DDABM-SOLAR Documentation

Haifeng Zhang and Yevgeniy Vorobeychik
Electrical Engineering and Computer Science
Vanderbilt University
{haifeng.zhang, yevgeniy.vorobeychik}@vanderbilt.edu

April 15, 2015

Introduction

DDABM-SOLAR is an agent-based rooftop solar diffusion model that is implemented in Repast Simphony. It is originally developed to forecast individual and aggregate solar adoption. In particular, the model itself consists of several components, such as solar system capacity model, system cost(buy/lease) model, electricity consumption model and adoption decision model. Notably, all these models are learned from empirical data using machine learning, following our proposed data-driven agent-based modeling (DDABM) methodology (see the paper below [1]). The current implementation also includes functions, such as, model sensitivity analysis to incentive budget, one-parameter incentive optimization and seeding policy optimization. For more details about our DDABM and policy optimization, please refer to our following publication:

[1] *Haifeng Zhang, Yevgeniy Vorobeychik, Joshua Letchford, and Kiran Lakkaraju. Data-Driven Agent-Based Modeling, with Application to Rooftop Solar Adoption. In AAMAS-15: Proc. 13th Intl. Joint Conference on Autonomous Agents and Multiagent Systems, 2015.*

Download

The source code is now publicly available on github under the MIT license. Users can obtain a copy of the source code in one of three ways:

1. http clone: `https://github.com/haffwin/ddabm-solar.git`
2. svn checkout: `https://github.com/haffwin/ddabm-solar`
3. ssh clone: `git@github.com:haffwin/ddabm-solar.git`

Installation

DDABM-SOLAR is developed on Repast Simphony ("Repast" hereafter) simulation platform, before running the model, user has to have Repast installed. If you have any questions about how to download and install Repast on your local system, please refer to Repast website as follows:

`http://repast.sourceforge.net/repast_simphony.php`

Once you have Repast installed, you can obtain a copy of our project source through one of three ways as shown above. For users using Apache Subversion (SVN), typical steps to install DDABM-SOLAR is as follows:

1. Open SVN repository perspective and add a new repository location: `https://github.com/haffwin/ddabm-solar`. Once the location is correctly added, it will be shown in your SVN repository perspective (see Figure 1)

2. Select the newly-added location and locate "trunk/solarPanelAdoption", where all project source is currently located at (see Figure 2).

3. Right click and choose "Find/Check Out As", seen in Figure 3. In the opened the "Check Out As" window, shown in Figure 4, choose "Finish". This will create us a copy of our DDABM-SOLAR project namely "solarPanelAdoption", as shown in your Repast package explorer (see Figure 5).

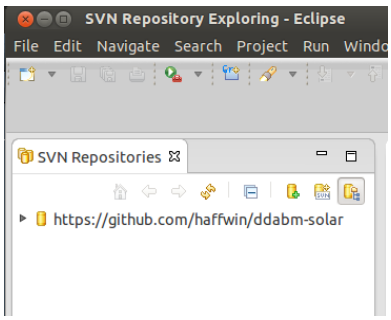

Figure 1: New SVN Repository Location

In Figure 6, we can see all the classes currently defined in "solarPanelAdoption" package. A summary of main purpose of each of these classes is as follows:

1. *HouseHold*: a basic household with home characteristics and other variables

2. *Adopter*: solar adopter inherits HouseHold

3. *NonAdopter*: solar non-adopter inherits HouseHold and makes solar adoption decisions

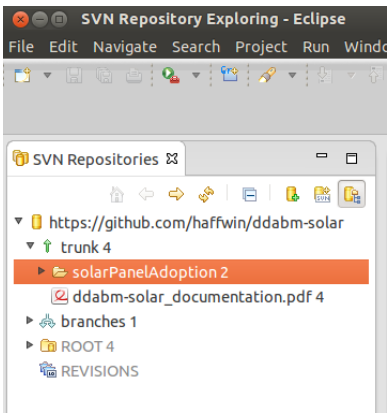

Figure 2: SVN Repository Structure

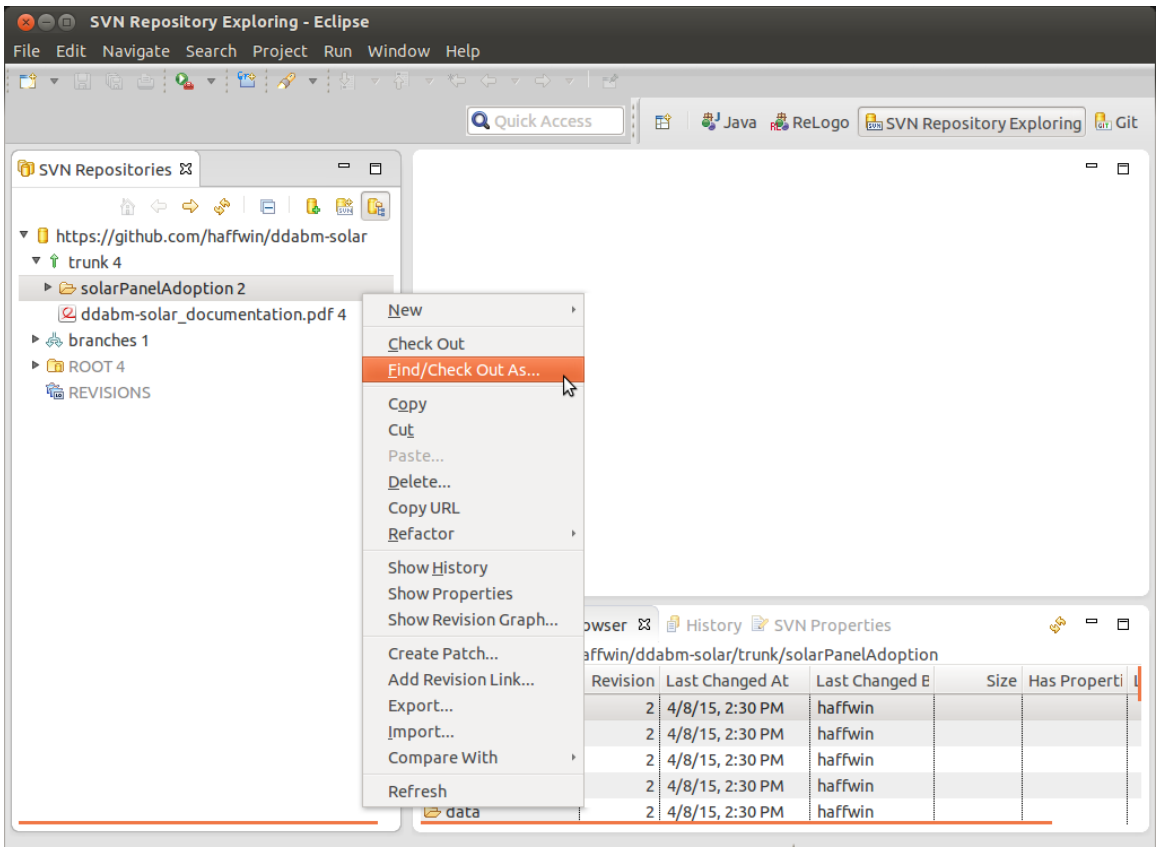

Figure 3: SVN Repository Check Out

4. *Updater*: a basic update agent that is scheduled to periodically update variables for each Household agent

5. *UpdaterAveStep*: a particular type of Updater agent, which utilizes "average step" idea to obtain expected adoption

