How to Become a Supermodel

Model creation for CESE is not that hard after all.

by Sergey Missan

This article discusses model creation with CESE API. Both extending the existing model and creating a new one are described. It provides information on interaction of models with CESE environment and model packaging. Data export from the model to the simulation environment is explained.

Table of contents

The object-oriented approach of CESE platform and rich API greatly simplifies models code, makes model development faster and easier.

1. Prerequisites

Even if you never dealt with Java before — don't worry. The subset of Java features you need to learn in order to create models is very limited, and example models packaged with CESE will give you a pretty good idea of the code organization. To program in Java you'll need:

- The Java development kit (J2SDK) for your platform.
- The Java IDE: Eclipse and Netbeans are both free and powerful.
- The CESE *source* distribution: visit our download page for the available options.

If you want to gain some additional Java skills and knowledge, The Java Tutorial is a proven starting point.

> Java is a case-sensitive language. Be careful when altering the source code in the examples below.

2. Extending the existing model — the simple way

Let's start with the relatively easy task of extending the existing model and updating the ionic current implementation.

As an example model we are going to use the Luo-Rudy phase I action potential model, included in CESE distribution. This model describes 6 currents using the conventional Hodgkin-Huxley (HH) formulations.

To begin, copy `$CESE_HOME/models/com/javable/ cese/models/LuoRudyModel_I_1991.java` to `$CESE_HOME/models/com/javable/ cese/models/MyModel.java` to preserve the original model.

Open `$CESE_HOME/models/com/javable/ cese/models/MyModel.java` in the Java IDE or in your favorite text editor. We are going to adjust formulation for the Fast sodium current (INa), changing its formula from `m^3*h` to `m*h`.

> Yes, this modification makes no sense at all. We are doing it only to illustrate the point.

Find the method that does INa computations: `protected void currentINa()`. At the end of this method find the line

```
ina = gna * m * m * m * h * j * (v - ena);
```
and replace it with

```
ina = gna * m * h * j * (v - ena);
```
We also have to make some changes in the correspondent BeanInfo class. This file provides information for CESE about parameters that our model contains.

First, copy `$CESE_HOME/models/com/javable/ cese/models/LuoRudyModel_I_1991BeanInfo.java` to `$CESE_HOME/models/com/javable/ cese/models/MyModelBeanInfo.java`

> The BeanInfo class name should match that of the model name with "BeanInfo" suffix added.

Open the `MyModelBeanInfo.java` file and change all the occurrences of `LuoRudyModel_I_1991 ...` to `MyModel ...`, for example:

```
 /** Array of property descriptors. */
    public MyModelBeanInfo() {
        icon = loadImage ("model_cardiac.png");
    }
```
As a final step we should compile and package our model into the jar archive that will be used for model distribution. We need to package `MyModel.class` and `MyModelBeanInfo.class` into the archive.

CESE uses a convinient Ant-based build system that makes it simple to build models. Modify `$CESE_HOME/bin/models/build.xml` script to include MyModel files. Run

```
$ANT_HOME/bin/ant models
```

in the `$CESE_HOME` directory to compile and package models. Your model will be automatically installed next time CESE runs. Use Model/Install Model command to install your new model without restarting CESE.

3. Extending the existing model — the OOP way

Even though we successfully modified the model code in the previous example, it was not very elegant. We simply copied the existing model into the new one, did some modifications, compiled and packaged it. In other words, we duplicated the code of the original model.

As an object-oriented programming language, Java offers us a much nicer solution. Instead of duplicating the whole model, we will subclass it, overriding only the part of the code that needs to be changed.

We will start with repeating steps from the previous example, copying model files (`LuoRudyModel_I_1991.java` and `LuoRudyModel_I_1991BeanInfo.java`) to `MyModel` files.

