible with declarative description. This can be addressed by the inclusion of PEPML experimental protocol [13] in the system. PEPML is an XML-based language that can describe the initial conditions and procedures of an experimental protocol. It describes boundary conditions in a mathematical form and is purely represented in a declarative manner, making it easy to combine with the current implementation of our system.

The attainable order of accuracy for approximating the Luo-Rudy 1991 model equations was determined by running simulations of the model using different ODE numerical methods. The results of the comparisons

between ODE solving schemes showed that the model can be accurately predicted by the first-order ODE solutions. The use of higher-order numerical solutions has a minimal effect on the values of the approximations (Figure 9). The Luo-Rudy model is a set of nonlinear differential equations that use higher order ODE solutions but it can be seen from the Taylor series expansion that equations from the second-order term has little or negligible effect. The structure of the system, where the ODE solver can be easily changed with the appropriate TecML file, allows one to confirm the accuracy of the model simulation such as the case with the Luo-Rudy 1991 model (see Additional file 1).

Although the system is fully implemented, further improvements are still to be added. The generated excitation propagation code does not consider the optimization of the model variables for parallel processing. Therefore, code parallelization is low and may have contributed to slower execution in the GPU environment. Also, the system can only handle equations with a single term in the left-hand side. Other equation forms will be addressed in future implementations.

Furthermore, simultaneous calculations using differential algebraic equations (DAE) are still not available. Simultaneous equations appear in some models and implicit numerical methods. The handling of implicit methods is available in the inputs but not in the current implementation of our system. A simultaneous equation library, other ODE numerical solutions library like CVODE, or a linear algebra library like LINPACK could be incorporated into our system. This can be a possibility with our plan to design a library interface description language to handle numerical solution inputs.

Future implementations also need to address the need for a declarative design of describing procedural methods. Complex ODE numerical methods like the ones described in the Kinetic Simulation Algorithm Ontology (KiSAO) [14] are difficult or impossible to encode in the current version of TecML. We are in the process of redesigning TecML and the system to incorporate methods that are suitable for the types of problems scientists are addressing with manually-created code, including adaptive methods. This can greatly enhance the usability of the proposed method.

In addition, the system may be integrated with SED-ML. SED-ML allows description of the simulation environment, which includes the name of the numerical method to be used in the simulation. A starting point for integration would be to provide TecML information of a numerical method for the KiSAO entry in SED-ML to allow simulation software to refer to this description. Another point would be to create support for SED-ML files in the code generation system in order to create more complex experiments that use more than one model or experiments with different simulation methods applied. This will allow users the flexibility of choosing not only the ODE solving methods but also the experiment protocol when generating their simulation codes.

Conclusion

In this paper, we proposed a method to automatically generate executable simulation codes using CellML physiological models and ODE numerical solution methods. The generated code describes the time-evolution of the set of differential equations enumerated by the CellML model. The code generation system is composed of a two-stage approach that allows flexible generation of complex sets of equations.

To evaluate the effectiveness of the proposed system, several combinations of physiological cell models and ODE solving schemes were generated. The output of the numerical approximations were in accordance with the published results of the cell models. Results for the Luo-Rudy 1991 simulations also indicated that it only has first-order accuracy. The comparison of execution time for 2D excitation propagation also showed that the use of GPU can accelerate the processing time by 50 times as compared to a CPU.

The code generation system allows executable simulation codes to be easily generated from CellML and TecML files. This can be very useful in the field of biological model simulation since it provides the tools to quantitatively evaluate the mathematical equations in these models.

Additional file

Additional file 1: Appendix.

Competing interests
The authors declare that they have no competing interests.

Author's contributions
FRP has been involved in the drafting of the manuscript, programming and gathering of experimental data. YY was responsible for programming of the code generator software and gathering data. NS was the main programmer of the code generator. MK was involved in the simulation experiments and data gathering as well as programming. TS, HK and YK acted as consultants for the algorithm and system design. AA has been the main architect of the algorithms and system design and has given the final approval of the version to be published. All authors read and approved the final manuscript.