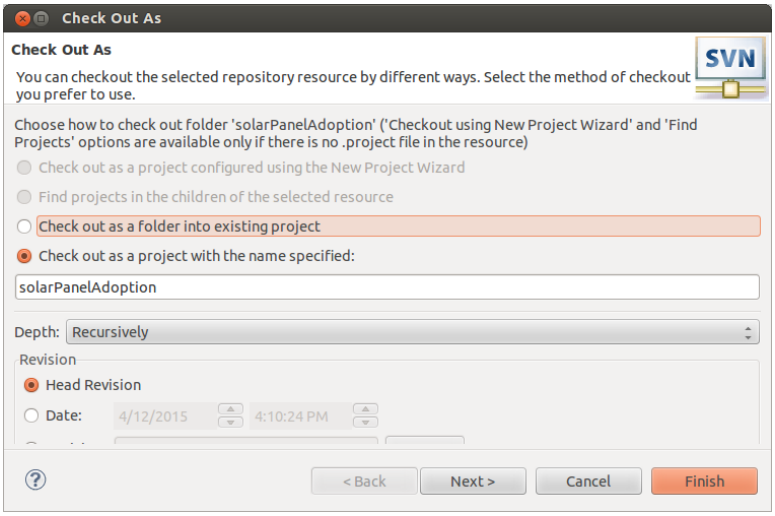

Figure 4: Check Out As

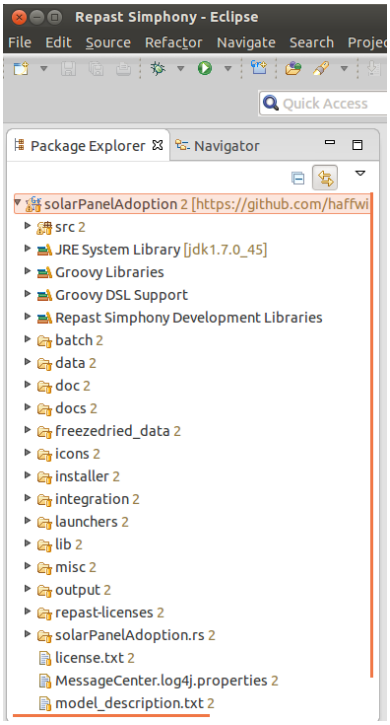

Figure 5: DDABM-SOLAR Shown in Repast Package Explorer

6. *solarPanelAdoptionBuilder*: a class used to initialize the ABM
7. *NegativeVariableException*: a class used to capture and handle negative value exception
8. *CsvFileReader*: a class used to read a CSV file into an array

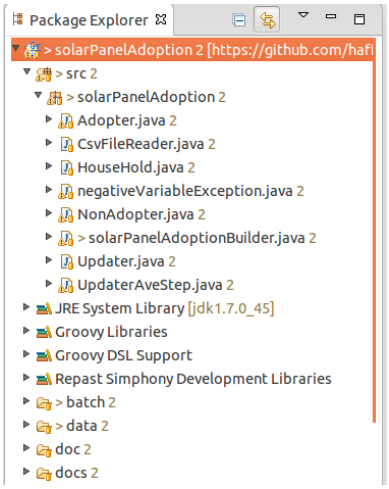

Figure 6: DDABM-SOLAR Java Classes

Demo

At this point, we assume that you have correctly installed Repast and downloaded the source of our DDABM-SOLAR model, which is essentially a Java Eclipse project. So, if you have some exposure to software development in Eclipse, the following procedures should be very familiar to you. In this section, we will show our users how to make predictions, how to test model sensitivity to CSI budget, how to do one-parameter incentive optimization and seeding policy experiments using our DDABM-SOLAR model.

Prediction beyond the Time-frame of Training

In your Repast GUI, at the top, locate "Run As" button and click on its dropdown list, then select "solarPanelAdoption Model" as shown in Figure 7. Your Repast project: *solarPanelAdoption* will be compiled and run, a new window will appear as seen in Figure 8. This is the standard GUI for all projects developed in Repast and major interface for our DDABM-SOLAR model as well.

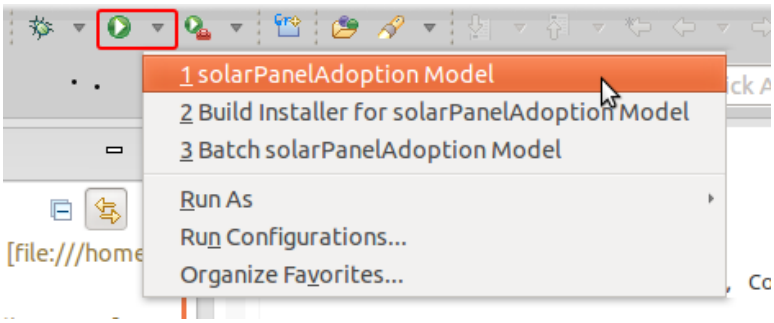

Figure 7: Run Repast Project

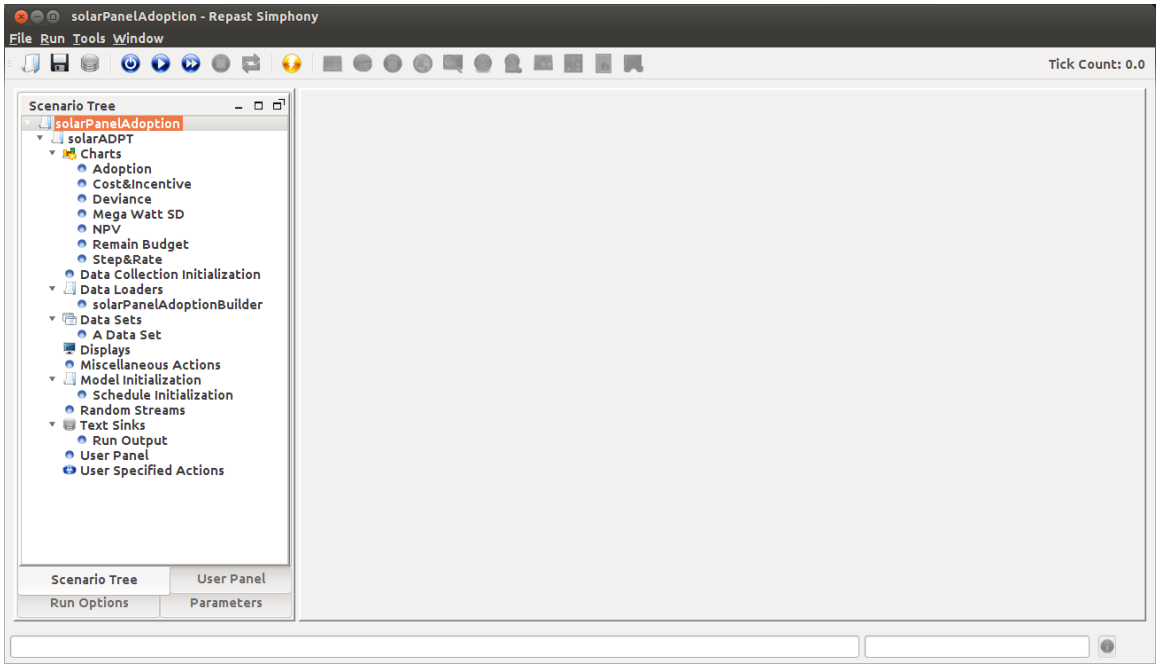

Figure 8: Main Model GUI

In Figure 8, on the left, let us first focus our view on the "Scenario Tree" (see Figure 9. This is where we define general settings of our ABM. In details, we can specify the Java class that is used to initialize agents (under "Data Loaders", that is "solarPanelAdoptionBuilder"), what information will be displayed while the ABM is running (via "Charts and Displays"), and what output can be expected (via "Text Sinks"). For example, as shown in Figure 9, we have added some items under "Charts", which are time-series charts for a set of variables that we are interested in and briefly described as follows:

