Myokit: A Framework for Computational Cellular Electrophysiology

Michael Clerx[1,2], Paul GA Volders[2], Pieter Collins[1]

[1] Department of Knowledge Engineering, Maastricht University, The Netherlands
[2] Cardiovascular Research Institute Maastricht, Maastricht University, The Netherlands

Abstract

This paper describes Myokit, an open source framework for simulation and analysis of models of cellular cardiac electrophysiology. It is designed to be lightweight, with a focus on model development and model use. The framework consists of a model definition language and a set of tools to manipulate models, run simulations and post-process the results. Single-cell and parallelized multi-cell simulation engines are provided, as well as import and export for various formats and tools for sensitivity analysis. This can provide a valuable addition to the body of software tools available for computational cellular electrophysiology.

1. Introduction

The electrical behavior of heart muscle cells (cardiomyocytes) is directly responsible for the controlled contraction of the heart. Changes to these cells' electrical properties due to mutations, drug-induced effects or other causes, lie at the basis of severe cardiac health issues such as Brugada syndrome, long QT syndrome and sudden cardiac death.

Cell models describe the active and passive ion fluxes through the cell membrane and keep track of the resulting changes in membrane potential and ionic concentrations. Modern models may include ion fluxes between different areas in a cell, calcium buffering or modulating influences such as adrenergic stimulation. All of these phenomena can be captured in a model formulated as a system of ordinary differential equations (ODEs).

This paper introduces Myokit, a software toolkit designed to simplify the use and development of such models. The software is open source[1] and runs on Linux, Apple and Windows systems. The source files can be downloaded from [1] along with extensive documentation, a number of example files and an installation guide for each supported platform. For windows users, an installation script is provided.

[1]The code is licensed under a GNU General Public License.

2. The Modeling Language

Cellular electrophysiology models are commonly implemented using general purpose or numerical languages such C, C++ or Matlab. An alternative approach is to separate the implementation from the equations and use a dedicated model definition language [2]. This allows re-use of both models and analysis tools, allows models and tools to be freely exchanged between researchers and can help ensure model consistency. A good example of a model description language is the XML based format CellML [3]. As of writing, the CellML model repository [4] lists 264 electrophysiological models and various tools can be downloaded from the website that can be used for cardiac cellular electrophysiology. The choice for XML ensures interoperability with a wide range of existing parsers and makes CellML highly suitable as an archiving and exchange format.

By contrast, Myokit is aimed at model use and *development*. It uses a a plain text, human-writable format that produces compact, easy to read files. The format is designed to be strict, unambiguous and free of implementation details, but at the same time easy to read, write and maintain. Myokit model files can be edited with any text editor and the provided toolkit can use the files directly to run simulations and perform other modes of analysis. It is the authors' hope that this will provide a valuable complement to the existing body of software tools used in cardiac cellular electrophysiology.

2.1. File Structure

Myokit models are stored in plain text files using the extension `mmt`. Each file contains a model definition and a set of realistic initial conditions and may also contain a pacing protocol and an example experiment. To this end, the file format is split into a required `model` segment and two optional segments `protocol` and `script`. The model segment is discussed in Section 2.2, the other two segments are described briefly below. The full specification of the `mmt` format is given in the online documentation [1].

Myocyte models are typically *paced*, i.e. driven with a periodic stimulus, the required strength and duration of which can vary from model to model. Simulation engines and exports with support for pacing can use the `protocol` section of an `mmt` file to obtain a suitable pacing protocol. This segment can also be used to implement non-periodic inputs, such as a series of voltages for voltage-clamp experiments.

The optional `script` section can be used to store a Python script that uses the Myokit API to set up a reference experiment. It uses the same syntax as an external script but has access to the magic methods `get_model()` and `get_protocol()` to read the other parts of the file.

2.2. Model definition syntax

Models are divided into *components* which contain the model's variables. Each variable is defined through a single equation. For state variables this is a differential equation written as `dot(x) = ...`, for all other variables this is simply an assignment `x = ...`. The variables can be specified in any order.

Within a component, variables can refer to each other directly, but to access a variable from another component its fully qualified name `component_name.variable_name` must be used. Variables can define child variables whose value is only visible to themselves and the child variables' siblings. This structure allows complex models to be written in a clear and structured way.

