Walkthrough modeling in l2-matlab

In this guide, we will create a model of emotion contagion step by step. The model is described in Bosse et al. (2009) and revolves around the concepts shown in Figure 1 below. A short description of the model is given here, for more details read the article mentioned before or Chapter 7 of the reader for Intregrative Modelling.

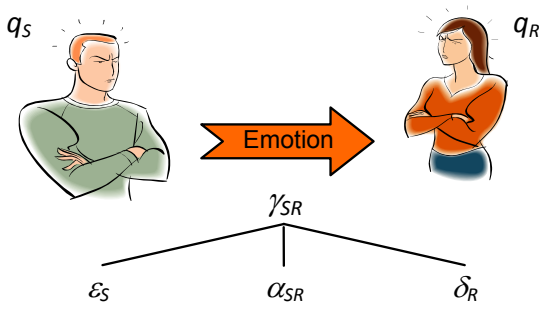


Figure 1. Aspects of emotion contagion

Each person has an emotion level, q_S and q_R in the figure above. The emotion level of the receiver *R* is influenced by the emotion of the sender *S* (and vice versa). The extent of this influence is dependent on a number of aspects; the expressiveness of the sender ε_S, the channel strength between sender and receiver α_{SR}, and the openness of the receiverδ_R. These three aspects together result in a contagion strength from the sender to receiver γ_{SR}. In this guide, the focus is on the following two rules only.

DDR1 *Determining contagion strengths*

```
has_expressiveness(B, E) &
has_channel_strength(B, A, C) &
has_openness(A, D)
↠ has_contagion_strength(B, A, E*C*D)
```

DDR2 *Updating emotion levels*

```
has_emotion_level(A, V1) &
has_emotion_level(B, V2) &
has_emotion_level(C, V3) &
has_contagion_strength(B, A, CS2)  &
has_contagion_strength(C, A, CS3)  &
A≠B & B≠C & C≠A &
step_size(DT)
↠ has_emotion_level(A, V1+CS2*(V2-V1)*DT+CS3*(V3-V1)*DT)
```

Setup working environment

For the first use, start by unzipping l2-matlab to a particular location, for example `C:\l2-matlab\`. To start using l2-matlab, this folder needs to be added to the Matlab path, by finding 'Set Path' (under file in older versions, or under home in new versions) and add the folder. If you save the path, this step has to be done only once.

Model folder

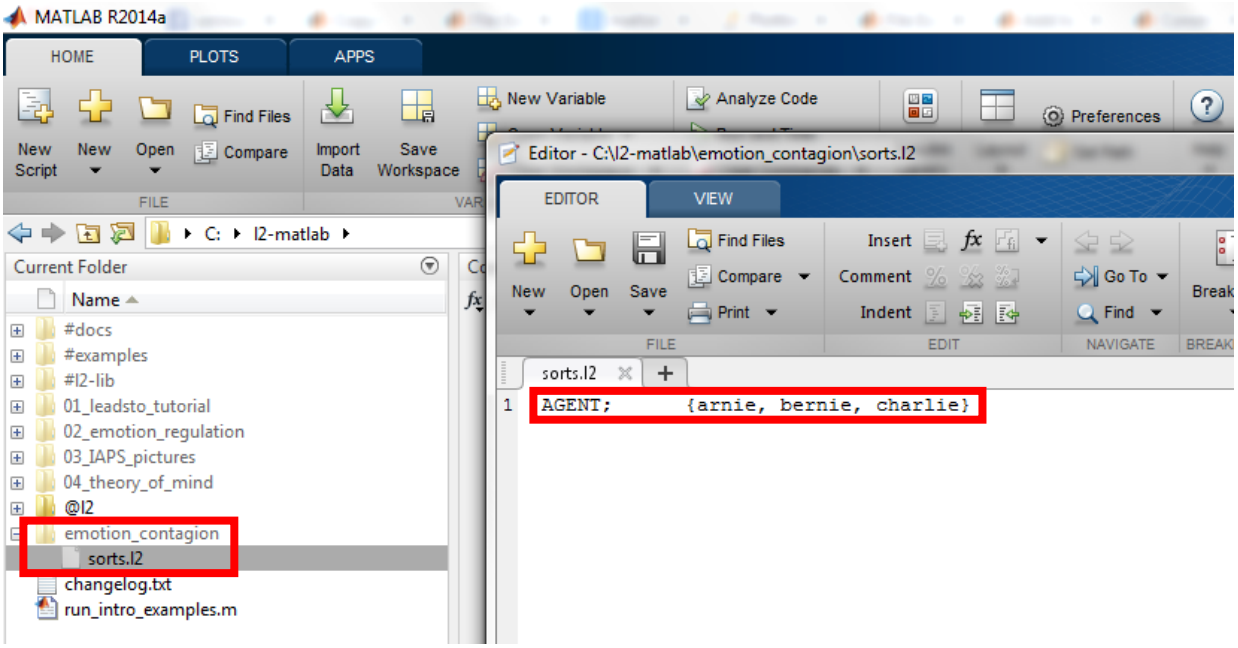
Each model is placed in its own folder. Create a new folder by right-clicking somewhere in the current folder window and select `New Folder`. Name this folder `emotion_contagion`.

Sorts.l2 file

This model makes use of two sorts, real numbers for all the numeric values and a sort AGENT for naming the particular agents. REAL is a sort that is available by default, however the sort AGENT needs to be implemented. Therefore, take the following steps:

1. Right click on the folder `emotion_contagion`.
2. Go to new file, and click script.
3. Rename the file to sorts.l2 (it is important that you rename the extension as well, as we in fact do not need a script but an l2 file).
4. Double click on the newly created file, such that it opens in the editor.
5. Add the following text to this file and save it.

```
AGENT;      {arnie, bernie, charlie}
```