Acknowledgements
The authors would like to thank the Japan Ministry of Education, Culture, Sports, Science and Technology (MEXT) for the Grant-in-Aid for Scientific Research (Kakenhi) funding.

Author details
[1]Graduate School of Life Sciences, Ritsumeikan University, Shiga, Japan. [2]Graduate School of Information Science and Engineering, Ritsumeikan University, Shiga, Japan. [3]ASTEM Research Institute of Kyoto, Kyoto, Japan.

Received: 28 March 2012 Accepted: 31 July 2012
Published: 19 October 2012

References

1. Cuellar AA, Lloyd CM, Nielsen PF, Bullivant DP, Nickerson DP, Hunter PJ: **An overview of CellML 1.1, a biological model description language.** *Simulation* 2003, **79:**740–747.
2. Hucka M, Kitano H: **The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models.** *Bioinformatics* 2003, **19:**524–531.
3. Yoshiyuki A, Yasuyuki S, Yoshiyuki K: **Specifications of insilicoML 1.0: a multilevel biophysical model description language.** *J Biol Sci* 2008, **58:**447–458.
4. Waltemath D, Adams R, Bergmann FT, Hucka M, Kolpakov F, Miller A, Moraru II, Nickerson DP, Sahle S, Snoep JL, Le Novere, N: **Reproducible computational biology experiments with SED-ML - the simulation experiment description markup language.** *BMC Syst Biol* 2011, **5:**198.
5. Fitzhugh R: **Impulses and physiological states in theoretical models of nerve membrane.** *Biophys J* 1961, **1**(6):445–466.
6. Garny A: **Cellular Open Source (COR): a public cellml based environment for modeling biological function.** *Int J Bifurcation and Chaos* 2003, **13**(12):3579–3590.
7. **The CellML Compiler and Code Generator.** [http://sourceforge.net/projects/cellmlcompiler].
8. Luo CH, Rudy Y: **A model of the ventricular cardiac action potential, depolarization, repolarization, and their interaction.** *Circ Res* 1991, **68:**1501–1526.
9. Matsuoka S, Sarai N, Kuratomi S, Ono K, Noma A: **Role of individual ionic current systems in ventricular cells hypothesized by a model study.** *Jpn J Physiol* 2003, **53:**105–123.
10. Luo CH, Rudy Y: **A dynamic model of the cardiac ventricular action potential. I. Simulations of ionic currents and concentration changes.** *Circ Res* 1994, **74:**1071–1097.
11. Ten Tusscher KH, Panfilov AV: **Cell model for efficient simulation of wave propagation in human ventricular tissue under normal and pathological conditions.** *Phys Med Biol* 2006, **51:**6141–6156.
12. Herd JR, Postak JN, Russell WE, Steward KR: *Software cost estimation study: Study results, Final Technical Report, RADC-TR77-220, vol. I. Rockville*. Doty Associates: Inc.; 1977.
13. Shimayoshi T, Amano A, Matsuda T: **A generic representation format of physiological experimental protocols for computer simulation using ontology.** In *Engineering in Medicine and Biology Society, 2007. EMBS 200. 29th Annual International Conference of the IEEE*; 2007:382–385.
14. **Kinetic Simulation Algorithm Ontology (KiSAO).** [www.biomodels.net/kisao].

doi:10.1186/1751-0473-7-11
Cite this article as: Punzalan *et al.*: A CellML simulation compiler and code generator using ODE solving schemes. *Source Code for Biology and Medicine* 2012 **7**:11.

Submit your next manuscript to BioMed Central and take full advantage of:

- Convenient online submission
- Thorough peer review
- No space constraints or color figure charges
- Immediate publication on acceptance
- Inclusion in PubMed, CAS, Scopus and Google Scholar
- Research which is freely available for redistribution

Submit your manuscript at
www.biomedcentral.com/submit