The syntax in a model definition is deliberately simple: a new component is started by writing its name between square brackets and then each following non-indented line defines a new variable. Meta-data, units and child variables are written indented, directly below their parent variable[2]. The syntax for expressions is similar to C-based languages. Due to the declarative nature of the language, no flow control is possible, but conditional expressions are provided through the function `if(<condition>,<then>,<else>)`. Any line starting with a hash mark (`#`) is taken to be a comment.

Units can be specified for any numerical constants appearing in expressions. Variables can also specify a canonical unit and unit arithmetic is provided so that these can be checked against the units derived from their expressions. Meta-data strings can be added to models, components and variables using the syntax `key: value`, where `key` is a single word property name and `value` is one or more lines defining the value.

Interfacing with external components happens through the *binding* and *labeling* of variables. Labeled variables can be used by simulation engines or other routines to identify variables with a well defined role. For example, in cell models suitable for multi-cellular simulation, the label `membrane_potential` is used to indicate the variable that represents the cell's membrane potential. This is then read by the simulation engine and used to calculate a diffusion current. A variable binding is used to indicate that a simulation engine may replace a variable's value with an external one, identified by the binding's name. For example, the statement `t = 0 bind time` defines a variable named `t`. Its default value is zero, but the bind statement declares any simulation engine can update its value to reflect the current simulation time.

A partial example of a model definition for a historically important model [5] is given below to illustrate the syntax. Further examples (historical and contemporary) are provided in the examples section on the Myokit website [1].

```
[[model]]
name: BR1977
# Initial values:
membrane.V = -84.622
ina.m      = 0.01
ina.h      = 0.99
ina.j      = 0.98

[external]
t = 0 bind time
p = 0 bind pace

[membrane]
C = 1 [uF/cm^2]
dot(V) = -(1/C) * (I_ion + I_diff + I_stim)
    in [mV]
    label membrane_potential
    desc: Membrane potential (in mV)
I_ion = ik1.IK1 + ix1.Ix1 + ina.INa + isi.Isi
I_diff = 0 bind diffusion_current
I_stim = external.p * amplitude
    amplitude = -25 [uA/cm^2]

[ina]
use membrane.V as V
gNaBar = 4 [mS/cm^2]
gNaC = 0.003 [mS/cm^2]
ENa = 50 [mV]
INa = (gNaBar * m^3 * h * j + gNaC) * (V - ENa)
dot(m) =  alpha * (1 - m) - beta * m
    alpha = (V + 47) / (1 - exp(-0.1 * (V + 47)))
    beta  = 40 * exp(-0.056 * (V + 72))
dot(h) =  alpha * (1 - h) - beta * h
    alpha = 0.126 * exp(-0.25 * (V + 77))
    beta  = 1.7 / (1 + exp(-0.082 * (V + 22.5)))
dot(j) =  alpha * (1 - j) - beta * j
    alpha = 0.055 * exp(-0.25*(V+78)) / (1+exp(-0.2*(V+78)))
    beta  = 0.3 / (1 + exp(-0.1 * (V + 32)))
```

As this example shows, the language is considerably more compact than its XML counterparts and requires very little code other than the bare model equations.

3. The Myokit Toolbox

This section describes the main tools offered through the Myokit API. Myokit is written predominantly in Python, but uses on-the-fly compilation to run performance critical tasks entirely in C. In addition to the performance benefits, this allows the framework to connect to specialized C libraries such as the ODE solver CVODE [6] and the parallelization library OpenCL [7].

Myokit's core defines a number of classes implementing models, components, variables and expressions. Models can be built up programmatically, using the same API

[2]A number of properties: units, labels, bindings and the meta-data property "desc" can also be written in a short-hand style directly after the variable equation

exposed to the parser. Once a model is completed, it is checked for cyclical dependencies and a solvable order for the equations is determined. Great care is taken at each step of the process to provide helpful error messages if needed.

The symbolic form of the model is useful in several types of analysis. For example, constants like external ion concentrations can be changed before simulations, but the same mechanism allows non-constant expressions to be changed, for example to partially block certain channels. Methods are provided to check Myokit implementations against reference ones by evaluating the state vector derivatives at different points in the state space. Dependencies between components or variables can be evaluated and graphed and a quick plot of state variable dependencies can be made which shows the shape of the Jacobian.

