Appendix A

L2-matlab

L2-Matlab is a software package for Matlab to easily implement discrete-time models. In essence, L2-Matlab is nothing more than a couple of functions for Matlab which uses a number of text-based files to describe your model and a set of Matlab functions to make your model dynamic. Appendix A.1 contains the manual that is distributed with the software and Appendix A.2 describes four examples that are used to introduce various possibilities of l2-matlab.

A.1 Manual

A.1.1 Installing L2

You can download the latest version of l2-matlab from www.few.vu.nl/~jmn300/l2-matlab/. There are three different versions available, choose the one that fits your needs. When you've downloaded the zip file, unzip it to some folder on your computer.

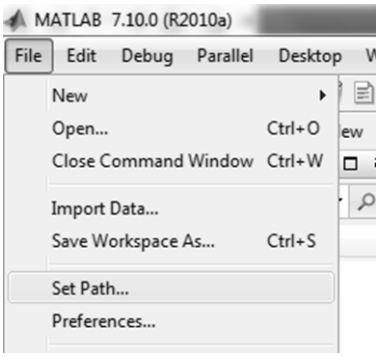
In Matlab, you need to make sure that it knows where to find the `#l2-matlab folder`. This needs to be done by adding it to the `Matlab Path`. Depending on your version, there are multiple ways this can be done. In older versions, you can probably find a `Set path` under the File menu. More recent versions contain a button `set path` in the `home` bar under `environment`. Afterwards, you need to add the `#l2-matlab` folder and save it for future use. If this has been done correctly and you type `help l2` in the command window, a short explanation should appear.

A.1.2 Terminology

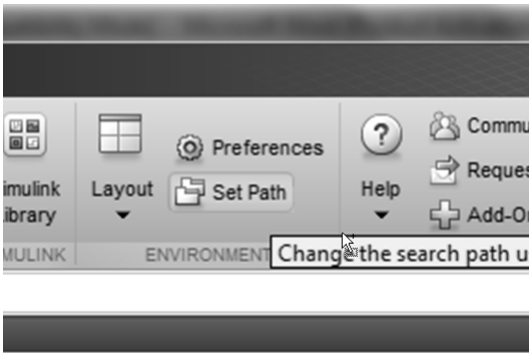
For creating models using l2-matlab, multiple terms are used to indicate something specific. To avoid any confusion, a list of such terms is given below, including an explanation of what is meant by these terms in this document.

A.1.3 Defining models in L2

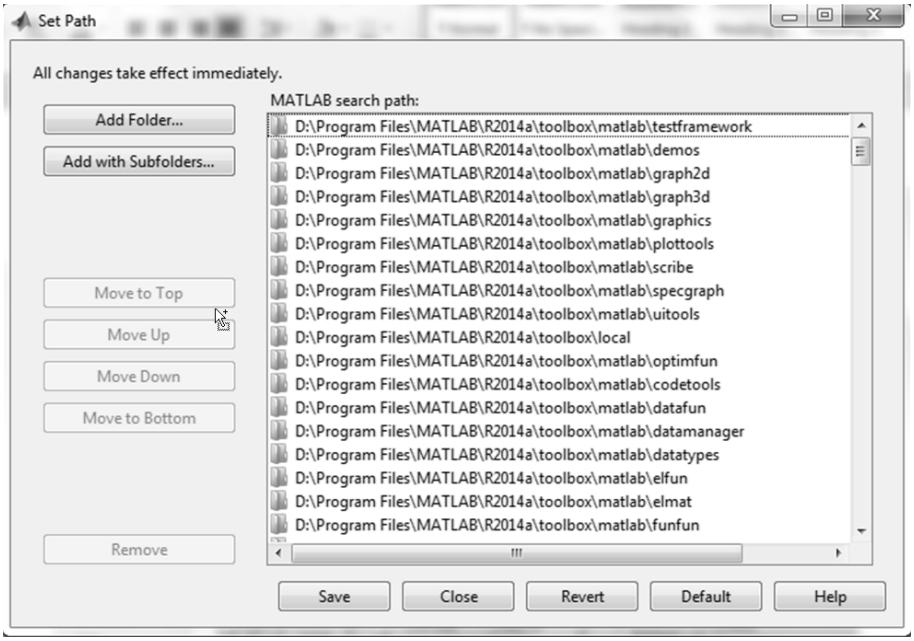
Models in Matlab are defined using a number of `.l2` files containing the static description of the model and one `.m` file for implementing the dynamics. All these files are contained in a single folder, here after called the `model folder`,



(a) Older versions



(b) Newer versions



(c) Set path

Figure A.1: Setting the Matlab Path

`l2-matlab`	The package of Matlab functions used for creating and simulation models.
`model folder`	The folder containing all files for a particular model.
`atom`	A specific value.
`sort`	A set of atoms and/or predicates
`predicate`	A predicate has a name and one or more arguments. Each argument refers to a sort that specifies the possible instantiations for that argument. The term is used for both predicates that are and are not instantiated with specific values.
`scenario`	The initial situation for a simulation.
`rule`	A Matlab function specifying some particular dynamics of the model.
`parameter`	A particular value which is constant throughout a simulation.
`trace`	A description of simulation results.

Table A.1: L2-matlab terminology

by which the model is identified. In this chapter, each of these files are explained, both their structure and function.

sorts.l2

Sorts define particular sets of instances which can be used in your model. In this file, both the possible sorts as well as their values are defined. Each line starts with the name of the sort, followed by a semi-colon and the possible values, separated by commas, enclosed in curly brackets (see below). Furthermore, a predicate can be used as a possible value for a sort, thus creating a possible nested structure for predicates. To indicate a predicate reference, angle brackets are used.