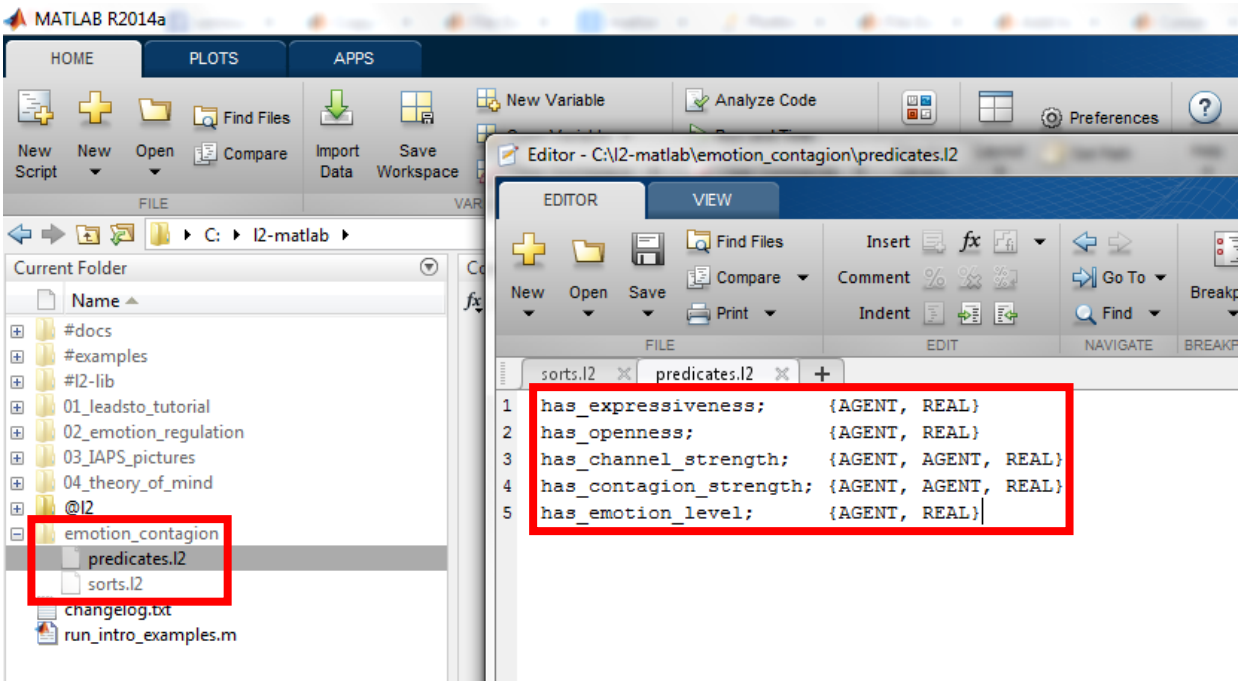


Predicates.l2 file

In a similar manner, create a file named `predicates.l2` and open it in the editor. Add the following text and save the file.

```
has_expressiveness;      {AGENT, REAL}
has_openness;            {AGENT, REAL}
has_channel_strength;    {AGENT, AGENT, REAL}
has_contagion_strength;  {AGENT, AGENT, REAL}
```

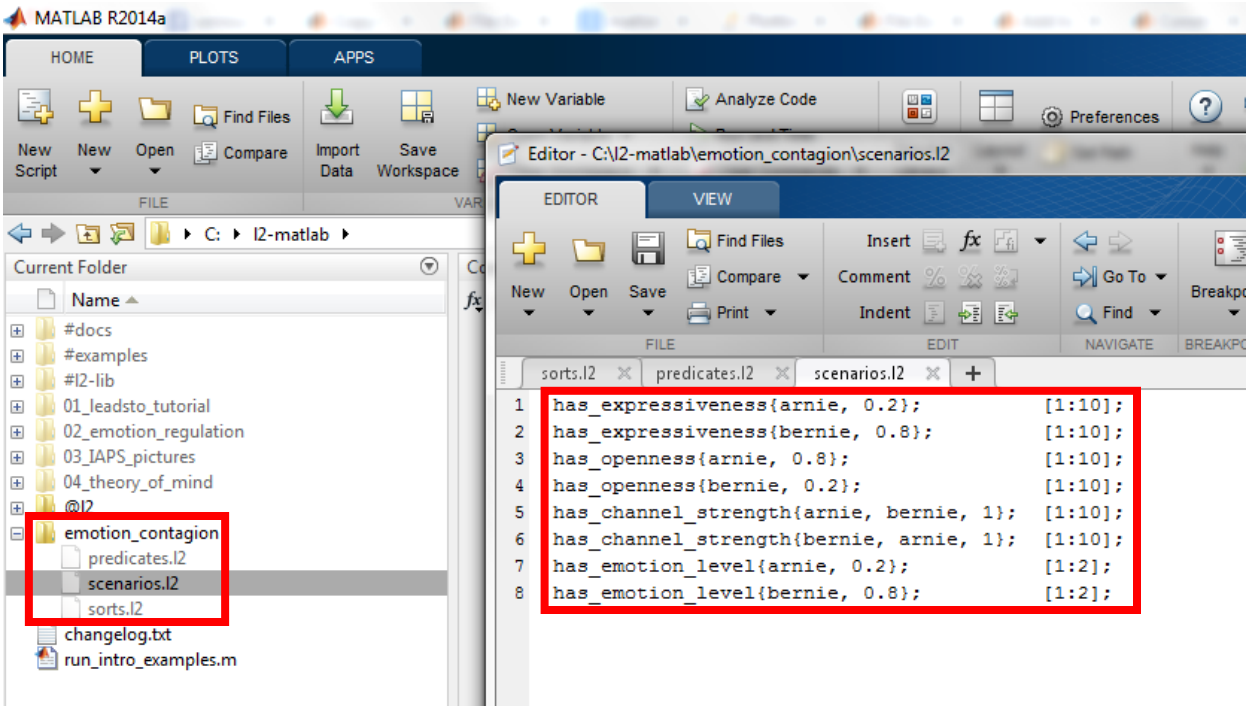
```
has_emotion_level;      {AGENT, REAL}
```



Scenarios.l2 file

To define an initial scenario for our simulations, a third file needs to be created named `scenarios.l2`. Do this now, add the following text using the Matlab editor and save the file.

```
has_expressiveness{arnie, 0.2};           [1:50];
has_expressiveness{bernie, 0.8};          [1:50];
has_openness{arnie, 0.8};                 [1:50];
has_openness{bernie, 0.2};                [1:50];
has_channel_strength{arnie, bernie, 1};   [1:50];
has_channel_strength{bernie, arnie, 1};   [1:50];
has_emotion_level{arnie, 0.2};            [1];
has_emotion_level{bernie, 0.8};           [1];
```