1. Deviance: model deviance, i.e., -2*log likelihood
2. Mega Watt SD: cumulative installed capacity in mega Watts
3. NPV: net present value for ownership and lease
4. Remain Budget: remaining budget in seeding policy experiments
5. Step&Rate: CSI step and rate
6. Adoption: ZIP code solar adoption
7. Cost&Incentive: system cost and incentive

How items under "Charts" are related to our project source? The secret is in "Data Sets", notice, under which there is a item called "A Data Set". To help you better understand the connection, please double click item "Adoption" under "Charts", that will open "Time

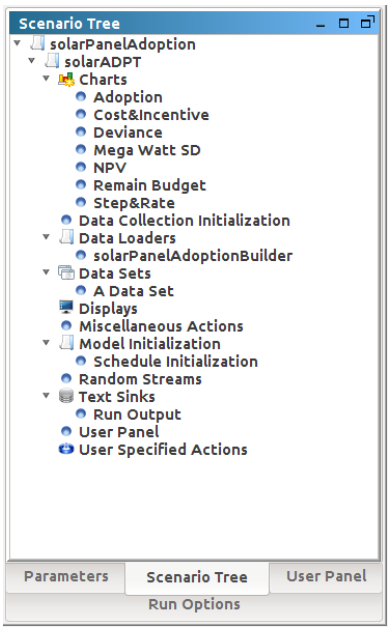

Figure 9: Scenario Tree

Series Editor" (see Figure 10). In Figure 10, we have chosen tab "Configure Charts Data Properties", which shows all variables displayed in the chart (note that variable names are shown in column "ID"). These variables must be predefined in "Data Sets". For example, let us keep variable "numAdopt" in mind and close "Time Series Editor". Then, we double click "A Data Set" and open "Aggregate Data Editor" (see Figure 11), and choose tab "Select Data Sources" on the left and then tab "Method Data Sources". You can find a variable with same name "numAdopt" in clolumn "Source Name". The entry tells us variable "numAdopt" is defined by summing over ("Sum" in column "aggregate operation") returned values from a method called "getCount"(in column "Method") that is defined in your Java class called "Adopter" (in column "Agent Type"). Generally, to define a new variable, i.e., display it in a chart or output it to file, user just simply clicks on "Add" button, in Figure 11 and choose appropriate agent type, method and aggregate operation. In addition, a variable can be removed by clicking on "Remove" button.

You can also add a spatial display for the DDABM-SOLAR model. To do so, we first right click "Displays" in "Scenario Tree" and choose "Add Display". In the popped window, edit "name" as "GIS Display" and choose "Type" to "GIS", and add projection "geography" by clicking on the green button (see Figure 12). Then we click "Next" button to choose agent style and other properties. Finally, for more general questions about how to setup the scenario tree, we strongly suggest you read the following Repast Java tutorial:

`http://repast.sourceforge.net/docs/RepastJavaGettingStarted.pdf`

Having a basic idea about "Scenario Tree", let us now click and select another panel called "Parameters" (see Figure 13). This panel includes all values we can pass to our ABM as parameters.

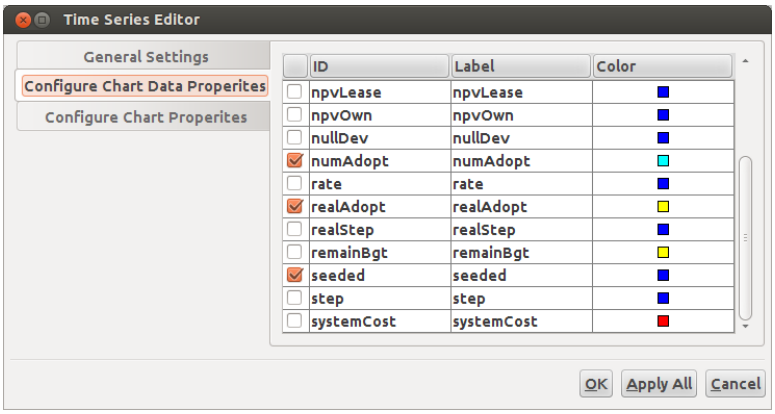

Figure 10: Time Series Editor

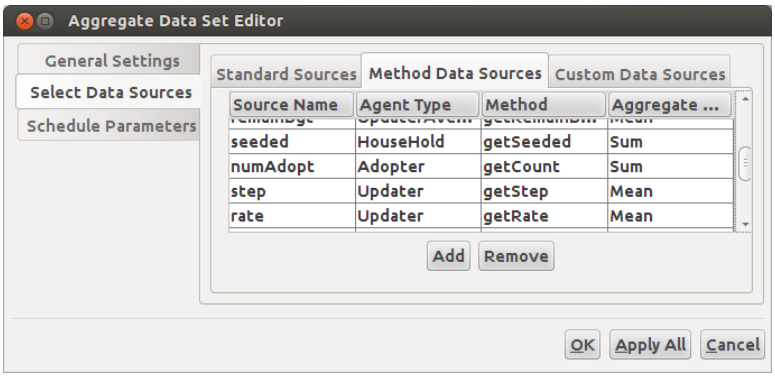

Figure 11: Aggregate Data Set Editor

Particularly, for prediction, now, we only need to modify the last parameter, called "Start From", from which ABM will start, and keep any other parameters to their default values, shown in Figure 13. From paper referenced earlier [1], our model is trained on the first 48 months of the time period we considered: from May 2007 through April 2013, and used to predict adoption for the last two years. In this case, we set "Start From" to 48. As a result, the model will be initialized to the adoption status at end of the 48th month. Note that, in Figure 13, user can also add new parameter in this panel by clicking "+" button. To define the new parameter, in Figure 14, user can specify variable name ("Name"), display name ("Display Name"), data type ("Type") and default value ("Default Value") etc. Moreover, to make sure our ABM can receive the parameter, we must use the same variable name (not "Display name") in Java source file. For further information, please refer to project source file: *solarPanelAdoptionBuilder.java*, which contains multiple examples of getting parameters from model GUI by first creating an instance of "Parameter" object via "RunEnvironment.getInstance().getParameters()" and then calling its "getValue" method. In addition, user can find useful information from above Repast Java tutorial as well.

Once all parameters are set up, we are ready to initialize ABM run. This can be done by clicking on the "Initialize Run" button in our model GUI (see Figure 15). The plot area

