A Parallel CellML Simulation Program Generator with a Nonlinear Simultaneous Equation Solver

Yoshiharu Yamashita*, Yuichiro Hayashi†, Naoki Soejima*, Masanari Kawabata‡, Punzalan Florencio Rusty‡, Takao Shimayoshi§, Hiroaki Kuwabara†, Yoshitoshi Kunieda† and Akira Amano‡

*Graduate School of Science and Engineering, Ritsumeikan University, Shiga, Japan

†College of Information Science and Engineering, Ritsumeikan University, Shiga, Japan

‡College of Life Science, Ritsumeikan University, Shiga, Japan

§ASTEM Research Institute of Kyoto, Kyoto, Japan

I. Introduction

In recent biological research, there has been increased focus on the analysis of complex biological functions of the human body using simulations [1]. Understanding the complex mechanisms in the human body is important for medical and drug development research. To analyze these complex mechanisms, a multiscale approach on biological function simulations is needed. In multiscale simulations, the tissue or organ is represented as a collection of equations representing a cell model. This poses a challenge in creating biological function simulation since the simulations consist of hundreds of complicated equations. To address this difficulty, a software that automatically generates simulation program from various cell models is necessary. This can be made possible with the help of description languages to handle machine-readable models [2][3]. One of the most famous model description languages is CellML [2]. CellML can be used for describing cellular function models and most of the models are provided in the CellML repository [4]. A few software applications also generate simulation program from model description languages [5][6].

In doing complex biological function simulations, it is necessary to consider coupling calculations between model equations and fields. Furthermore, parallel programs also need to be generated to deal with the complexity and computational requirements of multiscale simulations. All of these has not been available in a single software as of the writing of this paper.

We previously proposed a simulation program generator that handles coupling simulations and generates parallel programs [7]. The current implementation of the code generator supports explicit ordinary differential equations (ODE) and explicit ODE numerical methods. However, some models consist of implicit ODE and need implicit numerical methods such as implicit Runge-Kutta. Methods which can solve nonlinear simultaneous equations (NSE) are necessary to compute these models. Therefore, the handling of NSEs and implicit ODEs needs to be included in the simulation program generator. In addition, an automatic parallelization in CUDA [8] for the simulation programs of the implicit ODE and ODE methods is needed.

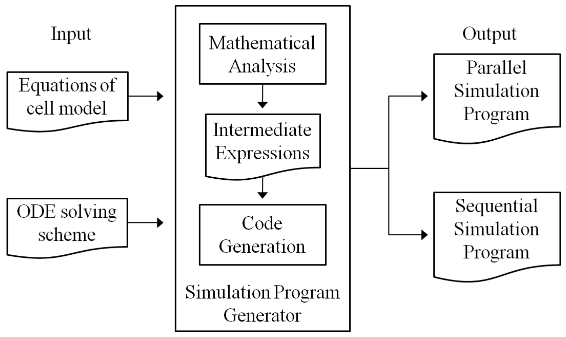

Fig. 1. Simulation program generator.

In this paper, we propose methods to automatically generate simulation programs that include an NSE solver. The simulation programs are generated from CellML models and are automatically parallelized in graphics processing unit (GPU) using CUDA programming. The proposed methods can be used to generate sequential and parallel simulation programs of implicit ODE and ODE methods from the description languages.

II. Simulation Program Generator

Simulation involves the calculation of model equations with help of numerical methods to solve differential equations. Example of numerical methods are Euler method and Runge-Kutta method. The basic elements of the simulation program are these equations and numerical methods. The simulation program generator generates the simulation program from equations of cell model and ODE solving schemes (Fig. 1). The simulation program generator has two major stages. The First stage is the mathematical analysis to construct intermediate expressions. The intermediate expressions are constructed from the equations of cell model and the ODE solving schemes. The Second stage is the code generation to generate the simulation program from these intermediate expressions.