There are two sorts which do not need to be defined and can be used by default. These are `REAL` and `BOOLEAN`. `REAL` contains all real values while `BOOLEAN` contains true and false as possible values.

sorts.l2

```
sort;      {atom, <predicate>}
sort;      {atom, atom, atom}
..
```

`sort`	Name of the sort
`atom`	Possible value for the sort
`<predicate>`	A predicate that can be nested as a value for `sort`

predicates.l2

A predicate defines a named relation over one or more sorts and are implemented in a similar syntax as used for sorts. Predicates are the main building blocks for your model for which you will define dynamical relations to simulate a particular scenario. Below is the generic structure of a predicates file shown.

predicates.l2

```
predicate;      sort
predicate;      {sort, sort}
..
```

`predicate`	Name of the predicate
`sort`	Sort used in the predicate, as defined in sorts.l2

scenarios.l2

A scenario defines the initial situation when you start simulating the model. Therefore, it is possible to define multiple scenarios in a single file. Each scenario is given a unique name, followed by a round (opening) bracket. It is good practice to always define a scenario named default first. If during a simulation no specific scenario is given, this default scenario will be used. On the subsequent lines, predicates and the time points for which these predicates holds are given. Here, a predicate is always completely instantiated, that is for each of its arguments a specific value is given. Time points are given in a Matlab array notation and can make use of the various ways such arrays can be defined in Matlab (see for examples below). At the end of a scenario, a closing round bracket is added, after which any number of other scenarios can be defined. If only a single scenario needs to be defined, the scenario name and round brackets can be left out and each line will be read as part of the default scenario.

scenarios.l2

```
scenario (
   predicate; time
   predicate; time
   ...
)

scenario (
   ...
)
...
```

`scenario`	Name of the scenario
`predicate`	A predicate with values specified
`time`	Timepoints for which the previous atom will hold

- `x` a single time point x
- `[x:y]` from timepoint x until y
- `[x y ..]` for timepoints x, y, etcetera

parameters.l2

Parameters are values that remain constant during a simulation. They are defined similar to scenarios, such that various different sets of parameters can be defined.

scenarios.l2

```
set (
   parameter; value
   parameter; {value, value}
   ...
)

set (
   ...
)
...
```

`set`	Name of the set of parameters
`parameter`	Name of the parameter
`value`	The value for the parameter

rules.m

Rules specify the dynamics of the model and are written in Matlab functions. In every rules file, a function as shown below needs to be added at the top. Below that, the custom rules for the model dynamics are added. As this is a major part of the model, more on this will be explained later on.

rules.m

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

%ADD RULES BELOW
function result = rule( trace, params, t )
    %some calculation of new value
    result = 'predicate', time, value;
end
...
```

`function [ fncs ] = rules() ... end`

Always add this function exactly as written here to your rules.m file.[1]

```
function result = rule( trace, params, t )
    %some calculation of new value
    result = 'predicate', time, value;
end
```

`rule`	Name of the rule
`predicate`	The predicate as defined in predicates.l2 you are setting a new value for
`time`	The time point this value is true for, usually t+1
`value`	The value

[1] As of Matlab 2013b this can be changed to `fncs = localfunctions();` and removing the for loop.

A.1.4 Writing rules

The rules are very important, as they define the dynamic part of a model. However, before we can start writing rules, an understanding is required of how predicates and the trace are represented in Matlab. Therefore, the next two sections will explain more on that, after which the global requirements of a rule are presented and some useful functions in the last two sections.

Predicate format

A predicate is implemented as a separate class in Matlab. A predicate contains a name as well as a list of arguments. There are various ways of creating a predicate in Matlab:

Creating predicates. The most basic way of creating a predicate is with no arguments. In that case, a predicate with no name and an empty argument list is returned.

```
>> p = predicate()

p =

  predicate with properties:

    name: ''
     arg: {}
```

A predicate can be created based on a string. This string contains both the name and the values for that predicate. Here, each of the values will be automatically transformed to the most suitable format; numbers will be stored as numeric values, `true` and `false` as boolean values and anything else as string values.

```
>> p = predicate('name{value1, 2}')

p =

  predicate with properties:

    name: 'name'
     arg: {'value1'  [2]}
```

In the last method, the name and values are given separately, whereby the values are passed on in a cell array with each particular value already in its required format. The name of the predicate is provided as the first argument in a string format.

```
>> p = predicate('name', {'value1', 2})

p =

  predicate with properties:

    name: 'name'
     arg: {'value1'  [2]}
```

Accessing predicates. The values of a predicate can be accessed via the name and arg properties. To retrieve the name of a predicate p, type p.name. Similarly, the arguments of a predicate can be accessed via `p.arg`. As these arguments are stored in a cell array, a particular argument can be retrieved using its index and curly brackets, i.e. `p.arg2`.

Operations on predicates. Predicates are implemented in such a way that simple Matlab operations can be performed on predicates that contain a single argument. In that case, values are cast to the required format and the resulting value is returned.

```
>> p = predicate('number{2}');
>> p+2

ans =

     4
```

Note, that due to how Matlab is implemented, some operations will provide you with an answer although it might be different than you might expect.