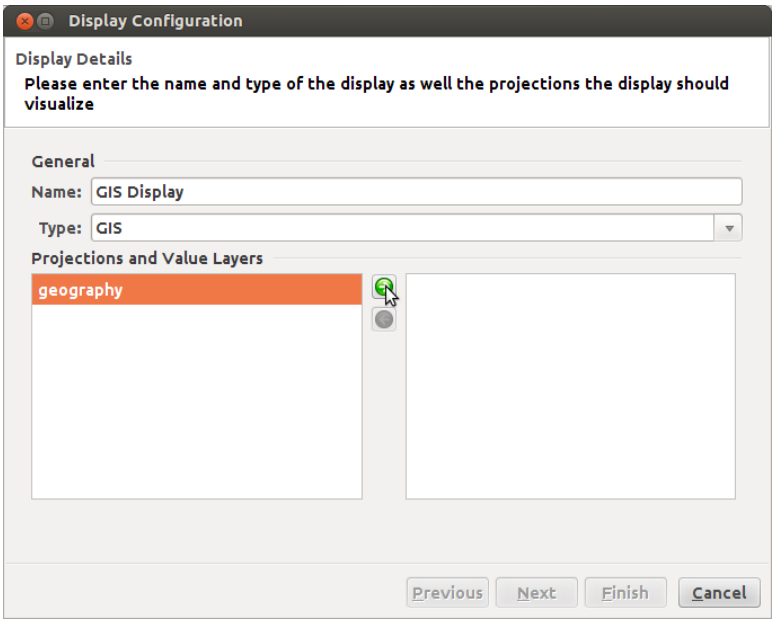

Figure 12: Display Configuration

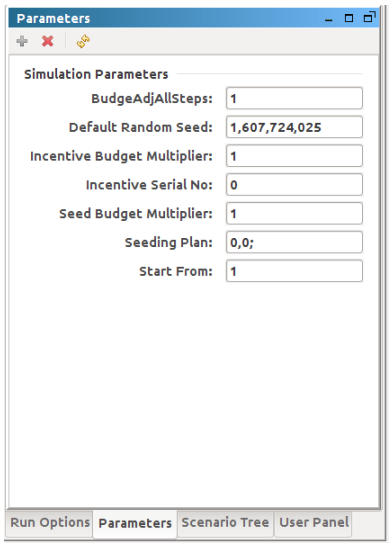

Figure 13: Parameters Panel

on the right of the model GUI will be updated accordingly as shown in Figure 16. Notice that the plot area includes multiple charts, which correspond to all items under "Charts" in "Scenario Tree". Once our ABM is initialized using parameters given in "Parameters" Panel and configured by settings in "Scenario Tree", it can be run by clicking the "Start" button (Figure 17). You should notice that all the charts are now updated simultaneously as your ABM runs. Notice, the ABM can also be run one step a time by choosing "Step Run" (i.e., double-arrow button right next "Start" button). If your simulation is running,

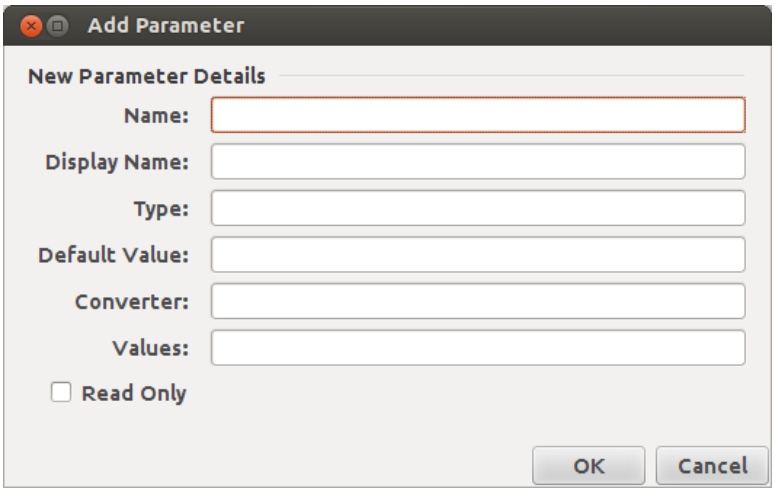

Figure 14: Adding Parameter

you can also terminate it using "Stop" button (i.e., button with a square on right after "Step Run" button). In addition, parameters can be reset by clicking on "Reset Run" button (i.e., button right after "Stop" button). For example, in Figure 18, the chart shows the simulated adoption v.s. actual adoption. Figure 19 shows how solar panel system cost evolves over time. After your simulation is terminated, some output will be generated and saved under a folder called "output" right under your project root directory. The user can also modify output directory and file format by double-clicking on the "Run Output" item under "Text Sinks". Again, more details can be found from above Repast tutorial.

Examining Model Sensitivity to CSI budget

Sensitive analysis to CSI budget is designed by multiplying current CSI rates but fixing step mega Watt target. This can be done by giving different parameters in "Parameters" panel. Particularly, we can set "BudgetAdjAllSteps" to a value, which is a multiple of current CSI rate. For example, if we want to examine adoption trend in a situation that CSI incentives are removed from the system, i.e., simply setting "BudgetAdjAllSteps" to zero (Figure 20). Following the same procedures shown earlier, user can initialize, start ABM runs and obtain the results.

One-parameter Incentive Optimization

To enable one-parameter incentive optimization, user need to modify and recompile the source code and run the model. Please look for a boolean variable called `APPLY_OneParBic`, in your Java source: *solarPanelAdoptionBuilder.java* and set it to *true* (Figure 21). Compile your project and run the model until its main GUI appear, and then navigate to the "Parameters" panel, as we have shown previously. This time we will make use of variable "Incentive Serial No". Its default value is zero, which corresponds to original CSI incentive

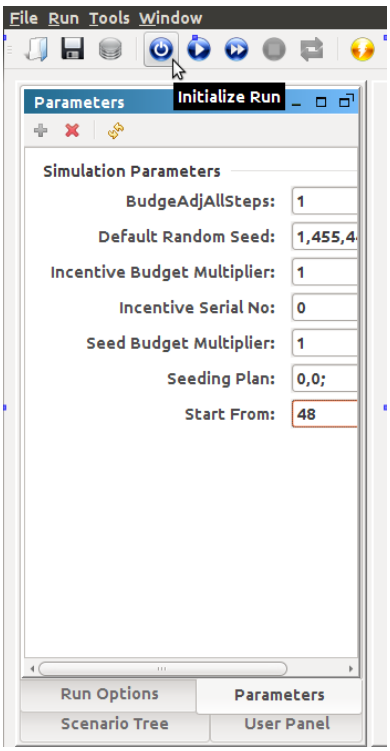

Figure 15: Initialize Run

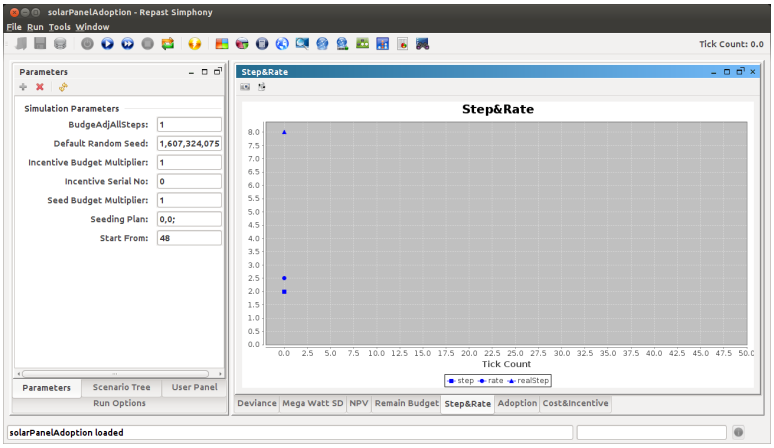

Figure 16: ABM Run Initialized

schedule and also the first row (i.e., 0 indexed) of a rate schedule file. This external file contains all rate structures we want to test in the scheme of one-parameter optimization, i.e., all with different initial rates and each decays exponentially in 9 steps. The file is called: `one_par_ebpp_rates.csv`, which is located in folder: *[path to your project]/data/models*. Moreover, for different rate structure, one can test different incentive budgets by setting different value to parameter "Incentive Budget Multiplier", as the name suggests, that is a