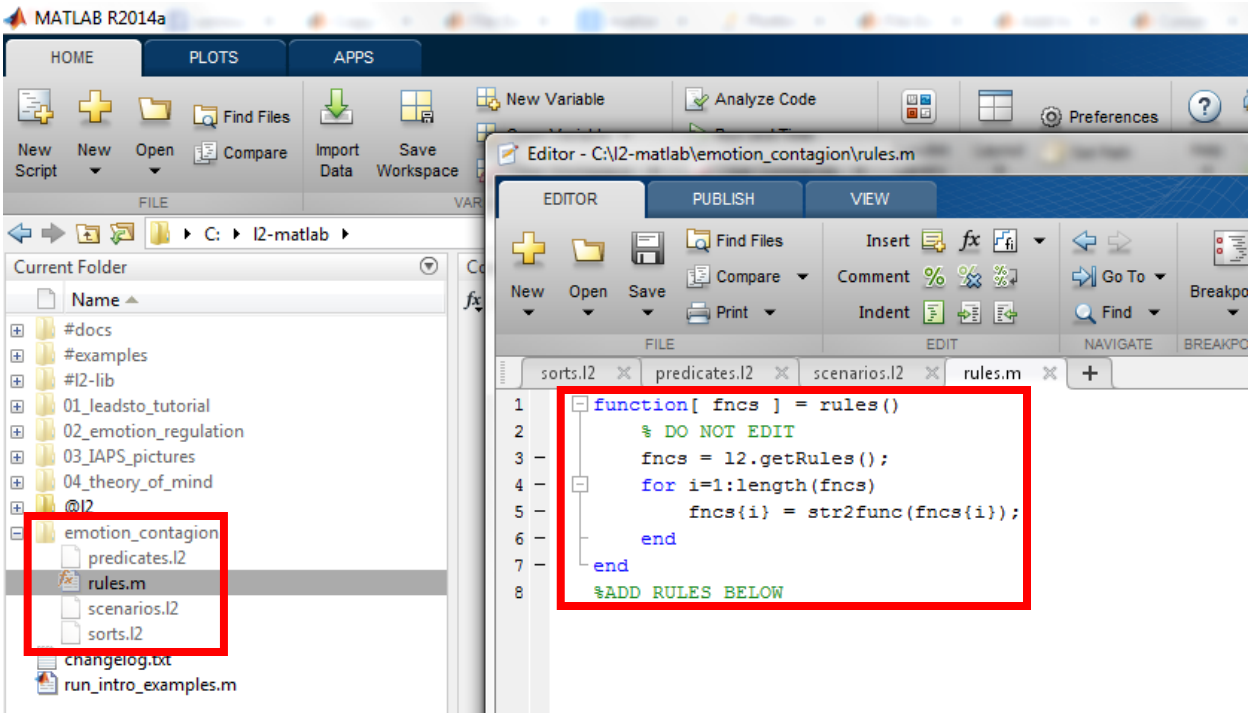
Recap. We have created three files. In sorts.l2 we defined a sort AGENT, which included three different agents as its elements. In predicates.l2, predicates were defined that are used in the model (e.g., `has_openness`). The possible values used in these predicates are of the sort AGENT defined before or the predefined sort REAL. In scenarios.l2 we defined the initial values for a simulation, thus the expressiveness, openness and channel strength for the entire simulation (from timestep 1 to 10) as well as the initial emotion value for both Arnie and Bernie (i.e., at timestep 1).

Rules.m file

To make the model dynamic, we need to create a set of rules defining the dynamics of the model. In this case, there are two rules; one for calculating the contagion strength, and a second for updating the emotion levels. These rules are defined in a Matlab script file named rules.m and can be created just like the other files. When you have created the file, open it in the editor and add the following lines. [1]

```
function [ fncs ] = rules()
    % DO NOT EDIT
    fncs = l2.getRules();
    for i=1:length(fncs)
        fncs{i} = str2func(fncs{i});
    end
end
%ADD RULES BELOW
```

[1] As of Matlab 2013b line 3 to 6 can be replaced by `fncs = localfunctions();`, a more robust native function.



Make sure you always add this function first in each of your rules.m files exactly as written here. This function is required to make l2-matlab work and to make it able to execute any of the rules you add to this file. We can now start by adding the two rules mentioned above (implemented as functions) in this file. First add the following below `%ADD RULES BELOW.`

```
function result = ddr1( trace, params, t )

end
```

This is the default outline for any rule you would like to write. In this case the rule (or function in Matlab) is named `ddr1`. Make sure each of your rules has a unique name. Afterwards, three variables are passed to each of your rules; the current simulation *trace*, up to the current time point *t,* as well as all defined parameters *params* (not yet used in this example).

Let us now start creating the rule to calculate the contagion strengths step by step. The first thing we need to do is make sure we calculate the contagion strength for each possible combination of two agents. As a first step, we go through each value for has_expressiveness, such that at least we find each agent that expresses some emotion on a particular time point. Add the following lines in between the two lines of the function.

```
%go through each expressiveness
for has_expressiveness = trace(t).has_expressiveness
    agent = has_expressiveness.arg{1}; %agent name
    expressiveness = has_expressiveness.arg{2}; %the expressiveness
```

The first line tells matlab to go through each value of `has_expressiveness` at time point *t* and return those values one by one as *has_expressiveness*. As this predicate contains a combination of an

AGENT and a REAL, the property arg is a cell array, with the name of the agent on the first index and the expressiveness on the second index. These values are retrieved in the next two lines.