```
>> p = predicate('text{value1}');
>> p+2

ans =

   120    99   110   119   103    51
```

Currently the following Matlab operators have been implemented:

- +a
- a + b
- a / b
- a .* b
- a .^ b
- a <= b
- a ~= b
- -a
- a - b
- a \ b
- a ./ b
- a < b
- a >= b
- a & b
- ~a
- a * b
- a ^ b
- a .\ b
- a > b
- a == b
- a | b

Predicate functions. There are various functions available to work with predicates.

`pred2str`	Convert a predicate to its string representation.
`str2pred`	Convert a string to its predicate representation.
`ispredicate`	Test if a particular variable is a predicate
`predcmp`	Compare two predicates for equality, both can be in either string or predicate format.

Trace format

All simulation results are stored in the model's trace. This trace starts with only that which is defined in the scenario, but is enriched with simulation results at each time step. To access these values within the rules, it is important to understand its structure.

The trace is stored as a Matlab structure, with each time point in its own layer. Thus while trace contains all simulation results, `trace(1)` gives only the results for the first time point and `trace(x)` those for time point x. Furthermore, predicates are sorted based on their name. Thus, each predicate has its own field in the trace; `trace.name` gives you all predicates with name *name* for all time points, while `trace(1).name` gives you only those for time point 1 and `trace(1).name(1)` gives you only the first result for time point 1.

At each time point, predicates with a similar name are stored in an array of predicates. Thus `trace(1).name` returns a single predicate array. However, as different time points are stored in separate layers of the trace, retrieving results for multiple time points results in a so-called comma separated list where each time point is returned one after the other. It is possible to 'catch' all these results in a cell array by adding curly brackets around the targeted results; `trace.name` returns a single cell array with each cell containing the results for name for a particular time point.

Parameter format

Parameters are stored in a structure similar to the trace. However, as parameters stay constant throughout the simulation, no different time points exist. Thus, if the parameter file defines a parameter p, this value can be accessed via `parameters.p`.

Rule format

Each rule is implemented as a single function and follows a similar format. Take a look at the following (empty) rule:

```
function result = rule( trace, params, t )
    ...
end
```

As can be seen the rule returns a result *result*, is named *rule* and receives three variables. The results will be explained below. The name that is given to each rule can be chosen freely, as long as it does not violate any naming rules of Matlab and each rule is given a unique name. As for the three variables that are provided - *trace*, *parameters*, *t* - different names may be chosen as well. The parameters contain the parameters in a structure as explained above. The trace contains the simulation results up to and including the current time point *t*, again in the structure as explained above.

Using the information provided, any Matlab code can be used to derive results, which are returned by the function. A result can be given in three different formats, but always contain the time point(s) for which the result holds and the predicate itself. The variants are shown below, whereby the first is a cell array with the time point(s) for which the result holds in the first location and the predicate as a string in the second. Alternatively, the predicate can also be given in the predicate format. And lastly, a cell array can be provided with again the time points in the first place, the targeted predicate name in the second and a cell array of the values in the last place. For the time points, a single time point is sufficient, but alternatively an array of numbers may be used instead in order for the result to hold for multiple time points.

```
result = {t+1, 'name{value1, 2}'}
result = {t+1, predicate('name', {'value1', 2})}
result = {t+1, 'name', {'value1', 2}}
```

To return multiple results, a cell array of results should be created and returned. This can be done in various ways, for example in a for loop as shown below.

```
result = {};
for i=1
    result = { result{:} {t+1, predicate('name{value1, 2}')} }
end
```

L2 functions. As mentioned, any Matlab code can be used to derive results. However, two functions are available for convenient access to the trace. `l2.getall` returns all predicates that match a particular pattern, while `l2.exists` does the same but as a primary result returns a boolean value indicating if any values exist.

The pattern is a predicate, but any value can be substituted by a ~ to indicate any value is OK for that argument. In the predicate these blanks will be represented by a `NaN`, which can be seen as a null value for Matlab. For example, `predicate('namevalue1, ~')`represents a pattern whereby the predicate should be named name, the first argument should contain the value *value1*, but any value may be contained at the second argument. Besides such a pattern, a targeted trace and time point are required inputs for the functions.

```
predicates = l2.getall(trace, t, pattern)
```

This function returns all predicates for time point t in the trace trace that match the pattern pattern.

```
[tf, predicates] = l2.exists(trace, t, pattern)
```

This function returns primarily a truth value tf indicating if any predicates exists for time point t in the trace trace that match the pattern pattern. If required, the matching predicates are returned as well as a second output.

Similar to the results, the pattern can also be given in string format or as two arguments, one containing the targeted predicate name and the other a cell array with the values. Take notice that you do need to use the `NaN` value instead of the ~, for example:

```
predicates = l2.getall(trace, t, 'name', {'value1', NaN})
```

A.1.5 L2 models in Matlab

To open, simulate and work with l2-models in Matlab, the `l2` class is used. This class takes care of reading the files, running the simulations and plotting the results. In the following sections each of these aspects are explained.

Constructor

We can construct a l2-model by creating an instance of the `l2` class in Matlab. There are various ways this can be done, for example an empty model is created as follows.

```
model = l2();
```

However, most often an l2-model of some description in a model folder has to be created. In that case, the location of the model has to be passed on to the l2 constructor. This can either be relative from the current working directory or an absolute path. In either case, l2 will save this path as an absolute directory, such that the model after creation is independent of the working directory.

```
model = l2('model-folder')
model = l2('C:/path/to/model-folder')
```

Files. L2 looks for a number of files in the model folder. These filenames are by default `sorts.l2`, `predicates.l2`, `parameters.l2`, `scenarios.l2` and `rules.m`. These filenames can be changed, for example by adding the file to be changed and the new filename as additional arguments.

```
model = l2('model-folder', 'sorts', 's.l2', 'rules', 'generic.m')
```

Alternatively, if the model has already been created, the files structure can be accessed directly and change the filenames in that way.