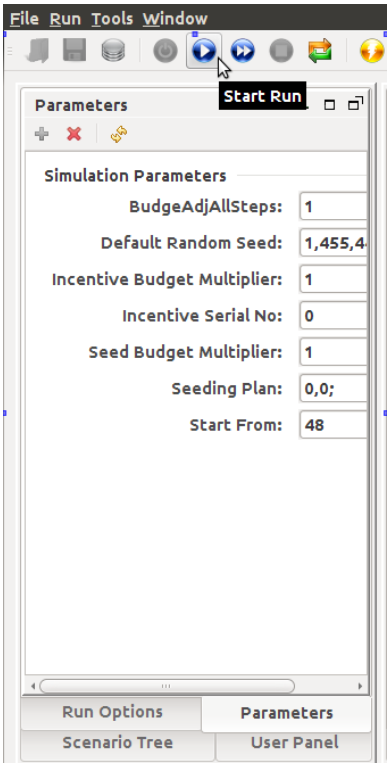

Figure 17: Start ABM Run

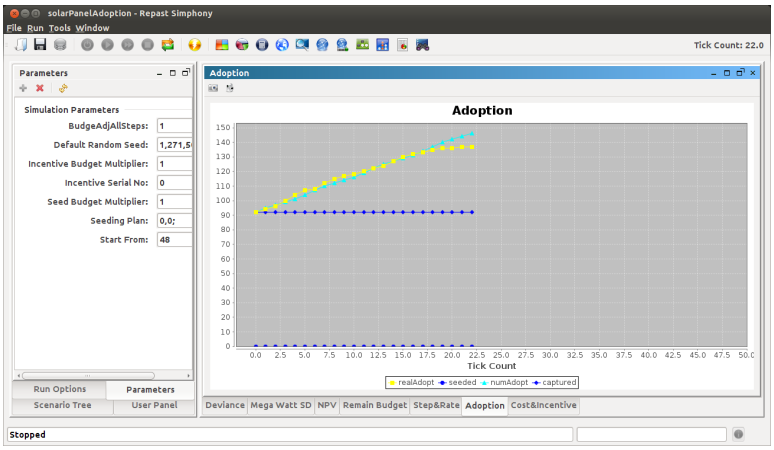

Figure 18: ABM Time-series Chart: Adoption

multiple of CSI budget. For more details about the one-parameter incentive optimization, we suggest you read our AAMAS paper mentioned at beginning of the documentation.

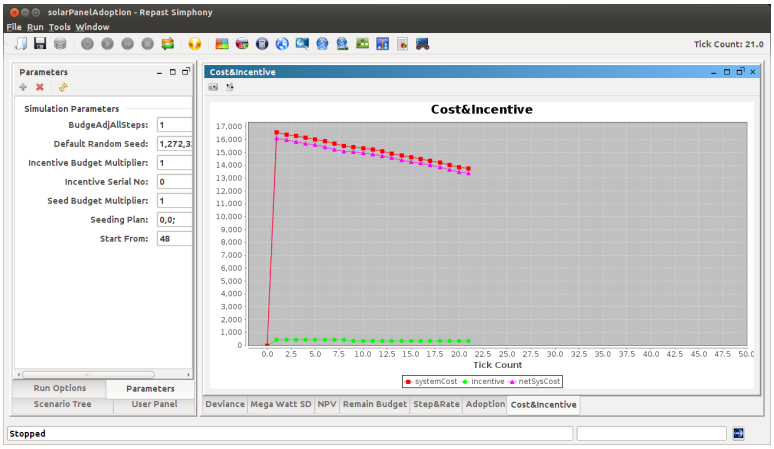

Figure 19: ABM Time-series Chart: System Cost

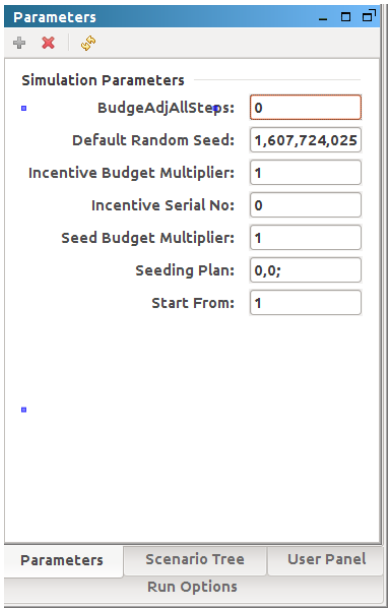

Figure 20: Test Scenario of Non-Incentive

Seeding Optimization

To enable seeding policy experiments, in our model, the user needs to slightly modify project source file. A boolean variable is called `APPLY_SeedingPlc`, which is grouped together with variable `APPLY_OneParBic` involved in last section (Figure 21). Set this variable to *true* and make `APPLY_OneParBic` to *false*. Note that, both variables can not be set to *true* at same time, since we do not want to mess up one-parameter optimization with seeding experiment. Formally, we define a seeding policy as a sequence of fractions of budget allocated to each month. Concretely, a seeding policy is encoded as a string in Java implementation. For example, "0, 0.1; 2, 0.3; 5, 0.6" represents a seeding policy which seeds 0.1 of budget at

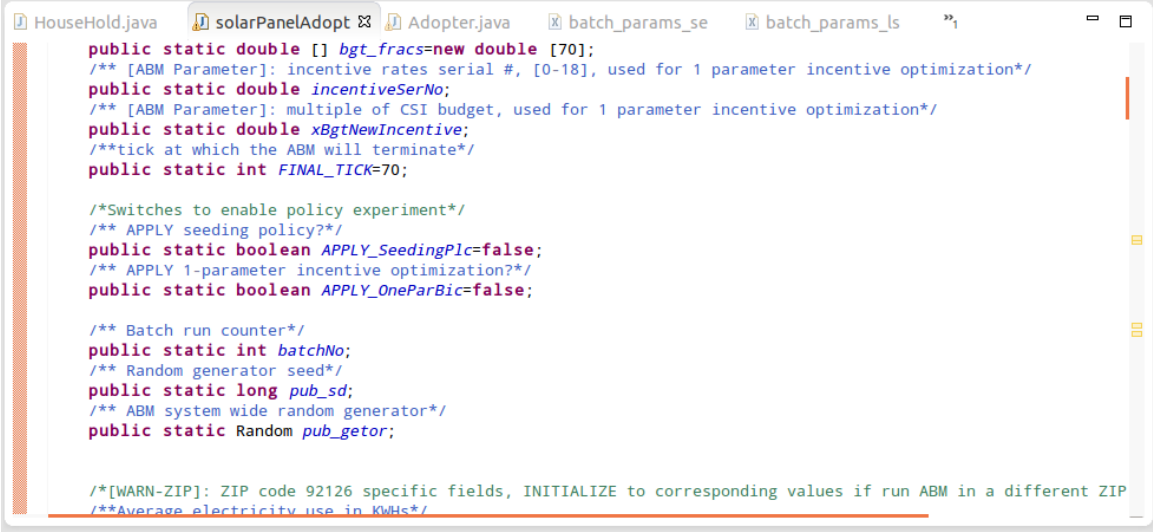

Figure 21: Switch to Different Mode

stage 0, 0.3 at stage 2 and 0.6 at stage 5. Moreover, one can specify a multiple of original CSI budget for seeding via parameter "Seed Budget Multiplier". Likewise, user can initialize and start the run, and observe aggregate adoption trend for any arbitrary seeding policy as we defined. As a second example, in seeding experiments illustrated in the AAMAS paper mentioned at very beginning of the documentation, we aim to find an optimal fraction applied in stage 0 given that the rest of budget will be used to at the stage right before the last stage. Your seeding string will be something like: "0, f; 68, 1-f", where f is a fraction. And, we want to find the optimal fs for different scales of CSI budget.

Batch Run

All examples given so far are based on Repast single run GUI. Sometimes, if we need large sample runs or evaluate a number of candidate policies, this is not appropriate. In dead, Repast has another GUI,"batch run", which allows us to configure a particular batch run that can be run either locally or remotely. In the course of developing our DDABM-SOLAR project, we have attempted to leverage the batch run interface, but ended up with unresolved technical issues. To this end, we only use Repast batch run GUI to generate batch parameter file and but use a Linux shell script to call the main Repast batch run routine.

Large Sample Runs

In this section we will show how to make large sample runs, e.g, 1000. In your Repast/Eclipse environment, under "Run As" choose "Batch solarPanelAdoption Model", seen in Figure 22, that will launch Repast batch run GUI (Figure 23). By default, it first shows "Model" tab, which includes "Model properties" and "Run properties". Let us use the default settings

on this tab and choose "Batch Parameters" tab (Figure 24). You will see the exactly same parameters as shown in "Parameters" panel in the single run mode. In Repast batch run mode, each parameter can be assigned as a constant, a number range or space separated list. If you give a number range to a parameter, the model will be initialized with each case and run respectively. Moreover, if you have several parameters that are specified in number ranges or lists, Repast will generate all possible cases in the parameter space and run each case accordingly.