But instead of keeping the whole code in `MyModel.java`, we will delete most of it, leaving only:

```
package com.javable.cese.models;

import java.io.Serializable;

/**
 * My Own Model, 2003.
 * Detailed list of equations and model description are provided in
 * <p>
 * Published somewhere.
 */

public class MyModel extends LuoRudyModel_I_1991 implements Serializable {

        /* Functions that describe the currents begin here */

        protected void currentINa() // Calculates Fast Na Current
        {
         super.currentINa();
         setina(getgna() * getm() * geth() * getj() * (getv() - getena()));
        }

}
```

Note the `extends LuoRudyModel_I_1991` declaration — we are extending LuoRudyModel_I_1991 class, altering only one method:

```
protected void currentINa() // Calculates Fast Na Current
        {
         super.currentINa();
         setina(getgna() * getm() * geth() * getj() * (getv() - getena()));
        }
```

We have to keep the same (`currentINa()`) method declaration as in the parent model, in order for subclassing to work. We also call the parent method:

```
super.currentINa();
```

and then change the current formulation using get- and set- methods (see below) to access parameters in the parent model (`LuoRudyModel_I_1991`).

Of course, this is a simplified example. In real life we will have to make much more overriding and more use out of accessing parent model parameters. Check the example models distributed with CESE to learn more.

We can also override the BeanInfo class also, reducing it to:

```
package com.javable.cese.models;

import java.awt.Image;
import java.beans.BeanDescriptor;
import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;

public class MyModelBeanInfo extends LuoRudyModel_I_1991BeanInfo {

        /** Icon for image data objects. */
        private Image icon;

        /** Extended descriptor for this bean. */
        private BeanDescriptor descriptor;

        /** Array of property descriptors. */
        public MyModelBeanInfo() {
                icon = loadImage("model_cardiac.png");
        }

        /** Provides the Models's icon */
        public Image getIcon(int type) {
                return icon;
        }

        /** Provides the Models's descriptor */
        public BeanDescriptor getBeanDescriptor() {
                descriptor = new BeanDescriptor(MyModel.class);
                descriptor.setDisplayName("My Model, 2003");
                descriptor.setShortDescription(
                        "My Own Model, 2003 <p>" + "Published somewhere.");

                return descriptor;
        }

        /** Create a new property descriptor
         * @return Property Descriptor
         */
        private PropertyDescriptor createPropertyDescriptor(
                String name,
                String gett,
                String sett,
                String group) {
                PropertyDescriptor pd = null;
                try {
                        pd = new PropertyDescriptor(
```

```
                          name, MyModel.class, gett, sett);
                    pd.setShortDescription(group);
            } catch (IntrospectionException ex) {
            }
            return pd;
      }

      /** Descriptor of valid properties
       * @return array of properties
       */
      public PropertyDescriptor[] getPropertyDescriptors() {
            return super.getPropertyDescriptors();
      }
}
```

Note the `extends LuoRudyModel_I_1991BeanInfo` at the beginning and how we rewrote `getPropertyDescriptors()` method.

Now we can compile our source code and package .class files into the jar archive, exactly like in the previous example. Our model is ready for the installation and execution in CESE.

4. Creating a new model

Every once in a while you need a completely new model that has nothing to do with CESE example models. I would still recommend to try the examples above, because you will perform essentially the same operations. It is also a good idea to familiarize yourself with the models source code.

All models that are being used by CESE are overriding the special class `com.javable.cese. templates.AbstractAPModel`. Take a look at this class API documentation (you will find it in the `$CESE_HOME/doc` directory). This class provides a number of parameters and methods that are necessary for the proper lifecycle of the model.

The core method that needs to be overridden in your model is `doSimulate()`. It's being called in every time step during simulation.

A typical `doSimulate()` implementation looks like:

```
protected void doSimulate() {
            setCurrents();
            setConcentrations();

            vnew = v - it * dt;
            dvdtnew = (vnew - v) / dt;
            t = t + dt;
            v = vnew;
            dvdt = dvdtnew;
}
```

where `setCurrents()` calls all the current calculation routines and `setConcentrations()` calls methods that update ionic concentrations. Again, you can take a peek at any of the prepackaged models to see how it's done. Even though you are free to use any name you want for your methods that calculate currents and concentrations, it's better to follow the convention used in the "standard", prepackaged models.