```
model.files.sorts = 's.l2'
model.files.rules = 'generic.m'
```

When using this method, it is important to make sure that the .l2 files are read before the model is simulated. This can be done using the reset function explained below.

Functions

Below, functions to simulate a model and show or save the results are explained. Furthermore, a number of other functions are covered which might be useful in some circumstances.

Simulate. To simulate a model, this function is needed. For a simulation, a maximum number of time steps needs to be provided as well as a scenario and parameter set. For each time step, each of the functions in the rules file will be called once and results are added to the trace as the simulation progresses. Assuming there is only one scenario and parameter set defined (or a 'default' has been created in both), the most basic way of running a simulation is by providing solely the maximum number of time steps. To run the simulation using another scenario, the name of that scenario is added as a second argument. To use a different parameter set, its name can be added as a third argument. Therefore, when using a different set of parameters, the scenario name always needs to be added. Examples are shown below.

```
model.simulate(10);
model.simulate(10, 'alternative');
model.simulate(10, 'alternative', 'extreme');
model.simulate(10, 'default', 'extreme');
```

Plot. After running a simulation, the results can be shown using the plot command. This can also be used when errors occurred during the simulation to see what results were successfully derived. In its most basic form, this function can be called without any arguments to plot all results. Alternatively, a cell array can be passed along containing a set of predicates to be plotted. If the results do not fit on the screen, multiple windows will be created.

```
model.plot();
model.plot({'pred2', 'pred4'});
```

When plotting results, the graph differs depending on the sorts used in a predicate. Predicates with a single boolean value are plotted with a dark blue line on top when that predicate is true during that time point, a greenish line at the bottom if it is false, or nothing if it is unknown or does not exist. Predicates with a single real value are plotted as a graph, and predicates with a real value and one other sort are combined using multiple lines in the same graph. More default plotting functions might be added in the future, but if these results do not suffice you can add custom plotting functions. This is explained in the advanced features chapter below.

Export. Exporting results is similar to plotting results, but instead of showing the results on the screen, they are saved to one or more files. Therefore, a filename needs to be provided to the function, and in line with the plot function a cell array of predicates to plot may be added if desired. The resulting files will roughly be of size A4 or smaller and if results do not fit, multiple files are created. For the filetype, an attempt is made to read the extensions given in the filename. If this fails, a png will be created by default.

```
model.export('filename.pdf');
model.export('filename', {'pred2', 'pred4'});
```

Reset. If an instance of l2 is created, .l2 files are read. In some cases, the external file may change or a different file should be used. In those cases the reset function should be called to force the l2 class to reread these external files. It is possible to change the model folder or filenames while using reset by providing pairs of names and filenames. In contrast with the constructor,

a model folder cannot be simply added as the first argument, but needs to be preceded by a `model` argument.

```
model.reset();
model.reset('model', 'new_model_folder', 'sorts', 's.l2');
```

Validate. Each predicate that is added to the trace is validated against the definitions in the sorts and predicates files. If for some reason you would like to validate any predicate against a model description, this can be done by using this function and passing a predicate (in predicate form only) to this method. Returned is a boolean value indicating if the predicate adheres to the model definition and if not, the resulting error is returned as an optional second output value.

```
model.validate( predicate('name{value1,2}') );
[tf, err] = model.validate( predicate('name', {'value1',2}) );
```

A.1.6 Advanced features

The section below explains some more advanced uses of l2-matlab. For regular use, this functionality is not required, but it may be useful in some project.

Custom plot functions

There is a limited number of plotting functions available by default. In some circumstances, it can be useful to create your own function to plot some predicate. Therefore, l2-matlab tries to figure out before plotting what the best method is for a particular method. On top of that list are custom plotting functions defined in the model folder. Thus, for any predicate X, l2-matlab first checks the model folder for a function `plot_X.m`. If that function exist, it is used to plot that predicate, otherwise l2-matlab falls back on any of the default plotting functions.

When creating such a plotting function, two variables are passed to the function, the complete model and the predicate to be plotted. To be returned is an array of one or more figure handles and an equally long array of heights of those figures (in mm). L2-matlab will copy each of these figures in the overview, whereby the x-axis is changed to match simulation length (and any labels are cleared). The y-label is printed in bold face and should contain the title of the graph. Any legend in the graph is copied as well and placed horizontally in the best position.

```
function [figs, heights] = plot_X( model, pred )
   ...
end
```

How the figure or figures are created within the function is up to you. It might be helpful to base any custom function on any of the default plotting functions, which can be found in the `#l2-matlab` folder, under `@l2/private`. Furthermore, the example `03_IAPS_picures` also uses a custom plot function for the emotion predicate and can be used as an example.

Parameter tuning

Using a custom fitness function for your model, it is possible to tune one or more parameters using an evolutionary algorithm. In the following sections an explanation is given on how to write this fitness functions and subsequently how to use both the evaluate and tune functions.

Fitness. To start to evaluate your simulation results or tune any of the parameters, an fitness function is required. This consists of a separate .m file in your model folder, containing a fitness function using the following format.

```
function [ score ] = fitness( model )
   ...
end
```

As you can see, the model is provided and only a single score should be returned. The model itself always contains a trace file containing the simulation results to be scored. Furthermore, it is necessary to write your fitness function in such a way that a score of 0 represents a perfect result, with higher scores representing worse results.

Evaluate. The function evaluate is used to calculate the fitness of a particular trace. The basic way of using this function is as follows, resulting in a fitness score of the current trace.

```
model.evaluate()
```

However, it is also possible to provide additional information, in which case those arguments will be used to first simulate the model after which the resulting trace will be evaluated. Thus, any method of using the simulate function can be used for calling the evaluate function as well.

Tune. When a fitness function is available, it is possible to tune one or more parameters of your model. For this, an evolutionary algorithm is used to come up with a good set of parameters based on that fitness function. Take note that using this function might require a large amount of time as many simulations have to be run in order to evaluate numerous parameter values. Furthermore, as this method is made to be as generic as possible, there is no guarantee what so ever that the resulting parameters are optimal or even very good. Having said that, let's take a look at how to use this function.