3.1. Simulations

Single-cell simulation is currently implemented by a back-end using the ODE solver CVODE [6], which uses an implicit adaptive multi-step method ideally suited for stiff ODEs. The simulation is implemented in a C-based Python module. Results are logged using standard Python data structures and made available to the user through a `SimulationLog` object, which also provides some post-processing and I/O options. Although the online examples (and parts of the graphical user interface) use MatplotLib [8] for visualization, Myokit is not tied to any particular visualization library.

Multi-cell simulation for one-dimensional fibers or two-dimensional tissues is implemented using a forward Euler solver implemented either in plain C or using OpenCL [7] for parallelization on GPU or multi-core CPU. A graphical user interface for visualizing time-variant 2D rectangular data grids is included. The model's sensitivity to a single parameter can be investigated using the `RangeTester` simulation. This is an uncoupled multi-cell simulation where each cell is simulated with a different value for the specified parameter. Finally, a tool is provided to calculate the Jacobian matrix during simulations, allowing sensitivity and stability to be investigated.

3.2. Import and Export

Models can be imported from CellML, SBML[3] and ChannelML, data and protocols can be read from Axon Binary Files. Model definitions can be exported to Matlab, C (and C++), Python, CUDA, OpenCL and CellML. For presentation purposes, model equations can be exported to Latex, HTML or MathML. A complete overview of supported formats can be found in the online documentation.

[3]The Systems Biology Markup Language

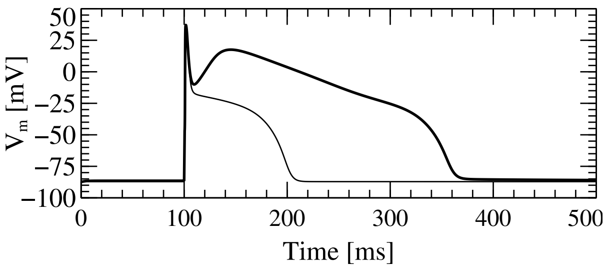

Figure 1. A normal action potential (thick line) and one under conditions of severely reduced external Calcium concentration (thin line).

3.3. User Interface

Myokit includes a graphical user interface (GUI) to edit `mmt` files, run experiments and perform imports and exports. The GUI also provides some debugging features, for example graphing variables or showing how they are evaluated. These functions help reduce the time needed to (re-)implement models from existing programming code or published equations. The GUI is not loaded as part of the framework core, which allows users to choose alternative graphical or windowing back-ends.

3.4. Examples

The code samples below show how to perform some basic tasks using Myokit. First, the Myokit library is loaded and an `mmt` file is read. Its model, protocol and script are stored in `m`, `p` and `x` respectively. The model used in the examples is the 2009 model by Decker et al. [9] and can be downloaded from the examples section on the website. Next, a single cell simulation is created and a 500ms simulation is run, storing the logged data in the `SimulationLog` object `d`. The results can be visualized directly or, as in this example, stored to disk for later processing. Figure 1 shows the simulated action potential.

```
import myokit
m, p, x = myokit.load('decker2009.mmt')
s = myokit.Simulation(m, p)
d = s.run(500)
d.save_csv('example1.csv')
```

Next, we lower the external Calcium concentration by selecting the variable `extra.Cao` and replacing its right-hand-side expression (RHS) with a lower value. To give the model time to adjust to this new setting, we pre-pace for 30 beats, and then run one more logged simulation.

```
m.get('extra.Cao').set_rhs('0.1')
s = myokit.Simulation(m, p)
s.pre(30000)
d = s.run(500)
```

To get a more complete picture of the influence of external Calcium on the action potential, we can use the

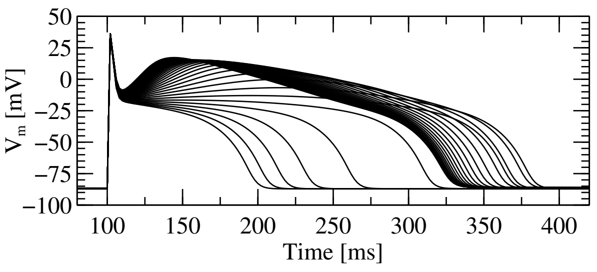

Figure 2. Thirty-five action potentials generated with external Calcium concentrations between 1.8 mmol/L (normal) and 0.1 mmol/L with steps of 0.05 mmol/L.