Now the JavaBeans specification comes to play. You probably will have lots of model parameters that you want to be able to modify and control. Notice, that these parameters (or "fields") are declared as "private" and wrapped in the getter and setter methods, for example:

```
private double gna = 4.0; // Max. Conductance of the Na Channel (mS/uF)
        public synchronized void setgna(double sgna) {
                gna = sgna;
        }
        public double getgna() {
                return gna;
        }
```

> Method names are very important — they should correspond to the field (e.g. "gna") name.

You can also make a field "read-only". Simply omit the setter method, for example:

```
private double ah; // Na alpha-h rate constant (ms^-1)
        public double getah() {
                return ah;
        }
```

It is up to you to decide what parameters should be read-only, and what should be writtable. Keep in mind, that the parameter that should be accessible for clamping or modification via the user interface should be writtable. It is also a good idea to make Ionic Current, Max Conductance, and Equilibrium Potential parameters for the given current writtable — it may come useful when you or somebody else will try to subclass *your* model.

Now we are ready to roll our own BeanInfo. Again — copy-pasting the existing one is not a bad idea, since it contains declarations for the commonly-used parameters, such as Time, Stimulation, Membrane voltage, Total current, etc.

If we have a new current with parameters wrapped in getters and setters we also want to declare the human-accessible form for these parameters in the BeanInfo. A typical declaration looks like:

```
createPropertyDescriptor ("Intracellular Na concentration (mM)",
        "getnai", "setnai", "Ion Concentrations")
```

Where the first string, "Intracellular Na concentration (mM)", declares the human-readable

name for the parameter; "getnai", "setnai" are the names of getter and setter for the field "nai". This is a writtable property, therefore we have a "setnai" method.

> Make sure that you actually have this method in the model code! Otherwise the model will not be installed.

Finally, "Ion Concentrations" is a group label — for convenience we group parameters with the similar meaning.

After we install our model, parameter "Intracellular Na concentration (mM)" will appear in the table of model parameters and will be available for modification and clamping.

You may have also noticed some strange declarations starting with "|OUT". For example:

```
createPropertyDescriptor ("|OUT Fast Na current (uA/uF)",
          "getina", null, "Fast Sodium Current (time dependent)")
```

These are the output, or display parameters that will appear in the Display tree (where user selects parameters for the output and visualization).

> These properties are being declared as read-only, with the setter name set to "null".

After the BeanInfo file is created, you can compile and package your model in the jar archive.

To summarize, using JavaBeans as a foundation for models gives a number of advantages:

- All model parameters ("properties") can be exposed to the simulation environment. This way the environment can provide a uniform user interface for the model parameters control and modification. In addition, because JavaBeans is the industry standard, 3-d party tools (for example, Java IDEs) can work with CESE models as well.
- Using JavaBeans for CESE models allows simple saving and restoring of model parameters using Java Serialization. This comes as a free feature to you, a developer, as long as your models are JavaBeans-compatible.
- Java provides a convenient way for packaging and distributing JavaBeans in the Java archives (jars). These .jar files can be copied or sent over the web just like .zip archives. You can share your custom models with your colleagues.

5. Exporting data from the model — data exporters

Simulations are used to generate the data. Your simulation results should somehow be made available to CESE or third party tools. For this purpose, CESE has a flexible mechanism for

taking the data from the model and redirecting it to CESE or to the external processing program — data exporters.

The base class for the data exporters is `com.javable.cese.templates.AbstractDataExport` — the abstract class that outlines a skeleton for data export and interaction of data exporters with models.

The three core methods in `AbstractDataExport` are:

```
public abstract void prepareExport();
```
this one is being called before the main simulation loop in a model begins (take a look at the `AbstractAPModel` source code). Override this method in your own DataExport implementation to create and initialize data arrays, open output files, connect to databases, etc.

```
public abstract void doExport(int count);
```
is being called at every time step during model execution. Implementations of this method will take the chunk of data from the model and put it into the output arrays, files, databases, etc. Parameter `count` gives you a number of the current simulation step for convenience. Finally,

```
public abstract void finishExport();
```
is being called after the simulation loop has ended, giving your code a chance to close output files, commit to the database and perform other cleanups.

CESE provides its own data exporter that communicates with the user interface: `com.javable.cese.DataExport`. If you plan to use CESE as your simulation environment, writing custom data exporter is not required, but it may come handy for some special output types that you need in a future.