Now, if we look at each channel strength from this agent to any other agent, we can find all the agents this agent influences. Therefore, in the next part of this rule, another for loop is created going through each channel strength originating from the current agent. Add the following lines directly after the previously added lines (still before the `end` of the function).

```
%go through each channel strength to find influenced agents
for has_channel_strength = l2.getall(trace, t, 'has_channel_strength',
                                                    {agent, NaN, NaN})
    agent2 = has_channel_strength.arg{2}; %agent name of the other agent
    channel_strength = has_channel_strength.arg{3}; %channel strength
```

As you can see, a similar for loop is used, where we go through a number of predicates. Therefrom, we can retrieve the other agent's name, which is on the second index as well as the channel strength from the third index. However, to get the relevant predicates, a function `l2.getall` is used. We cannot simply get all the channel strengths from the *trace* at time point *t* as done with the expressiveness, as we are only interested in those channel strengths that originate from our current *agent*. Although it is possible to check manually whether the channel strength is relevant for the current *agent*, by using this function this can be done in one step. By providing the *trace*, the relevant time point which in this case is the current time point *t*, the requested predicate `has_channel_strength` and a pattern that needs to be satisfied (in this case `{agent, NaN, NaN}`), we get exactly those predicates that we are interested in. The pattern consists of a cell array containing a number of fields equal to the number of values the predicate has, in this case three. For each field, you can specify a particular required value, as for the first field which should be *agent*. If the value may be anything, a *NaN* should be placed in that field, just as has been done for the second and third values.

Thus far, we have one agent, his expressiveness and a channel strength to another agent. To calculate the contagion strength, we also need to know the openness of the receiving agent. This can again be done by simply using the `l2.getall` function. Add the lines below after the previously added lines.

```
%get the openness of that agent
has_openness = l2.getall(trace, t, 'has_openness', {agent2, NaN});
 openness = has_openness.arg{2}; %the openness
```

Thus far, each of the leadsto lines in ddr1 has been replaced by a for loop. We could do the same here, however we do not necessarily need it here. Based on how the model is constructed and implemented, we are certain that there is exactly one predicate matching this pattern. Thus, we can ask for that predicate and retrieve the openness from the second index.

At this point, we have all the information required to calculate the contagion strength. Add the following lines again after the previously added lines.

```
%calculate the contagion strength
contagion_strength = expressiveness * channel_strength * openness;
```

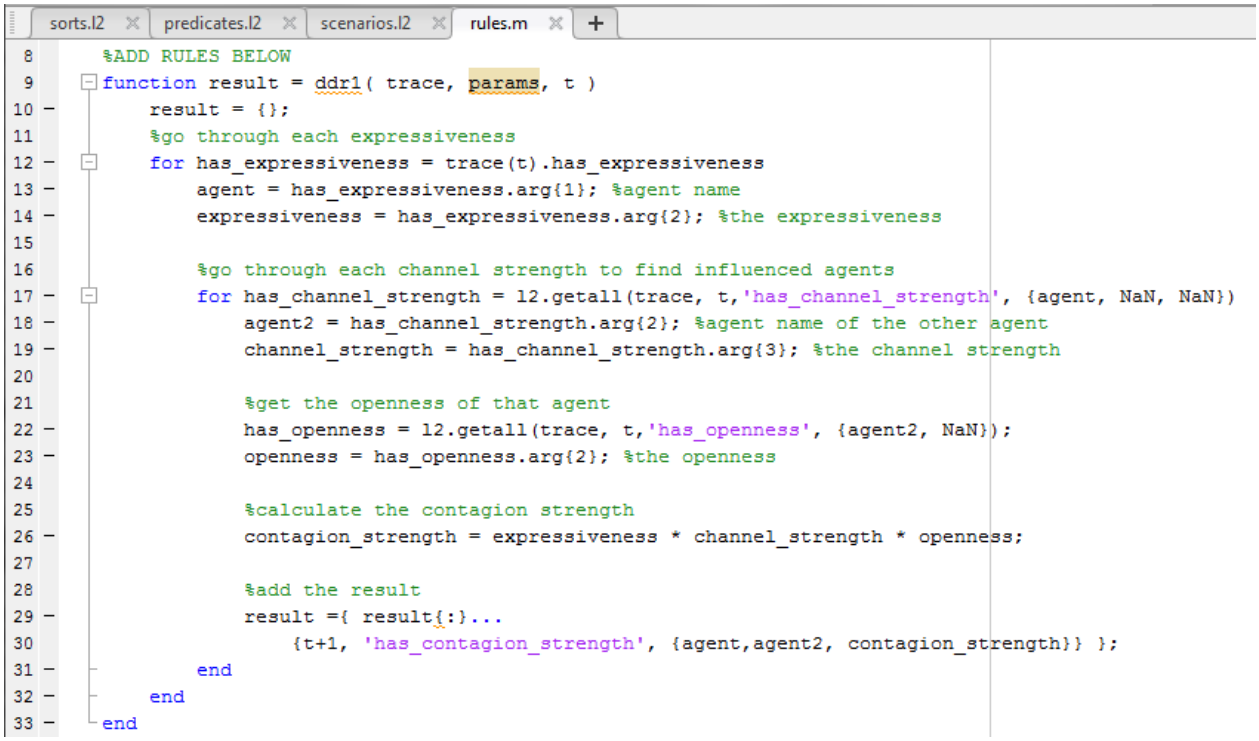
In this function, many results are calculated for different combinations of agents. Therefore, to return all those results, a cell array of results needs to be formed and each result is added to this array. Directly

after calculating the contagion strength, add the following lines to add the result to the (for now not yet existing) result array and close both for loops.[2]

```
%add the result
result = { result{:} ...
 {t+1, 'has_contagion_strength', {agent, agent2, contagion_strength}} };
end
end
```

The last step to finalize this function is to create an empty result array at the beginning of the function. Add the following line directly after the line containing `function result`, and save your file.

```
result = {};
```