```
model.tune({'param1', [0 1], 'param2', 'SORT1'}, 5)
```

As a first argument, a cell array of parameters to be tuned is given. Following each of the parameter names in the cell array, the possible values of that parameter are given. For numeric parameters either a range of real numbers is given by its minimal and maximal value as for `param1` above, or the precise set of possible values can be given as well. The other possibility is to provide a specific (non-numeric) sort as the possible values for a parameter, for example `BOOLEAN` or a custom sort as is done for `param2`. As a second argument for tuning, the simulation length for each evaluation is given. Thus, in this example, two parameters are tuned on simulations of length 5.

Above, the model is tuned on the default scenario. However, it is also possible to tune on a different scenario or even multiple scenarios at once. As a third argument, the name or names of those scenarios are given, as shown below.

```
model.tune({..}, 5, 'alternative')
model.tune({..}, 5, {'default' 'alternative'})
```

Optionally, a fourth and last argument can be added to change the base parameter set used for simulating the model. As it is not required to tune all parameters, this provides the flexibility to change the remaining parameters using the parameter sets as defined in `parameters.l2`. As with simulating models, this requires the scenario to be specified even when the default scenario should be used.

```
model.tune({..}, 5, 'default', 'extreme')
```

A.2 Introductionary examples

A.2.1 Leadsto tutorial

Consider an experimental setting, involving two positions (say, p1 and p2), an animal, a piece of food, and a transparent screen (e.g., a window). Suppose

the animal is placed at position p1, the food is placed at p2, and the screen is placed in between, separating the animal from the food. Multiple trials are performed in which, at some variable moment, the screen is raised, and the animal is free to go to any position. After a number of trials, it turns out that a regularity can be observed in the behavior of the animal. This regularity can be expressed informally by the following property:

Every time that the agent observes that there is food at p2,
and no screen is present, it will go to p2.

In semi-formal form, this property can be written as follows:

```
LP1 (Local Property 1)
at any point in time,
if    the agent observes that food is present at position p2
and   it observes that no screen is present,
then  it will go to position p2
```

Implementation

To implement this model, we create the following files.

sorts.l2

The only sort we will use is `BOOLEAN`, which is defined by default. Therefore, there is no need for a `sorts.l2` file.

predicates.l2

```
goes_to_p2;                BOOLEAN
observes_food_at_p2;       BOOLEAN
observes_no_screen;        BOOLEAN
```

These are the three predicates used in LP1, each of them using a boolean value.

scenarios.l2

```
default (
   observes_no_screen{true};   3
   observes_food_at_p2{true};  [1:3]
)

no_food (
   observes_no_screen{true};   3
)
```

Here we define our scenarios, in this case two. In the first scenario the animal observes food at time points 1, 2 and 3 and no screen at time point 3. In the second scenario, the animal observes no screen on time point 3, but does not observe any food.

parameters.l2

This example does not make use of any parameters.

rules.m

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

%ADD RULES BELOW
function result = ddr1( trace, params, t )
    move = trace(t).observes_food_at_p2 & ...
              trace(t).observes_no_screen;
    result = {t+1, predicate('goes_to_p2', move)};
end
```

Our `rules.m` file only contains one rule, namely lp1. Here, we check whether both `observes_food_at_p2` and `observes_no_screen` are true at time point t in the current trace. The comparison tells us whether the animal goes to p2 at the next time step, thus this is our result for `goes_to_p2` at time point $t+1$.

Running the example in Matlab

```
clear all
close all
model = l2('01_leadsto_tutorial')
model.simulate(5)
model.plot()
```

This code clears all variables and closes all windows, after which it loads, runs and plots the model. The resulting plot looks as follows.

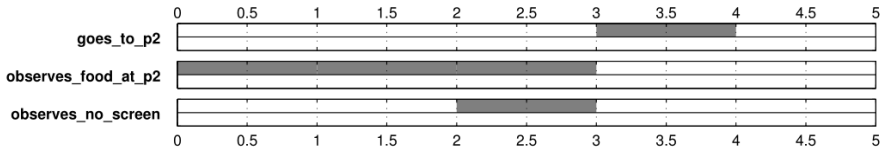


Figure A.2: Simulation results for Leadsto tutorial

A.2.2 Emotion regulation

Emotion regulation refers to 'all of the conscious and nonconscious strategies we use to increase, maintain, or decrease one or more components of an emotional response' (Gross, 2001). This ability to regulate our own emotional state is one of the properties that distinguish humans from other higher animals, providing us a high behavioral flexibility, which may result in a higher sense of well-being and mental health. A specific strategy for emotion regulation that is widely studied is reappraisal. Gross (2001) defines reappraisal as a process where 'the individual reappraises or cognitively re-evaluates a potentially emotion-eliciting situation in terms that decrease its emotional impact'.