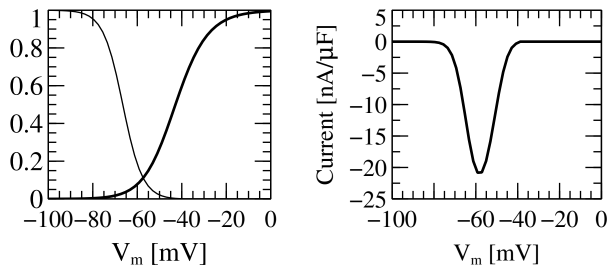

Figure 3. *Left:* The steady-state activation (thick line) and inactivation (thin line) curves of the model's Sodium current. *Right:* The window current calculated from the steady state activation and inactivation curves.

`RangeTester` simulation. In this example, we tested 35 linearly spaced values between 0.1 mmol/L and the original value of 1.8 mmol/L. Results are shown in Figure 2.

```
n = 35
v = np.linspace(0.1, 1.8, n)
r = myokit.RangeTester(m, p, var='extra.Cao', ncells=n)
r.pre(30000, values=v)
d = r.run(500, values=v)
```

Finally, we present an example without simulations: Two parameters of Sodium channel activation are retrieved from the symbolic model representation and converted into Python functions. These are then used to calculate the channel's activation curve. We can use the same process to find the inactivation curve. Finally, we retrieve the maximum conductance `Gbar` along with the equilibrium potential `ENa` and calculate the window current for this channel. The results are shown in Figure 3.

```
v = np.linspace(-100, 20)

a = m.get('ina.m.alpha').pyfunc('membrane.V')
b = m.get('ina.m.beta').pyfunc('membrane.V')
a_inf = a(v) / (a(v) + b(v))

a = m.get('ina.h.alpha').pyfunc('membrane.V')
b = m.get('ina.h.beta').pyfunc('membrane.V')
i_inf = a(v) / (a(v) + b(v))

g = m.get('ina.Gbar').rhs().eval()
E = m.get('nernst.ENa').rhs().eval()
w = g * (a_inf ** 3) * (i_inf ** 2) * (v - E)
```

4. Conclusion

In this paper, an open-source software framework for the creation and analysis of cellular electrophysiological models was presented. The used model definition language is human readable and writable and can serve as a complement to existing archiving or exchange formats. A number of simulation types are provided, with an easy to use Python front-end and a fast C-based back-end. The combination of a clear model syntax, a high level API and the analysis tools provided can serve to reduce model development and debugging times, freeing valuable resources for physiological investigation.

Acknowledgements

The authors would like to thank Dr. Enno de Lange for our valuable discussions.

References

[1] Myokit website. http://myokit.org/.

[2] Hedley WJ, Nelson MR, Bellivant D, Nielsen PF. A short introduction to CellML. Philosophical Transactions of the Royal Society of London Series A Mathematical Physical and Engineering Sciences 2001;359(1783):1073–1089.

[3] Cuellar AA, Lloyd CM, Nielsen PF, Bullivant DP, Nickerson DP, Hunter PJ. An overview of CellML 1.1, a biological model description language. SIMULATION December 2003;79(12):740–747.

[4] CellML model repository. http://models.cellml.org/.

[5] Beeler GW, Reuter H. Reconstruction of the action potential of ventricular myocardial fibres. J Physiol June 1977; 268(1):177–210.

[6] Hindmarsh AC, Brown PN, Grant KE, Lee SL, Serban R, Shumaker DE, Woodward CS. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. ACM Trans Math Softw September 2005;31(3):363–396.

[7] Stone JE, Gohara D, Shi G. OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science engineering 2010;12(3):66.

[8] Hunter JD. Matplotlib: A 2D graphics environment. Computing In Science Engineering 2007;9(3):90–95.

[9] Decker KF, Heijman J, Silva JR, Hund TJ, Rudy Y. Properties and ionic mechanisms of action potential adaptation, restitution, and accommodation in canine epicardium. Am J Physiol Heart Circ Physiol April 2009;296(4):H1017–H1026.

Address for correspondence:

Michael Clerx
Maastricht University
P.O. Box 616, 6200 MD Maastricht, The Netherlands
michael.clerx@maastrichtuniversity.nl