Fig. 2 shows the inputs and outputs of the simulation program generator. The inputs are CellML [2], TecML (Time Evolution Calculation Markup Language) [7] and RelML (Relation Markup Language) [7]. CellML is widely used as a description language for cellular function models. CellML consists of metadata and mathematical equations. CellML

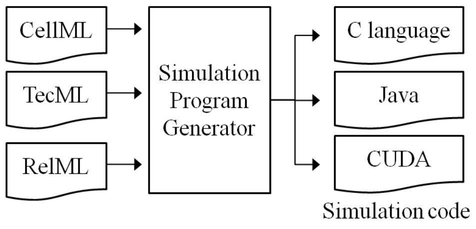

Fig. 2. Input and output of the simulation program generator.

provides equations and cell structure as metadata in a cell model. The equations in CellML are described in MathML. TecML is a description language to mathematically describe ODE solving scheme such as the Euler method, Modified Euler method and Runge-Kutta method. TecML offers flexibility selecting ODE solving scheme depending on speed and stability requirements. In TecML, five variable types and two function types are defined. The variable types are diffvar, derivativevar, arithvar, constvar, timevar and deltatimevar. Diffvar is a differential variable, derivativevar is derivative variable, arithvar is arithmetic variable (this is used for variables which can be mathematically substituted), constvar is constant, timevar is time stamp, and deltatimevar is the time step. The function types are diffequ and nondiffequ. Diffequ is used for differential equations while nondiffequ is used for nondifferential equations. RelML is a description language to specify correspondence of variable types in CelML to types defined in TecML. A CellML variable needs a TecML variable type to use an ODE solving scheme described in TecML. These variable types are not defined in CellML so RelML assigns it to the the CellML variables.

The output of the generator is a simulation code. A number of code generators (C language, Java, CUDA C [8]) are prepared to support parallel and sequential execution environments. A user can change an execution environment by selecting a specific code generator.

The intermediate expressions are constructed in the mathematical analysis from model equations described by CellML, ODE solving scheme described by TecML and information about types of CellML variables described by RelML. Fig. 3 shows expressions construction from TecML describing the backward Euler method (an implicit ODE method), CellML file of FHN (FitzHugh-Nagumo) model [9] and RelML file of the FHN model. The intermediate expressions in Fig. 3 had to be simultaneously calculated, because the intermediate expressions equal the following formulas:

$$x_{t+\delta} = x_t + (x_{t+\delta} - x_{t+\delta}^3/3.0 - y_{t+\delta} + a) * \delta \quad (1)$$

$$y_{t+\delta} = y_t + (b * (x_{t+\delta} + c - d * y_{t+\delta})) * \delta \quad (2)$$

$x_{t+\delta}$ and $y_{t+\delta}$ are computed at same time so an NSE solver is necessary to simultaneously calculate them.

III. Solving Nonliner Simultaneous Equation

A. Finding strongly connected components and sorting equations

To generate a simulation program with an NSE solver, finding strongly connected components in the equations and sorting equations are necessary [10]. We use Tarjan's algorithm [11] to find the strongly connected components and sort the equations in Fig. 4 (a) (b). Intermediate expressions are constructed from sorted equations adapted for the specific ODE solving scheme in Fig. 4 (c).

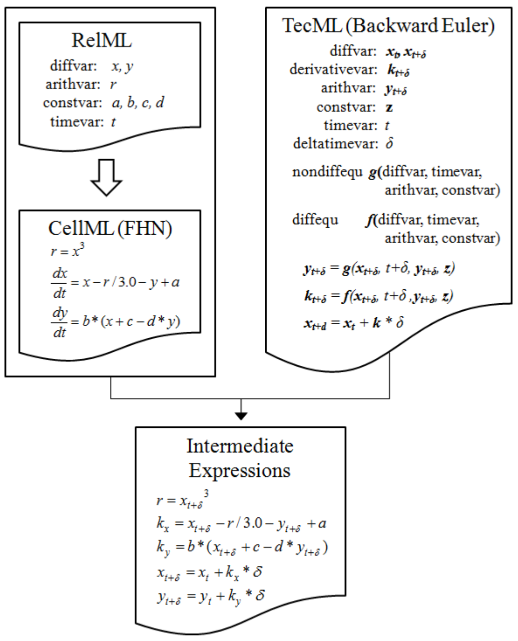

Fig. 3. Constructing the intermediate expressions from CellML, TecML and RelML.