A simple simulation model of reappraisal, which is based on the model presented in Bosse et al. (2010) consists of the following two difference equations:

1. $new\_ERL = ERL + (1 - \beta) * (((1 - w) * v_1 + w * v_2) - ERL)\Delta t$
2. $new\_v_2 = v_2 - (\alpha_2 * d/d_{max})\Delta t$ (where $d = ERL - ERL_{norm}$)

Implementation

To implement this model, we create the following files.

sorts.l2

The only sort we will use is `REAL`, which is defined by default. Therefore, there is no need for a `sorts.l2` file.

predicates.l2

```
erl;                    REAL
d;                      REAL
v2;                     REAL
```

These are the three values calculated in the description above. Note, we could also calculate d 'on the fly' in the rule for `v2`, thereby eliminating the need for this predicate.

scenarios.l2

```
erl{0};    1
d{0};      1
v2{0};     1
```

We only define one scenario, with all the values at time point 1 being 0.

parameters.l2

```
default(
   beta;       0
   w;          0
   v1;         0
   a2;         0
   delta_t;    0
   d_max;      1
   erl_norm;   0
)

reappraisal(
   beta;       0.5257
   w;          0.5773
   v1;         0.4170
   a2;         0.0426
   delta_t;    0.0439
   d_max;      1
   erl_norm;   0
)
```

This example uses multiple parameters. We define two sets, one default set containing meaningless values and one set tuned to resemble a reappraisal pattern.

rules.m

```
function [ fncs ] = rules()...

%ADD RULES BELOW
function result = ddr1( trace, params, t )
    erl_new = trace(t).erl + (1-params.beta) * ...
                (( (1-params.w) * params.v1 + ...
                params.w * trace(t).v2 ) - ...
                trace(t).erl) * params.delta_t;
    result = {t+1, 'erl', erl_new};
end
```

rules.m (continued)

```
function result = ddr2( trace, params, t )
    v2_new = trace(t).v2 - ( params.a2 * trace(t).d / ...
                params.d_max ) * params.delta_t;
    result = {t+1, 'v2', v2_new};
end
function result = ddr3( trace, params, t )
    d_new = trace(t).erl - params.erl_norm;
    result = {t+1, 'd', d_new};
end
```

Our `rules.m` file only contains the two formulas given in the text, as well as a separate function for d.

Running the example in Matlab

```
clear all
close all
model = l2('02_emotion_regulation')
model.simulate(100, 'default', 'reappraisal')
model.plot({'erl'})
```

This code clears all variables and closes all windows, after which it loads, runs and plots the model. For this simulation run, the default (and only) scenario is used, but the parameter set 'reappraisal' is used instead of the set of default values. For the plot, only 'erl' is of our interest.

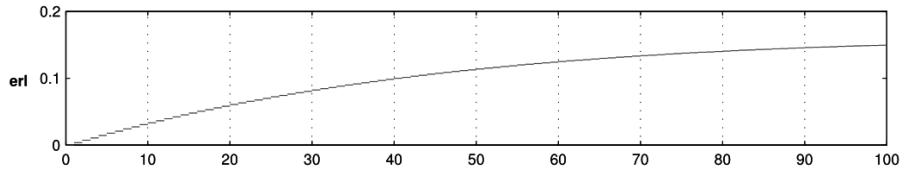


Figure A.3: Simulation results for emotion regulation

A.2.3 IAPS pictures

'The International Affective Picture System (IAPS) is [..] a set of normative emotional stimuli for experimental investigations of emotion and attention.' (Lang et al., 1997) Each of these pictures has a rating for both valence and arousal. Consider an agent viewing a number of these pictures. The valance

and arousal experienced by that agent will adept to those of the pictures viewed. The speed of this process depends on the speed parameters w_1 and w_2 respectively.

```
DDR1 (Domain Dynamic Relation 1)
at any point in time, for each agent X
if    agent X has a valence V and arousal A
and   an agent X observes a picture P from the IAPS picture set
and   picture P has a valance rating V1 and arousal rating A1
then  agent X will get a valence of V+w1(V1-V)
      and arousal of A+w2(A1-A)
```

Implementation

For the implementation of this model, the following files are created.

sorts.l2

```
AGENT;          arnie, bernie, charlie
PICTURE;        p1, p2, p3
```

We consider here both agent and pictures as sort, with 3 agents and 4 pictures possible.

predicates.l2

```
% picture(PICTURE, VALENCE, AROUSAL)
picture;        PICTURE, VALENCE, AROUSAL
% emotion(AGENT, VALENCE, AROUSAL)
emotion;        AGENT, VALENCE, AROUSAL
observes;       AGENT, PICTURE
```

For this model, three predicates are necessary. The first, `picture`, describes the valence and arousal of each picture. The second, `emotion`, describes the current valance and arousal value of an agent. The last predicate, `observes`, is used to denote if an agent views a particular picture.

parameters.l2

```
w1;         0.1
w2;         0.1
```

To keep it simple, we only consider one possible set for the parameters given in the model description.

scenarios.l2

```
% IAPS picture valence/arousal
picture{p1, 1, 9};          [1:Inf]
picture{p2, 9, 9};          [1:Inf]
% Starting emotion of the agents
emotion{arnie, 4.5, 4.5};  1
emotion{bernie, 4.5, 4.5}; 1
% Observing of pictures
observes{arnie, p1};        [1:25]
observes{bernie, p1};       [1:25]
observes{arnie, p2};        [26:50]
```

One default scenario is defined, using two pictures and two agents. The first lines describe the static valence and arousal values for the two pictures; one very arousing sad picture (p1) and another very arousing happy picture (p2). The next lines describe the starting emotion values of arnie and bernie. The last lines describe that both agents view the sad picture after which only arnie looks at the happy picture.

rules.m

```
%ADD RULES BELOW
function result = ddr1( trace, params, t )
  result = {};
  %go through each agent's emotion
  for emotion = trace(t).emotion
    x = emotion.arg{1}; %agent name
    v = emotion.arg{2}; %valence
    a = emotion.arg{3}; %arousal
    % go through each picture that agent observes
    for observes = l2.getall(trace,t,'observes',{x,NaN})
      p = observes.arg{2}; %picture name
      %get the valence and arousal of that picture
      [~,pic] = l2.exists(trace,t,'picture',{p,NaN,NaN});
      v1 = pic.arg{2}; %picture valence
      a1 = pic.arg{3}; %picture arousal
      %adjust v and a accordingly
      v = v + params.w1 * (v1 - v);
      a = a + params.w2 * (a1 - a);
    end
```

rules.m (continued)

```
    %add the new emotion level to the results
    result = { result{:} {t+1, 'emotion', {x, v, a}} };
  end
end
```

One, relatively large rule is needed to implement this model. The majority of the rule is part of a for loop going through the emotion of each agent. For that agent, each picture it observes is retrieved. For those pictures, the valence and arousal value is looked at and using those values the emotion level of the agent is changed. When each picture an agent looked at is processed, the new emotion level is added to the results.