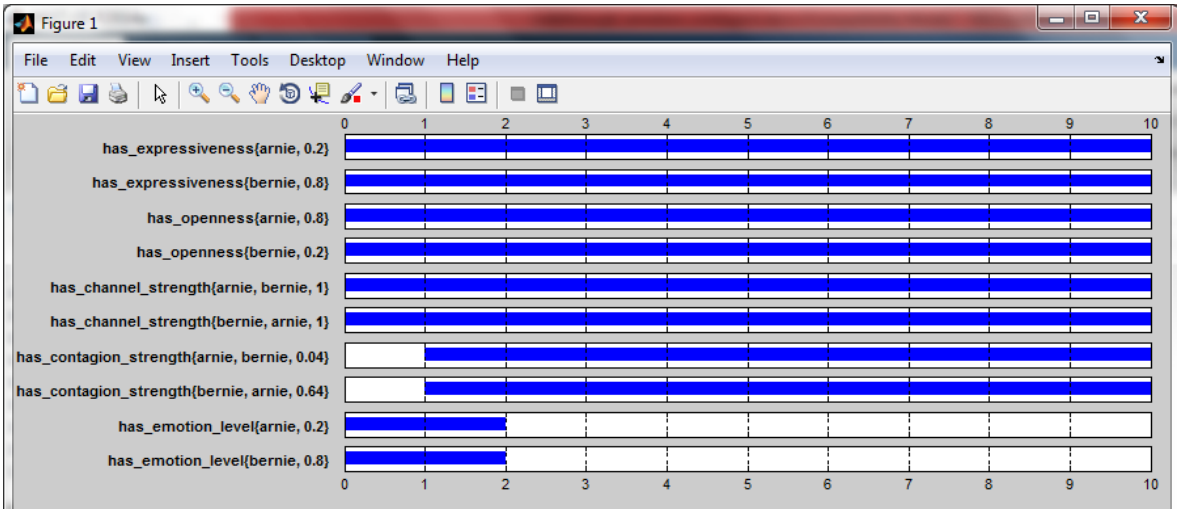


At this point, the first function is finished. You should be able to load, run and plot the first results. Try this by executing the following lines of code (this can be done either by copying the lines directly into the 'command window' or by creating a new script file that includes the lines, and pressing the Run button).

```
clear all
close all
model = l2('emotion_contagion')
model.simulate(10)
model.plot()
```

[2] You'll see three dots in the code below. This is only there to tell Matlab the code continues on the next line as if it was on the same line. You can remove those dots and put both lines on the same line in Matlab.

The first two lines are optional and clear all variables and close all windows. The third line creates an l2 model of the files in the folder `emotion_contagion`. The fourth line runs a simulation for 10 timesteps and the last line plots the result.



Exercise: ddr2

To complete the model, a second rule is required which updates the emotion levels of the agents. For reference, a simplified version for 2 agents of ddr2 is added below. This rule can be added in the rules.m file below the first rule. Hints for two different approaches to implement this rule are given below.

DDR2 *Updating emotion levels*

```
has_emotion_level(A, V1) &
has_emotion_level(B, V2) &
has_contagion_strength(B, A, CS2)  &
A≠B
↠  has_emotion_level(A, V1+CS*(V2-V1))
```

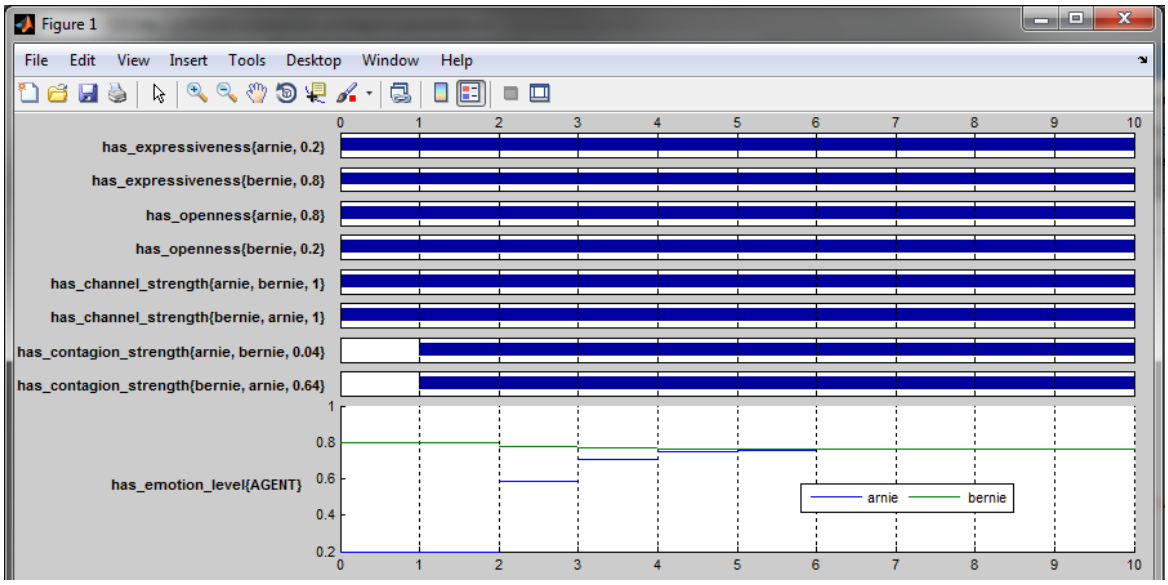
Option a. The first option stays very close to the rule as given and is the more simple approach. Using a for loop for the first three lines, you should be able to implement the rule in l2-matlab. You also need to change some of the intervals in your scenario. For calculating the new emotion use something like below, although variable names can differ of course.

```
new_emotion_level = emotion_level + ...
      contagion_strength * (emotion_level2 - emotion_level);
```

Option b. The second option makes a few changes to the way the rule is implemented, with the benefit of being more generic, which will save you time and effort later on. However, this approach significantly differs from the way the leadsto rule above is written and goes through the predicates in a different order. You'll need two for loops, use the `l2.getall` function twice and incrementally calculates the change of emotion by combining the influence of the other agents one by one (in this case it is only one, but keep the rule general in case of more agents). For calculating the emotion change use something like below, although variable names can differ of course. Carefully consider at which point in the function you add the new emotion level to the result array in order not to add multiple emotion levels for the same agent.

```
emotion_change = emotion_change + ...
      contagion_strength * (emotion_level2 - emotion_level);
```