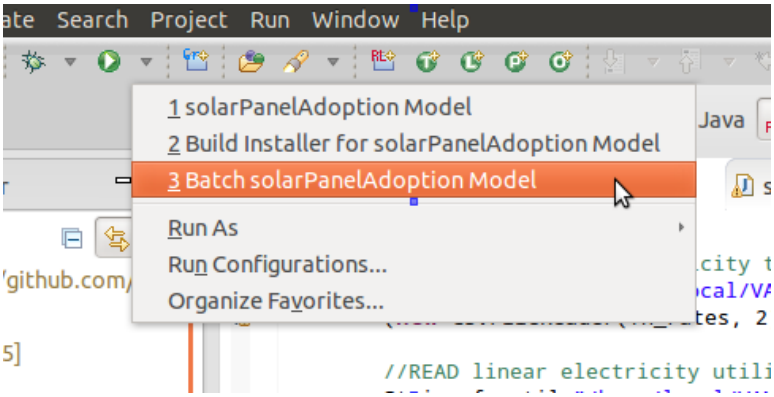

Figure 22: Batch Model

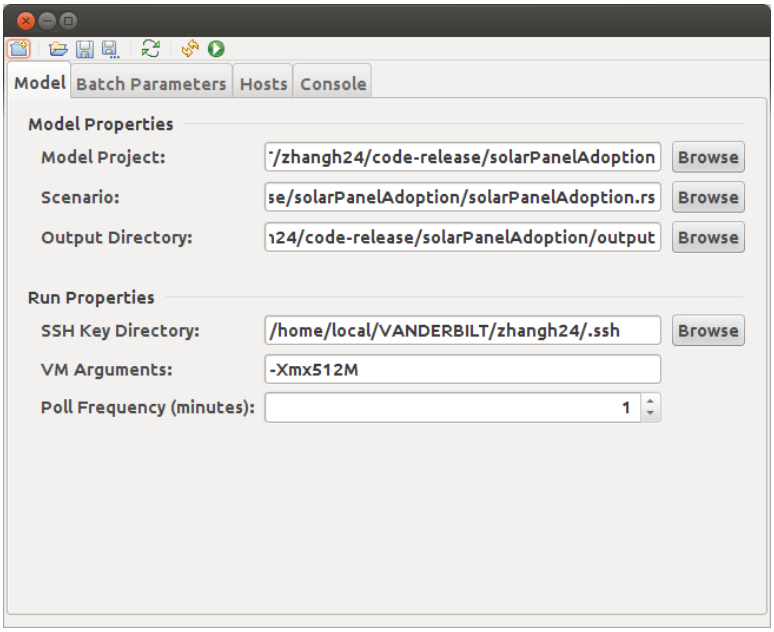

Figure 23: Batch Run GUI: Tab model

In Figure 24, click on "Generate" button, which will save batch parameters to: `[path to your project]/batch/batch_parms_lsr.xml`, as shown in field: "Batch Parameter File".

Then, in your Repast package explorer, go to that directory and open the saved parameter file with XML editor. To specify number of runs for each instance of parameter set, we just need to modify field named "runs" (Figure 25). To run 1000 sample runs in a batch, let us set it to 1000 and save the XML file. The modified file will be used as an argument for our Linux shell script, `runbatch_direct.sh`, which is located at folder:*[path to your project]/batch/*. Typically, in you Linux terminal, Repast can be run in batch mode as follows:

```
sh runbatch_direct.sh batch_params_lsr.xml
```

Note that to make the shell script work on your own computer, you have to replace the path in the script with the actual path in your local system (see the shell script for more details). Also, before running the script, make sure that the variable called `AVERAGE_RUN` in *SolarPanelAdoptionBuilder.java* is set to 1, which indicates the ABM will be run totally random without averaging multiple (≥ 2) outcomes in each step.

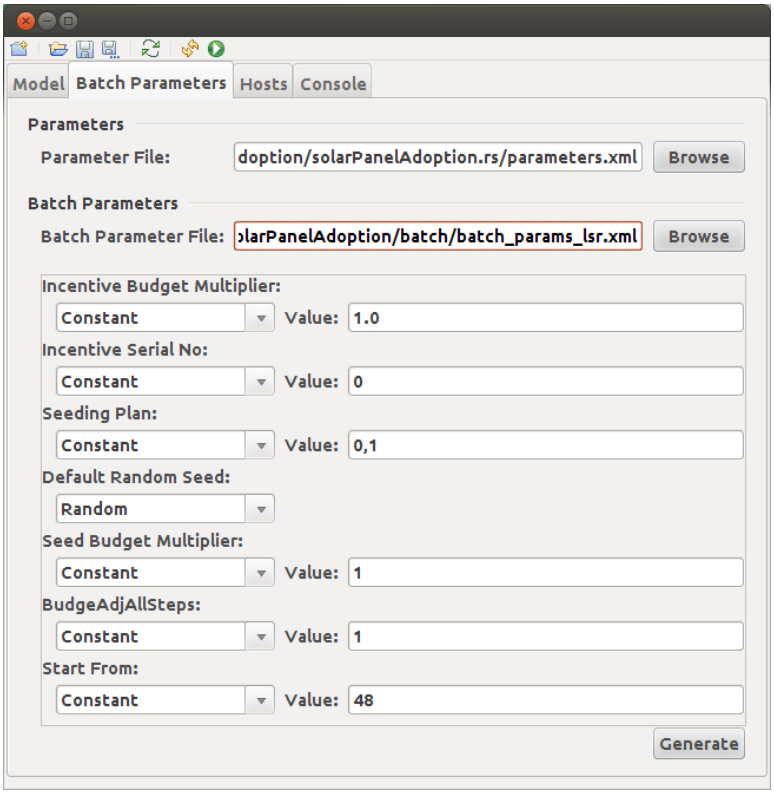

Figure 24: Batch Run GUI: Tab Batch parameters

Large Parameter Space

We can use Repast batch run to generate results for large parameter space, which turns out to be very convenient in case of one-parameter incentive optimization and seeding policies