Running the example in Matlab

```
clear all
close all
model = l2('03_IAPS_pictures')
model.simulate(50, 'default', 'reappraisal')
%Plot is by default too large for the screen,
%but a custom plot_emotion.m has been written.
model.plot('emotion')
```

The plot is shown below, but as you might have noticed uses a custom `plot_emotion.m` that is residing in the model folder. More on how to write custom plot functions can be found in the manual.

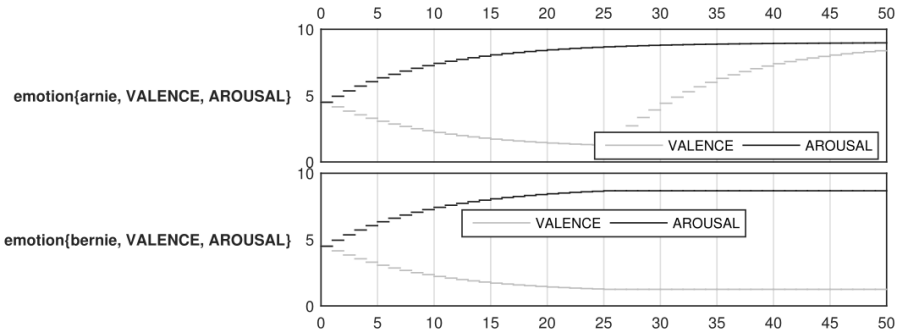


Figure A.4: Simulation results for IAPS pictures

A.2.4 Theory of minds

One of the most important milestones in theory of mind development is gaining the ability to attribute *false belief*: that is, to recognize that others can have beliefs about the world that are diverging. In the 'appearance-reality', or 'smarties' task, experimenters ask children what they believe to be the contents of a box that looks as though it holds a candy called smarties. After the child guesses (usually) smarties, it is shown that the box in fact contained pencils. The experimenter then re-closes the box and asks the child what she thinks another person, who has not been shown the true contents of the box, will think is inside. The child passes the task if he/she responds that another person will think that there are smarties in the box, but fails the task if she responds that another person will think that the box contains pencils. Gopnik & Astington (1988) found that children pass this test at age four or five years.

Implementation

This experiment has been implemented as follows.

sorts.l2

```
AGENT;          arnie, bernie, charlie
INFO;           smarties, pencils, <belief>
```

As can be seen, we have three different agents, and some `INFO` element which holds information about the contents of the box (smarties or pencils). However, it is also stated that the `INFO` element can be `<belief>`, meaning that instead of either smarties or pencils a predicate `belief` is also a valid `INFO` element. This predicate, as well as two others are defined below.

predicates.l2

```
age;            AGENT, REAL
belief;         AGENT, INFO
observe;        AGENT, INFO
```

Three predicates are used, one describing the age of the agent, one for the beliefs of the agent and a third predicate for the observations of an agent. Keep in mind, that the belief can be either smarties, pencils or a belief about some other agent having a particular belief.[2]

[2]Using this formalization, an agent can also have an observation about some particular belief. This however is not used or required for modeling this experiment.

parameters.l2

```
age_tom;        4
```

One parameter is used for this model, setting the age from which an agent can apply theory of mind, thereby understanding that a different person, not having seen the contents of the box, will falsely belief that there are smarties instead of pencils in the box.

scenarios.l2

```
default (
  age{arnie, 3};                              [1:5]
  belief{arnie, smarties};                    [1]
  belief{arnie, belief{bernie, smarties}};    [1]
  observe{arnie, pencils};                    [2]
)

older (
  age{arnie, 5};                              [1:5];
  belief{arnie, smarties};                    [1];
  belief{arnie, belief{bernie, smarties}};    [1];
  observe{arnie, pencils};                    [2];
)
```

We define two scenarios, one where arnie is too young to apply theory of mind, and another where he is old enough. Furthermore, we initially set both the belief of arnie that there are smarties in the box and his belief that he beliefs that bernie thinks there are smarties in the box. Lastly, at timepoint 2 arnie observes there are pencils in the box.

Below, the rule to implement this model is shown. It cycles through all beliefs and for each beliefs considers all observations made by the agent that has that belief. For each observation, we first check whether the belief was a belief of the agent itself or a nested belief about some other agents belief. This is done by checking if the observation is a structure, which it would be in the case of a nested belief. If it isnt a structure, we update the belief with the current observation. Otherwise, we need to check whether or not the agent is able to apply theory of mind. If not, the nested belief also gets updated to match the current observation.

rules.m

```
%ADD RULES BELOW
function result = ddr1( trace, params, t )
  result = {};
  %go through each belief
  for belief = trace(t).belief
    agent = belief.arg{1}; %agent name
    belief = belief.arg{2}; %the belief or a nested belief
    % go through each observation of that agent
    for observe = l2.getall(trace,t,'observe',{agent, NaN})
      observation = observe.arg{2}; %info or nested
      %belief is own belief, not a nested belief
      if ~ispredicate(belief)
        belief = observation;
      %belief is a (nested) belief of someone else
      else
        %get age to check old enough to apply ToM
        [~, age] = l2.exists(trace,t,'age',{agent, NaN});
        tom = age.arg{2} > params.age_tom;
        %if no ToM, simply update nested belief
        if ~tom
          belief = predicate('belief', ...
                             {belief.arg{1} observation} );
        end
      end
    end
    result = { result{:} {t+1, 'belief', {agent,belief}} };
  end
end
```