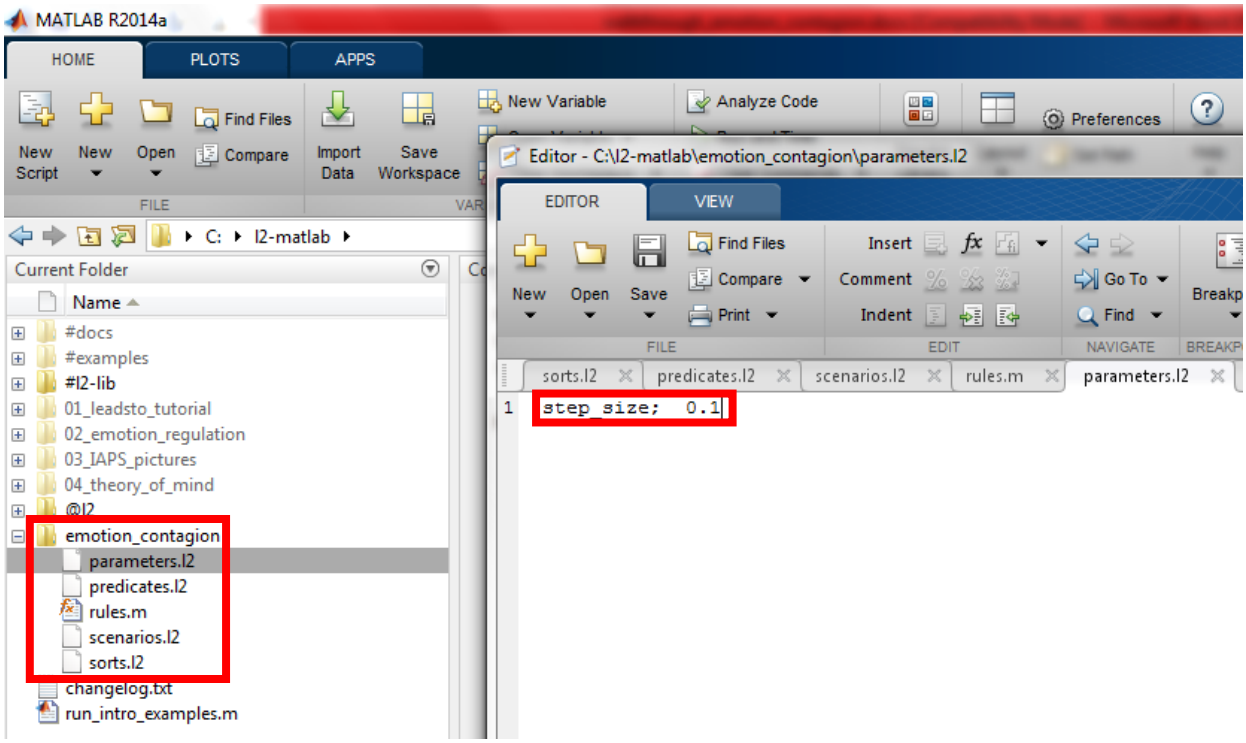
If you finished writing the function, you can simulate and plot the model again. The resulting plot should look as follows.



Parameters.l2 file

If you read the model description carefully, you'll find that the updating of the emotion level is not done correctly yet. For updating the emotion level, a step size Δt is used, such that the emotion levels not run outside the [0,1] interval. We can add this step size as a parameter to the model. Create a file named parameters.l2 and add the following line of code, after which you save the file.

```
step_size;  0.1
```



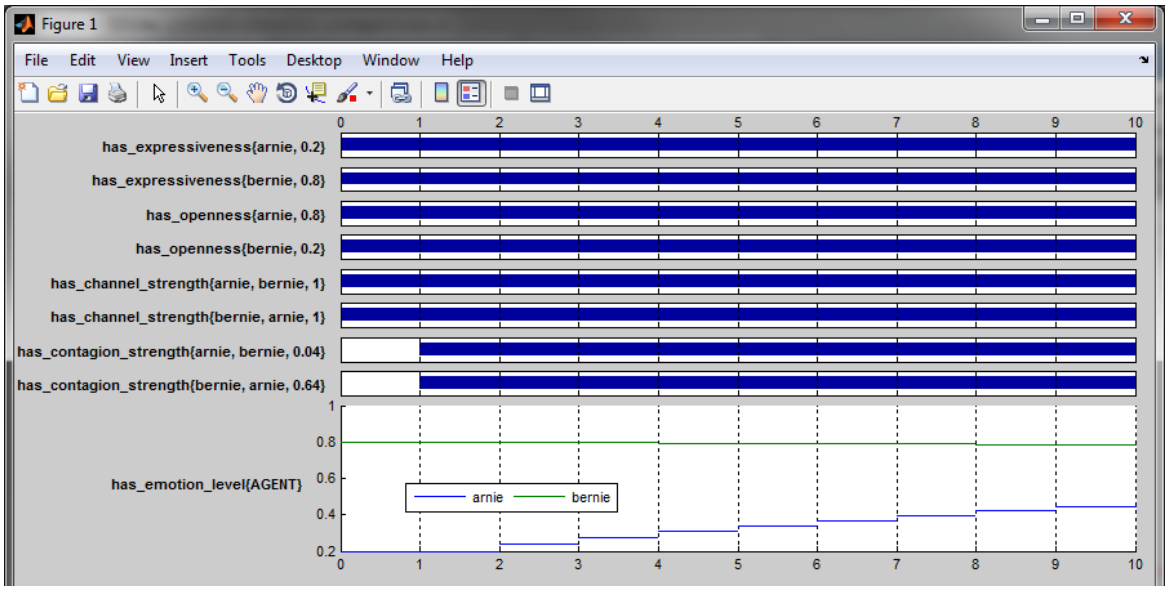
In the function ddr2 update the line where the emotion level gets changed, such that it uses the step size parameter. We can access this parameter from *params* using `params.step_size`. Update the line changing the emotion level as follows (first part of the line depends on the approach chosen earlier).

```
[option a or option b] + ...
  contagion_strength * (emotion_level2 - emotion_level) * params.step_size;
```

As some changes have been made to the l2 files, we need to make sure these files are reread. This can be done by either loading the model again using the same commands as before or issuing the reset command on the current model as follows.

```
model.reset();
```

If you simulate the model again and plot the results, you should see the emotion levels change more slowly than before.



Exercise: increase simulation duration

As the emotion levels change more slowly using the time step parameter, we would like to increase the duration of the simulation to 50 timesteps. For this, you not only need to change the simulate command, but also a few things in one of your l2 files. Try this out by yourself.