$$\begin{array}{l} f_1(x_1,x_3,x_4)=0 \\ f_2(x_2)=0 \\ f_3(x_2,x_3,x_5)=0 \\ f_4(x_1,x_2)=0 \\ f_5(x_3,x_5)=0 \end{array} \qquad \begin{array}{l} f_2(x_2)=0 \\ f_4(x_1,x_2)=0 \\ f_3(x_2,x_3,x_5)=0 \\ f_5(x_3,x_5)=0 \\ f_1(x_1,x_3,x_4)=0 \end{array}$$

$$S_1 = \begin{array}{ccccc|c} x_1 & x_2 & x_3 & x_4 & x_5 & \\ 1 & 0 & 1 & 1 & 0 & f_1 \\ 0 & 1 & 0 & 0 & 0 & f_2 \\ 0 & 1 & 1 & 0 & 1 & f_3 \\ 1 & 1 & 0 & 0 & 0 & f_4 \\ 0 & 0 & 1 & 0 & 1 & f_5 \end{array} \qquad S_2 = \begin{array}{ccccc|c} x_2 & x_1 & x_3 & x_5 & x_4 & \\ 1 & 0 & 0 & 0 & 0 & f_2 \\ 1 & 1 & 0 & 0 & 0 & f_4 \\ 1 & 0 & \boxed{1} & \boxed{1} & 0 & f_3 \\ 0 & 0 & \boxed{1} & \boxed{1} & 0 & f_5 \\ 0 & 1 & 1 & 0 & 1 & f_1 \end{array}$$

(a) (b)

$$f_2\begin{cases} exp0 \\ exp1 \end{cases} \quad f_4\begin{cases} exp2 \\ exp3 \end{cases} \quad f_3, f_5\begin{cases} \boxed{\begin{array}{l} exp4 \\ exp5 \end{array}} \text{ strongly connected component} \\ exp6 \\ exp7 \end{cases} \quad f_1\begin{cases} exp8 \\ exp9 \end{cases}$$

(c)

Fig. 4. (a) Equation system with associated adjacency matrix in a model. (b) Equation system after sorting and finding a strongly connected component using Tarjan's algorithm. (c) Intermediate Expressions after adapting ODE solving scheme to the equation system of (b).

B. Embedding an NSE solver in the simulation program

The code generator generates a simulation program with NSE solvers from the intermediate expressions. The connected components in the intermediate expressions are then calculated in the NSE solvers. Fig. 5 shows a code generation process where the simulation program with the NSE solver is generated from the intermediate expressions and connected components. The program simultaneously calculates *exp2* and *exp3* in an NSE solver. Even if *exp2* and *exp3* are linear equations, the program calculates them using the solver. This allows the current code generator to support linear simultaneous equations.

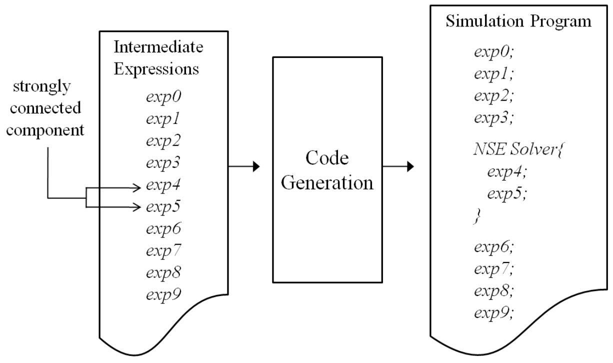

Fig. 5. Simulation program including NSE solver.

IV. Parallelization in CUDA

The computational complexity of a general biological function simulation program is enormous so it needs program parallelization. Using parallel programs, we accomplished an acceleration ratio of 66 between the GPU and the single CPU computation time in a multi-cell simulation (cells numbers from 10240 to 208000) [7]. In this section, we propose an automatic parallelization in CUDA for one cell simulation program including NSE solver from the intermediate expressions and its dependency information. CUDA is NVIDIA's parallel computing architecture, which can attain high parallelism in GPU [8]. In parallelizing only the NSE solver in CUDA, speed up is difficult due to the loop dependency in the solver. In this case, the effect of data transfer latency is larger than the effect of parallelization. We can reduce ratio of the data transfer by the parallelization of the whole program. The generator not only parallelizes the NSE solver but the whole computational process in one time step loop. Fig. 6 shows the parallelized calculation and the CUDA functions that calculate them. CUDA kernel2 uses at least two warps to concurrently calculates the equations. Warp is a unit of CUDA calculation, one warp consists of 32 threads. One warp calculates *exp2* and *exp3*, while another warp calculates *exp4*, *exp5*, *exp6* and *exp7*. The NSE solver of *exp4* and *exp5* is also parallelized.