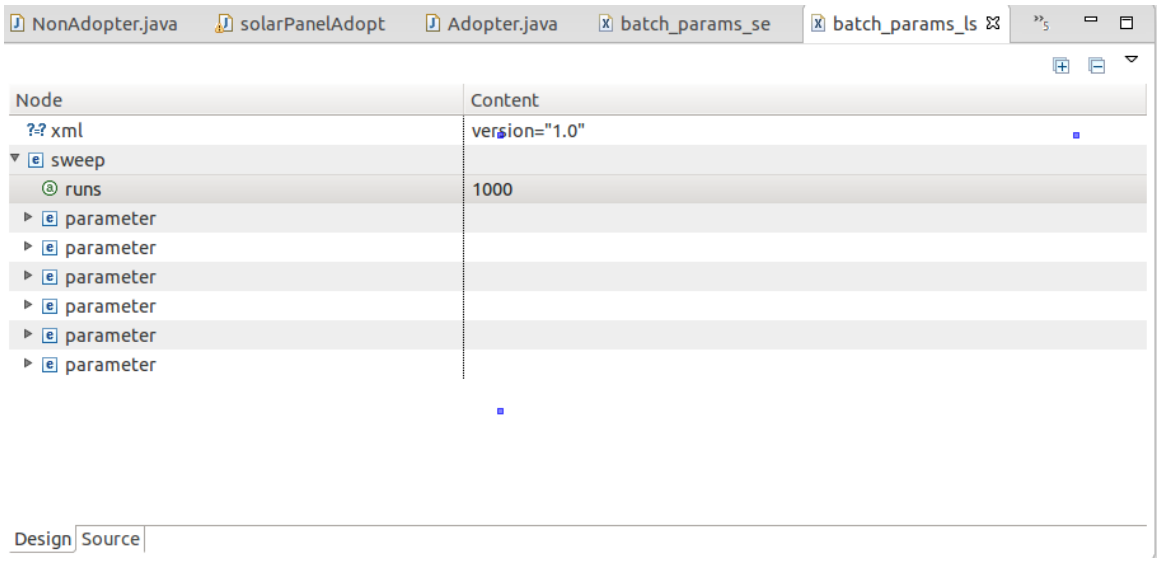

Figure 25: Batch Parameter XML File

experiments. For example, a user wants to test all K candidate one-parameter incentive structures for different budget scales, such as, 1, 2 and 4. In Figure 26, we can just give a number range to parameter "Incentive Serial No", i.e., 1 to $K(= 9)$ with step 1, and a list i.e., "1 2 4" to parameter "Incentive Budget Multiplier", and run Repast in batch mode using script we provided above. Further, if we want to average results of large sample runs, similarly, we can modify "runs" filed in the batch parameter XML file. What if we have large number of seeding policies to evaluate? Recall that a seeding policy is represented as a string. User can either generate those policy strings on-line or store them somewhere and then call Repast batch run routine with corresponding parameter XML file as an argument.

Future Development

The current DDABM-SOLAR model can be easily alternated to run in a different ZIP code and also flexible enough to plug in alternative models.

Applying to an Arbitrary ZIP code

To apply our current ABM to an arbitrary ZIP code, the first thing is to prepare agent property dataset for the zip code. Notice that our current implementation relies on GIS shape file to initialize agents, i.e., all properties of agents are encoded in GIS shape file that can be further processed by Repast. More details about how to load shape file can be found in source file: *SolarPanelAdoptionBuilder.java*. If your shape file is successfully loaded, Repast will create agents according to properties encoded in the shape file. In particular, those properties include GIS coordinates of each agent or home, that can be processed very efficiently using

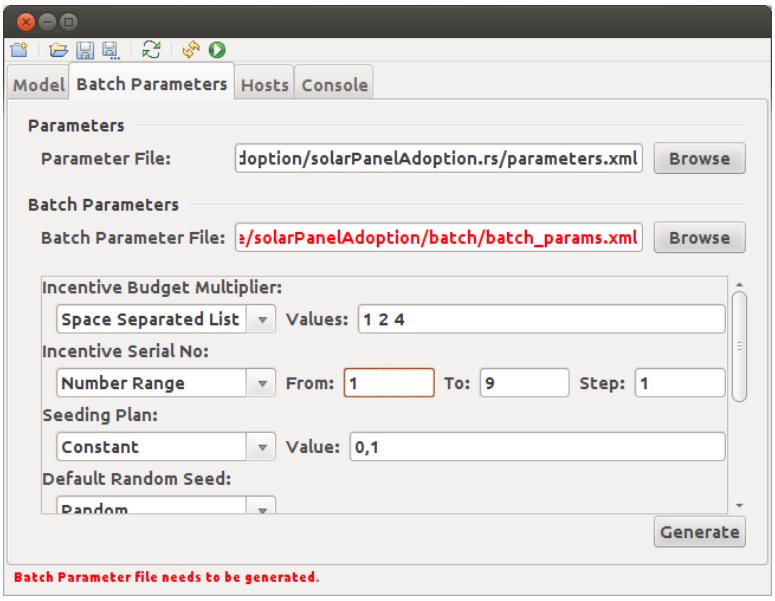

Figure 26: Batch Parameters of One-parameter Incentive Optimization

Repast GIS package. By convention, all shape files are stored in directory: `[path to your project]/data/agents`. Also, please note that the current shape files available on GitHub are not the actual agents but synthetic agents used for illustration only. Thus, one should not expect the same results seen in the AAMAS paper referenced earlier [1].

Suppose your initial agent property dataset is in comma-separated values (CSV) format. To convert it to GIS shape file, for example, we currently make use of a free software called QGIS Desktop. For more information about this tool, please refer to the following website:

`http://www.qgis.org/en/site/`

Note that, user should be informed about what coordinate reference system (CRS) the GIS coordinates in your CSV file are associated with. In QGIS Desktop, general steps to make shape files from CSV files are as follows:

1. Open QGIS desktop;
2. From menu, choose “Layer-Add Delimited Text layer” (seen Figure 27);
3. In the opened window, choose CSV file and specify ”Geometry definition” properly (see Figure 28);
4. In ”Coordinate Reference System Selector”, choose correct CRS (see Figure 29);
5. Right click the added layer and select ”Save as” (see Figure 30);
6. Choose file name and location, and save the shape file (see Figure 31).

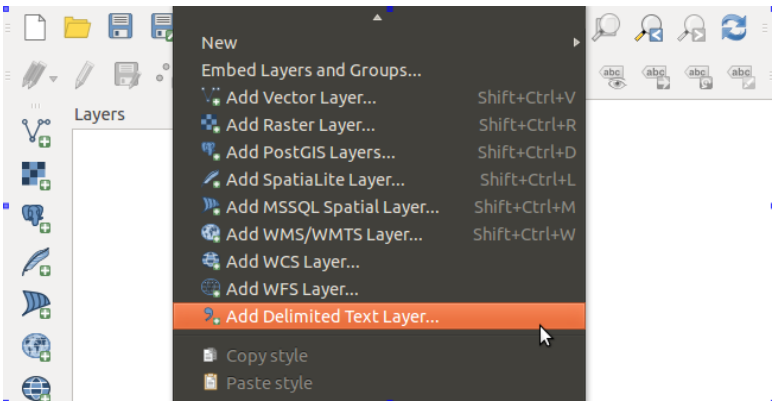

Figure 27: Add Layer thru Menu Option

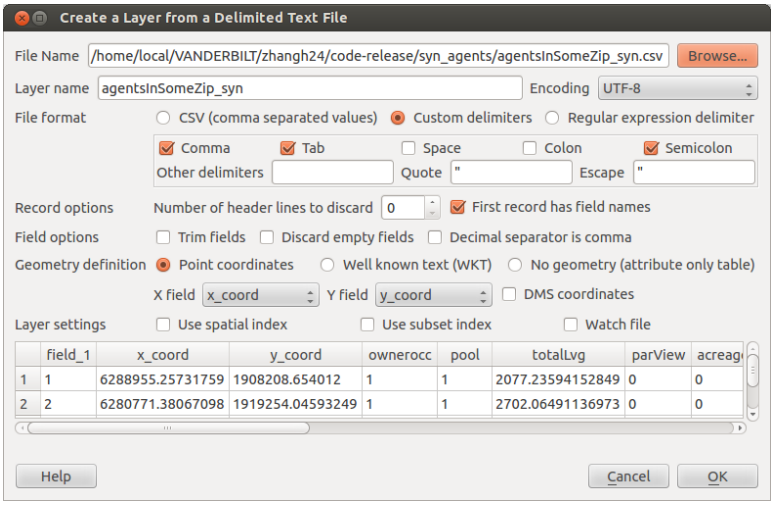

Figure 28: Create a Layer from a Delimited Text File

Once the shape files are made, developer needs to look for values in the source files that are particular to a ZIP code and replace them with those valid for the your desired ZIP code. Of course, this can be done in a more efficient way, i.e. putting all zip code specific information in a single configuration file. However, to this end, we leave the decision to be made by future developers. Nevertheless, while we write the code, we intentionally grouped all ZIP code specific fields together and add special comments to ease fast reference. Particularly, there are three Java source files which contain ZIP code specific information, which are: *HouseHold.java*, *SolarPanelAdoptionBuilder.java* and *UpdaterAveStep.java*. Developers simply need to search for key word "WARN-ZIP", then the particular information for ZIP code can be located immediately. The comments in the source also provide further guidelines of application to different ZIP code; please read them carefully.