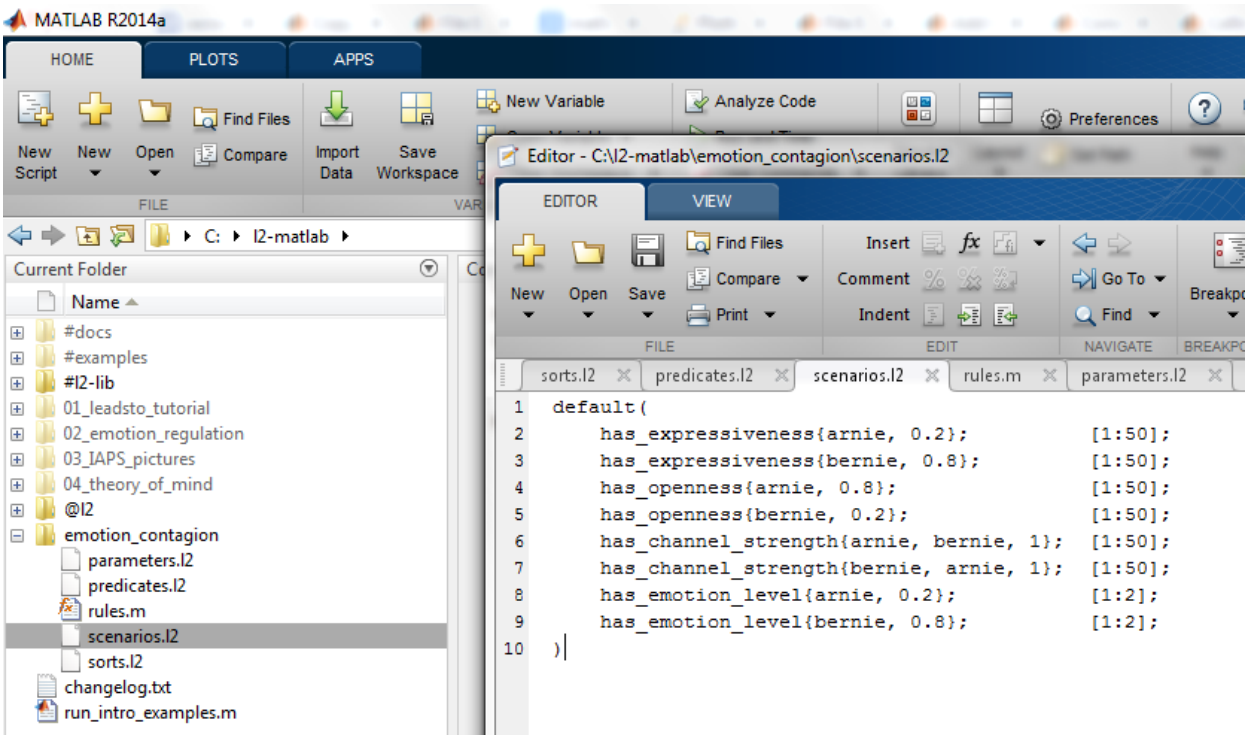
A second scenario

If you think back of the sort AGENT, we added three different ones; Arnie, Bernie and Charlie. The current scenario only uses two agents. Now we are going to add a second scenario, using all three agents. Go back to the scenarios.l2 file. Add the following line to the top of the file and the second line to the bottom, such that it encloses the current scenario.

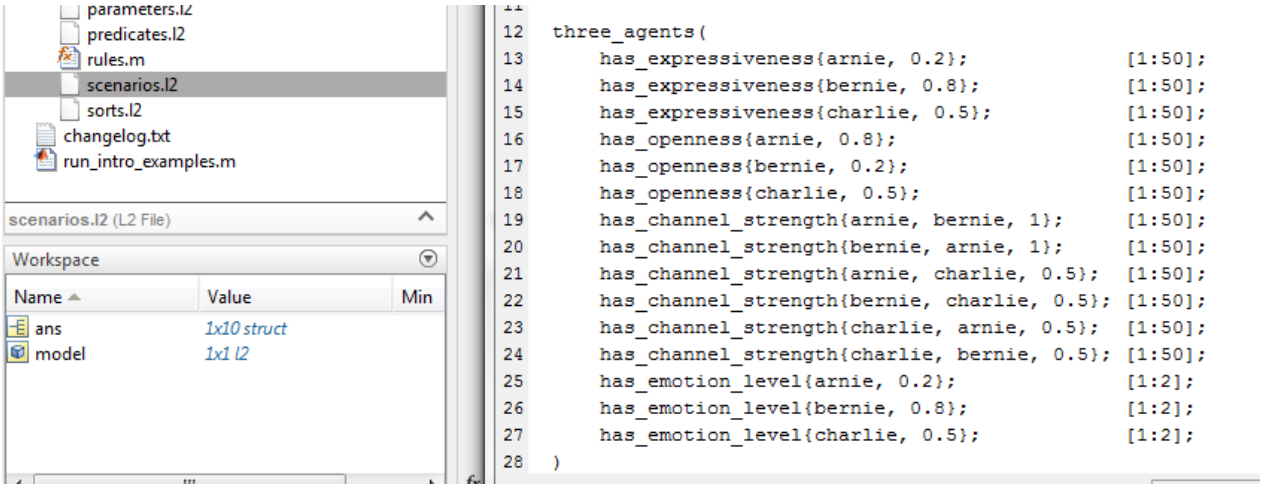
```
default (

)
```

The current scenario will now be used by l2-matlab as the default scenario. If you create multiple scenarios, it is useful to always create a default scenario.[3]

[3] You might notice the added semi-colon add the end of each line in the screenshot. Although not required, it may be added if desired.



Now, select everything and copy-paste it below the current scenario. Rename that second scenario from default to three_agents and add an expressiveness, openness, channel strengths(!) and emotion level of 0.5 for Charlie. The resulting scenario should look as follows.



If you previously chose to implement ddr2 using the approach described in option a, you need to rewrite that rule such that it works for three agents. Update the rule such that it implements ddr2 as given in the leadsto notation below. Note that after updating the rule, your model won't work anymore for the default scenario (with two agents). Alternatively, you could try implementing option b, which will work for any number of agents. If you have implemented option b, there should also be a small difference in your scenario compared with the screenshots above.