V. Discussion

The proposed method to generate a simulation program solving NSE from the description languages is considered to be beneficial in terms of allowing users to generate simulation codes easily. The NSE solver is necessary to incorporate explicit and implicit ODE methods.

We also proposed another method which is the automatic parallelization in CUDA of the simulation programs with NSE solvers. The high parallelization reduces computational complexity of a simulation program that contains implicit ODE or uses implicit ODE methods. We consider that the optimization of this generator offers more flexibility than general optimization using static analysis. Since operations in this generator are mathematical, operations such as replacement algorithm can be used for optimization.

Future studies involve addition of experimental protocols, coupling calculation and fields for an elaborate biological function simulation.

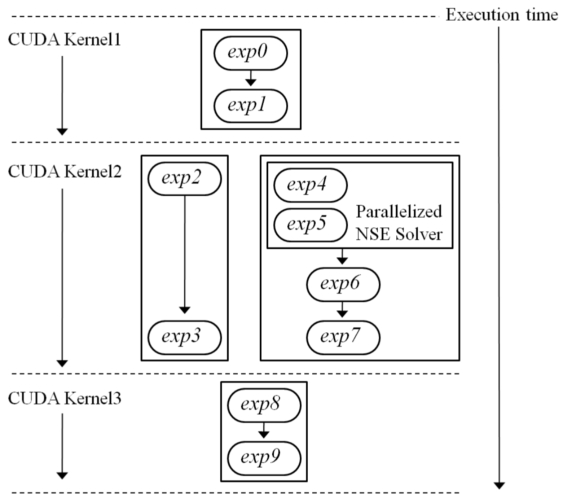

Fig. 6. Parallel simulation program of Fig. 5.

VI. Conclusion

We proposed two methods to incorporate implicit ODE and ODE methods in biological function simulations. The first method is to directly generate a simulation program of implicit ODE and ODE methods from description languages. The second method is the automatic parallelization of the simulation program that contains an NSE solver. These methods are considered to be beneficial for the generation of multiscale simulation programs. Henceforth, we will implement these methods in the simulation program generator.

References

[1] P. J. Hunter, P. Robbins, and D. Noble, "The iups human physiome project," *Pflügers Archiv European Journal of Physiology*, vol. 445, no. 1, pp. 1–9, Oct. 2002.

[2] A. A. Cuellar, C. M. Lloyd, P. M. F. Nielsen *et al.*, "An overview of cellml 1.1, a biological model description language." *Simulation*, vol. 79, no. 12, pp. 740–747, 2003.

[3] Y. Asai, Y. Suzuki, Y. Kido *et al.*, "Specifications of insilicoml 1.0: A multi-level biophysical model description language." *J Physiol Sci*, 2008.

[4] (2011) Cellml model repository. [Online]. Available: http://models.cellml.org/cellml

[5] A. Garny, D. Noble, P. J. Hunter *et al.*, "Cellular open resource (cor): current status and future directions." *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, vol. 367, no. 1895, pp. 1885–1905, May 2009.

[6] Y. Suzuki, Y. Asai, H. Oka *et al.*, "A platform for in silico modeling of physiological systems iii." *Conf Proc IEEE Eng Med Biol Soc*, vol. 1, 2009.

[7] A. Amano, N. Soejima, T. Shimayoshi *et al.*, "A general cellml simulation code generator using ode solving scheme description," in *Engineering in Medicine and Biology Society,EMBC, 2011 Annual International Conference of the IEEE*, 30 2011-sept. 3 2011, pp. 940 –944.

[8] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[9] R. FitzHugh, "Impulses and physiological states in theoretical models of nerve membrane," *Biophysical journal*, vol. 1, 1961.

[10] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 1st ed. Wiley-IEEE Press, Jan. 2004.

[11] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.