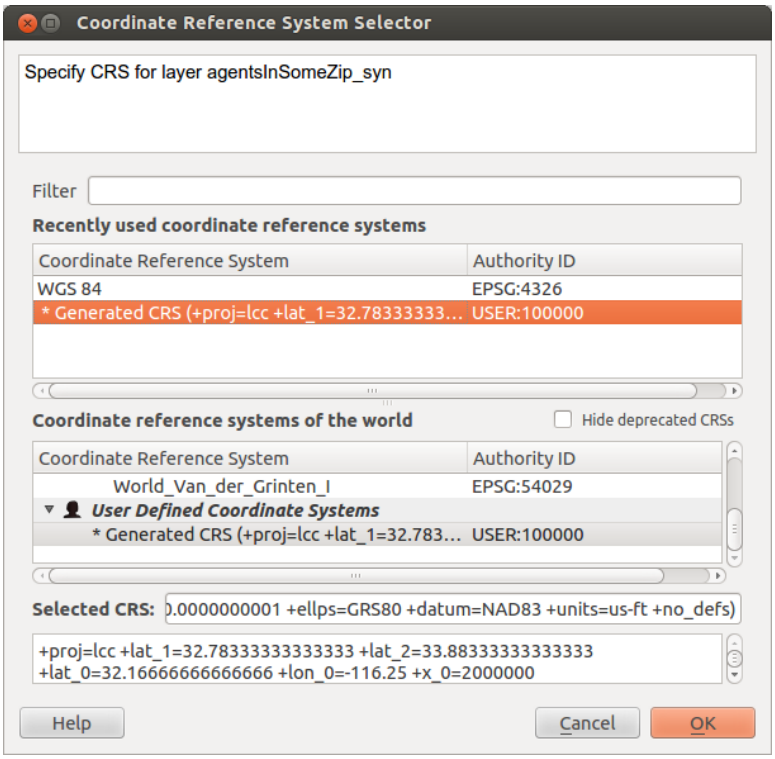

Figure 29: Coordinate Reference System Selector

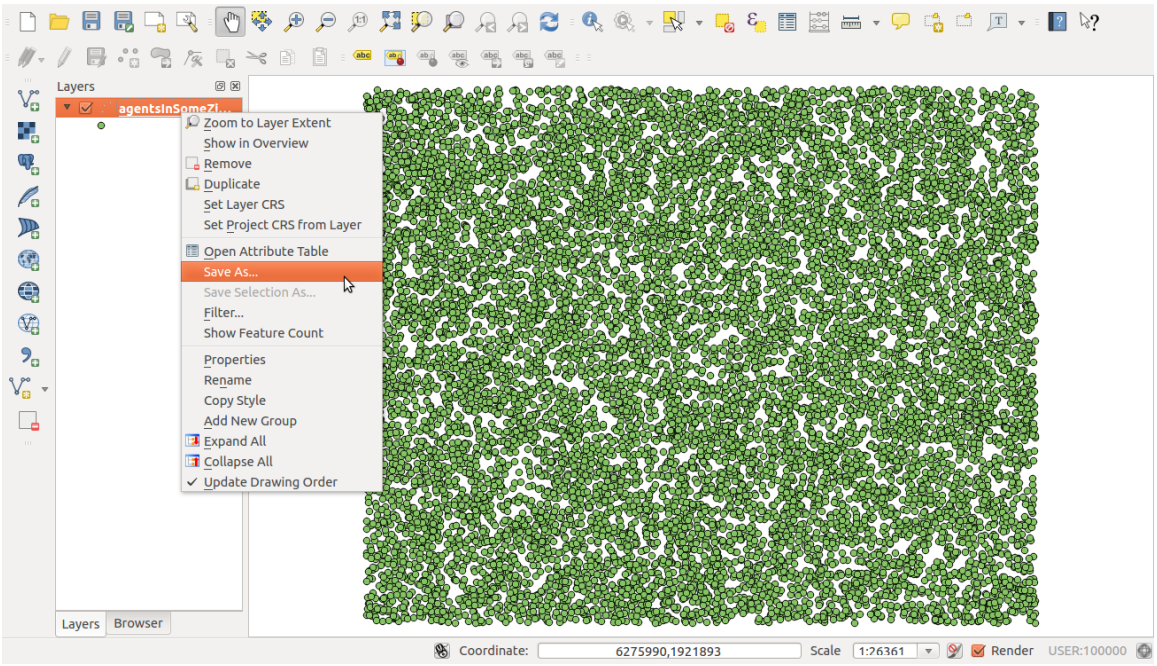

Figure 30: Save Layer

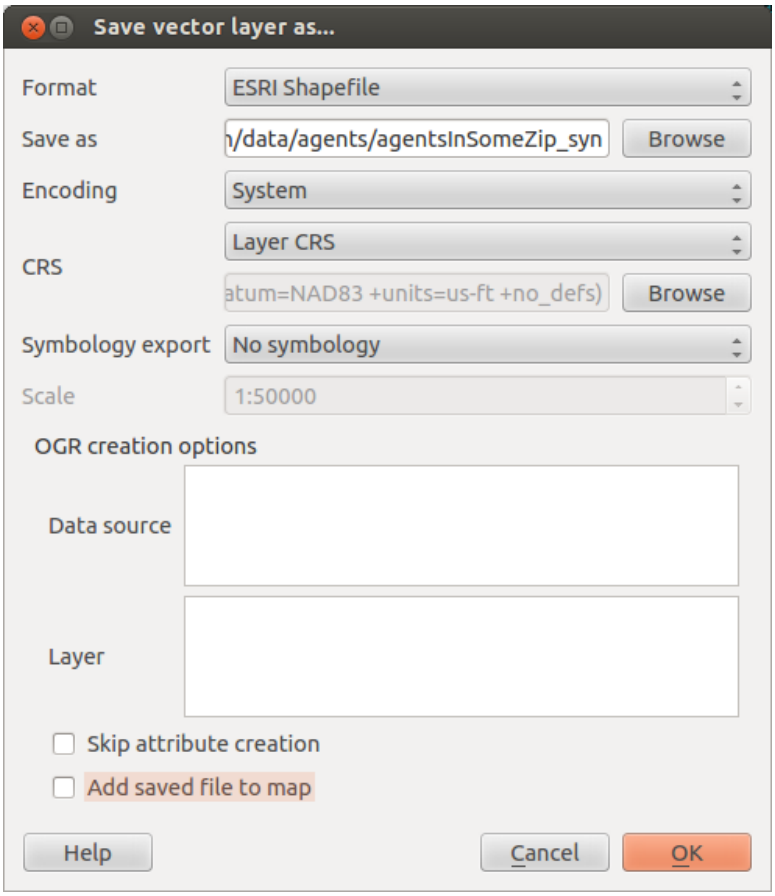

Figure 31: Save Layer As

Plugging in Alternative Models

Developers can plug in alternative models by either overriding a current method or create brand-new own methods. Although, we have already added comments in source files, we summarize these models and their corresponding method names in Table 1. Note that all these methods are defined in *NonAdopter.java*.

Final Remarks

Please feel free to leave comments, or contact authors if you have any questions. Enjoy the beauty of Agent-Based Modeling!!!

Table 1: Integrated Models and Corresponding Function Names

Model	Function Name
system capacity model	estimateCSIrating
ownership cost model	estimateOwnCost
lease cost model	estimateLeaseCost
electricity utilization model	calEconBenifit
Ownership and lease adoption model	predict