DDR2 *Updating emotion levels*

```
has_emotion_level(A, V1) &
has_emotion_level(B, V2) &
has_emotion_level(C, V3) &
has_contagion_strength(B, A, CS2)  &
has_contagion_strength(C, A, CS3)  &
A≠B & B≠C & C≠A &
step_size(DT)
↠  has_emotion_level(A, V1+CS2*(V2-V1)*DT+CS3*(V3-V1)*DT)
```

Remember, after any of the l2 files have changed, you first need to reset the model. Afterwards, we can simulate and plot the model using this second scenario as follows.

```
model.reset();
model.simulate(50, 'three_agents');
model.plot();
```

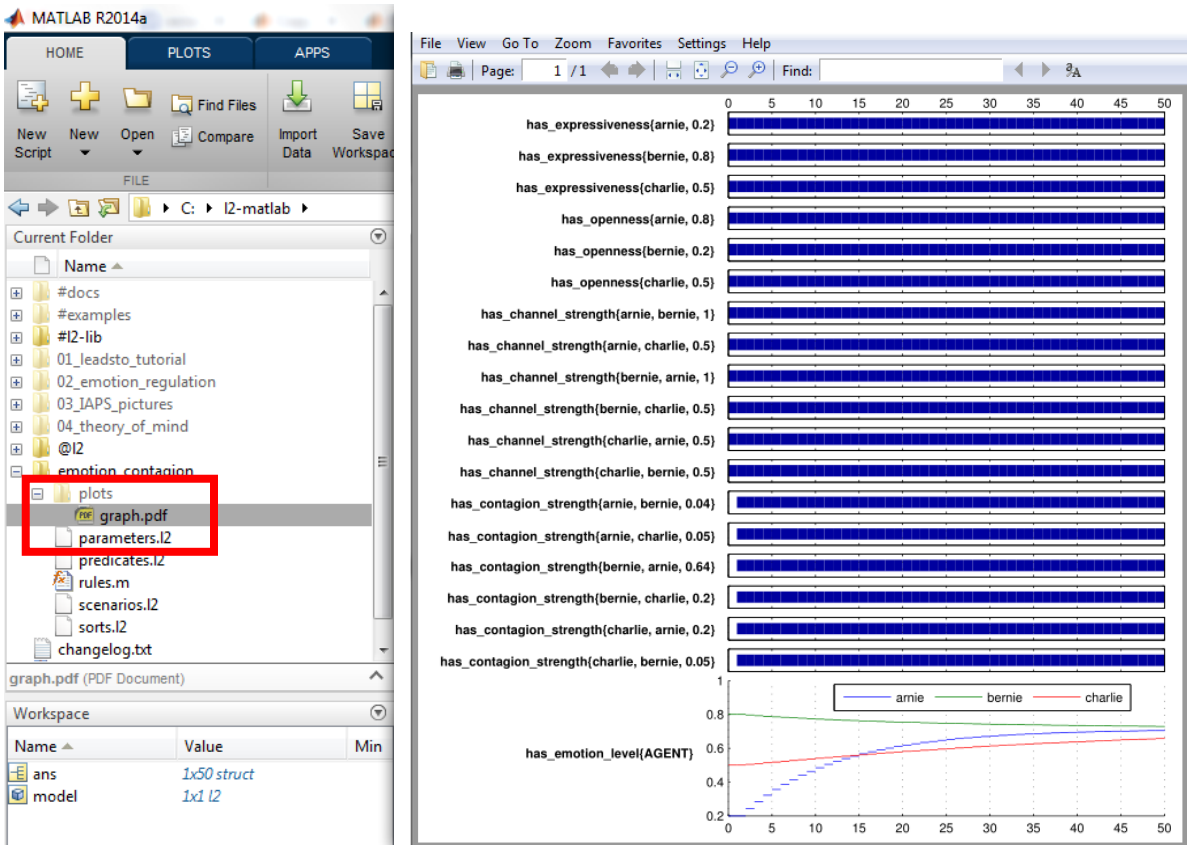
Notice, that within the simulate command, we added the name of the scenario we just created as a second input. Thus, `model.simulate(50, 'default')` is equal to `model.simulate(50)`. In a similar manner, you can create different sets of parameters in the corresponding l2 file. To run a simulation with a different sets of parameters, add the name of that set as a third input. In that case, you are always required to name the scenario you wish to use, even if it is the default scenario.

Plot to file

If you start creating more elaborate models, you may notice that graphs start to disappear at the bottom of your screen, thereby making part of plot unavailable. Other times, you might like to include your graph in some other document and would like to save it as an image. In such cases, it is possible to save the plot to a file. Try this by executing the following command.

```
model.export('graph.pdf');
```

When Matlab finishes plotting and saving the file, you should see a 'plots' folder appear in your model folder. In that folder, you can find the graph.pdf file, containing the entire plot of your results. Note, different file types are supported, producing different qualities of the resulting file.



Selective plotting

When plotting your results, it might be that you are only interested in some of the predicates. For example, we could leave out the expressiveness, openness and channel strengths from the previous plots, leaving only the contagion strength and emotion levels to be shown. This can be done by passing along a cell array containing the names of those predicates you like to be visible to either the .plot() or the .plotFile() function as the last argument. This would look as follows, which you are encouraged to try out on any of the scenarios.

```
model.plot({'has_contagion_strength' 'has_emotion_level'});
model.export('filename', ...
   {'has_contagion_strength' 'has_emotion_level'});
```

Conclusion

By now, you have seen most of l2-matlab's functionality. However, much of what you can do using l2-matlab hinges on your Matlab knowledge. As you become more and more experienced using Matlab, you will be able to write even more complex rules for your models or create your own graphs for your simulation results. Included in the l2-matlab distributable are a number of examples showing a number of examples of varying difficulties. These four examples are explained in more detailed in the l2-matlab manual bundled with the software.

References

Bosse, T., Duell, R., Memon, Z.A., Treur, J., and Wal, C.N. van der (2009), A Multi-Agent Model for Mutual Absorption of Emotions. In: Otamendi, J., Bargiela, A., Montes, J.L., Pedrera, L.M.D. (eds.), Proceedings of the 23th European Conference on Modelling and Simulation, ECMS'09. European Council on Modeling and Simulation, 2009, pp. 212-218.