Although this method works, it relies on the assumption that any nested belief does not contain another nested belief. For example, consider the following belief where arnie believes that bernie believes that charlie believes the box contains smarties.

```
belief{arnie, belief{bernie, belief{charlie, smarties}}}
```

In order to cope with any number of nested beliefs, the rule needs to be implemented differently using a form of recursive programming. This method will be explained next and is included in the example code as well (using a different name for the rules file). For the given scenarios, results for both methods are similar. Below, the rule to implement this rule recursively is shown. The

first part of this rule is similar as different from before. It cycles through all the beliefs and updates them for the relevant observations. However, beliefs are now updated by a nested function `updateBelief`, such that it is possible to cope with any amount of nested beliefs.

recursive.m

```
%ADD RULES BELOW
function result = ddr1( trace, params, t )
  result = {};
  %go through each belief
  for belief = trace(t).belief
    agent = belief.arg{1}; %agent name
    belief = belief.arg{2}; %the belief or a nested belief
    % go through each observation of that agent
    for observe = l2.getall(trace,t,'observe',{agent, NaN})
      observation = observe.arg{2}; %info or nested
      %get age to check old enough to apply ToM
      [~, age] = l2.exists(trace,t,'age',{agent, NaN});
      tom = age.arg{2} > params.age_tom;
      belief = updateBelief(belief, observation, tom);
    end
    result = { result{:} {t+1, 'belief', {agent,belief}} };
  end

  function belief = updateBelief(belief, observation, tom)
    %belief is a info element, not a nested belief
    if ~ispredicate(belief)
      belief = observation;
    %belief is a (nested) belief of someone else
    else
      %if no theory of mind, simply update nested belief
      if ~tom
        belief = predicate('belief', {belief.arg{1} ...
          updateBelief(belief.arg{2}, observation, tom)});
      end
      %otherwise, do noting with the belief
    end
  end

end
```

Consider the function `updateBelief` and firstly notice that it is part of the function `ddr1`. It takes as input the current belief, the observation and whether or not theory of mind is applicable. If the belief is not a predicate (~`ispredicate(belief)`), the belief becomes equal to the observation. Otherwise, if the belief is a nested belief, that belief is only updated if no theory of mind is applied. The new belief is a new structure, named belief (the predicate name). The value for the agent is equal to the agent name in the previous belief. To update the belief in this nested belief, the `updateBelief` function is called again with the belief value from the nested belief, the current observation and whether or not theory of mind is applied. Using this recursive structure, it does not matter how many nested beliefs there are, they always get updated as expected.

Running the example in Matlab

```
clear all; close all;
model = l2(close '04_theory_of_mind')
model.simulate(3);
model.plot();
set(gcf,'name','Default scenario','numbertitle','off')
model.simulate(3, 'older');
model.plot();
set(gcf,'name','Older scenario','numbertitle','off')
```

To run this example, the model is simulated for both scenarios. The resulting graphs are shown below as well. Note the difference between the age of arnie and whether or not the nested belief of bernie gets updated based on the observation arnie makes at timepoint 2.

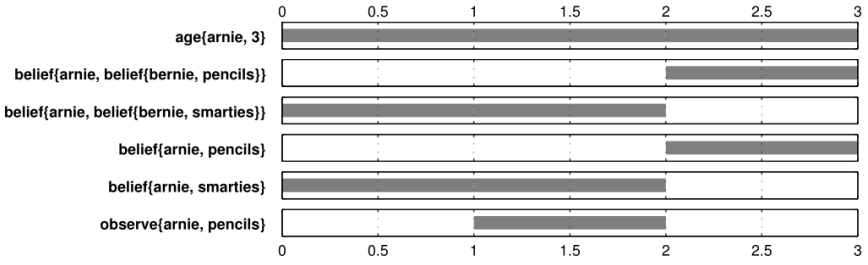


Figure A.5: Simulation results for theory of mind, default scenario

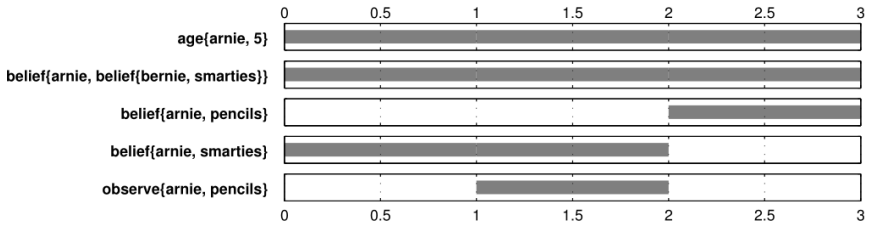


Figure A.6: Simulation results for theory of mind, older scenario

References

Bosse, T., Pontier, M., & Treur, J. (2010). A computational model based on gross' emotion regulation theory. *Cognitive systems research*, *11*(3), 211–230.

Gopnik, A., & Astington, J. W. (1988). Children's understanding of representational change and its relation to the understanding of false belief and the appearance-reality distinction. *Child Development*, *59*(1), 26–37.

Gross, J. J. (2001). Emotion regulation in adulthood: Timing is everything. *Current directions in psychological science*, *10*(6), 214–219.

Lang, P. J., Bradley, M. M., & Cuthbert, B. N. (1997). International affective picture system (iaps): Technical manual and affective ratings. Tech. rep., NIMH Center for the Study of Emotion and Attention, Gainesville, FL.