SBMLevolver: Documentation

December 3, 2008

Abstract

This documentation describes the SBMLevolver, a software tool developed for the artificial evolution of biological network models.

In the course of the ESIGNET project (www.esignet.net), the software program SBMLevolver has been developed, built around an evolutionary algorithm (EA) that evolves artificial CSNs (represented in the SBML language [1]) performing pre-specified functions. It works directly on SBML, the most common interchange format for biochemical models, which can be simulated and analysed by a variety of other tools. We believe that by using SBML, model interchange and comparison as well as the usability of the program for the Systems Biology community will be greatly enhanced. Provided and distributed as open source software, the SBMLevolver is exclusively based on freely available libraries and packages.

Contents

1 Installation **2**

2 Outline of the software system **2**

3 User interface **2**

4 Example: Evolving an addition-network **5**

1 Installation

For information on how to install the system, see the file `INSTALL` in the root directory.

2 Outline of the software system

The program SBMLevolver has been written entirely in C/C++ and can be used from the command-line (details on the usage are given in section 3). The program has been implemented as a set of modules or classes. The main modules include the user interface, population management, fitness evaluation, fitting of model parameters, and the creation of new models via mutation and recombination.

- **BuildingBlocks:** stores and provides the elements from which the models are build
- **CentralControl:** starts and stops the program, provides user interface
- **Individual:** encapsulates a model together with its additional attributes and methods
- **Mutator:** generates new models by mutation and recombination
- **ObjectiveFunction:** stores the evolutionary objective and provides functions to calculate the fitness of a model regarding this objective
- **ParameterFitter:** fits model parameters w.r.t. the given objective
- **ParamObj:** stores and provides all parameters of the current run
- **Population:** handles a population of models, manages selection and reproduction
- **RandomNumberGenerator:** facilitates the generation of random numbers
- **sbmltools:** contains all methods manipulating SBML structures

3 User interface

Currently, the SBMLevolver is command-line based, but work on a graphical interface is in progress. As an experimental software, the system is very flexible and most parameters of the algorithm can be defined by the user. A full list of possible options is given below. To run the program, an objective function, a set of building blocks for the networks, and the options have to be specified in text files, examples of which are included with the software.

3.1 The options

In this file (default is `option.txt`), the algorithmic parameters and any other controls are given as:

```
<parameter> <value> [further values]
```

These can also be accesses by calling the programm with the `-h` or `--help` option. Each parameter is given on a new line. Here is a list of all available parameter options:

Input & Output

output_file	output file (additional to stdout)
input_dir	input directory (working directory before run)
output_dir	result directory (working directory while and after run)
objective	objective function file (default: objectivefunction.txt)
bblocks	building blocks file (default: buildingblocks.txt)
first_model	initial ”seed” model file
save_every	number of generations after which to save models

num_save	number of models to save from population every *save_every* turn
Experiment Settings	
num_pop	number of populations (default 1)
iso_time	isoluation time when using multiple populations
mu	population size (default: 10)
lambda	number of offspring (default: 100)
comma_selection	if given, comma selection is used instead of plus selection
mutation_probs	relative frequencies for the different kinds of mutations (default: 1 1 1 1 1 1 1 0; no mutation, add/delete species, replace reaction, add/delete reaction, duplicate species)
max_mutations	maximal number of mutations per mutation-turn (default: 1)
cross_prob	probability of crossover instead of mutation
pf_settings	population size, number of offspring, number of generations of local parameter fitter (CMA-ES-algorithm, (mu,lambda) strategy)
Termination Conditions	
stop_time	time limit (in seconds)
stop_turn	iteration limit
stop_fitness	fitness to reach (default: 0.0)

3.2 The objective function

The modular architecture allows the user to specify his or her own fitness function in as a separate file in the directory `src/objectives`. This file then has to be included into the class `ObjectiveFunction` (just follow the way `io-table.cpp` is included now). However, a default objective function based on an input-output table is available which can readily be used. A text file `objectivefunction.txt` has to be provided which specifies the desired input-output behaviour of selected species in the evolved models. In the default case, fitness is then calculated as the mean square difference between the desired and the realised timecourses. An example objective function file is given in figure 4.

3.3 The building blocks

The software system deals with SBML models as solution candidates for the optimisation problem. Since these can have nearly arbitrary structure, it is necessary to lay down a-priori model specifications that the optimisation process can use. This is done in the file `buildingblocks.txt` and handled in the class `BuildingBlocks`. The building-blocks specification consists of the following elements:

- The minimum and maximum number of species that can be used in the model
- The minimum and maximum number of reactions
- Lower and upper bounds for the initial concentration of each species
- A list of allowed reaction mechanisms, given in the following format (on one line):

  ```
  Re1 + Re2 + ... | Mod1 + Mod2 + ... -> Pr1 + Pr2 + ... :
  kinetic_law number_of_parameters (name lb ub)* [T]
  ```

  where (name lb ub)* denotes a series of parameter names followed by their lower and upper bounds, and the optional flag `T` at the end denotes transport reactions, which can involve species from different compartments (others cannot).

The reserved names `Reactant<num>`, `Modifier<num>`, and `Product<num>` can be replaced by arbitrary species in the evolver. This way, a mix of species-specific and general reaction mechanisms can be specified. E.g., the reaction

```
Reactant0 + 02 -> Product0
```

describes the oxidization of an arbitrary species, i.e. the reaction of an arbitrary species with Oxygen to another arbitrary species. An example of a `buildingblocks.txt` file is given in figure 3.

3.3.1 Importing building blocks automatically

The class `BuildingBlocks` provides the method `learnReactionsFromModel`, which loads reaction mechanisms from a given SBML model into the building-blocks (abstracting from the specific participants in the original reaction). This can be used to construct a library of reaction mechanisms from published SBML models.

3.4 The initial population

For the initial population, the program will load any SBML models in files of type `<modelname>.model.xml` from the current directory, and fill it up with random SBML models containing the number of species and reactions specified in the options, as well as any species required for the objective function. Also, it is possible to specify one initial starting model that is mutated once to create initial solutions (using the `first_model` option in `options.txt`). If no initial population is given, random models are created, for which the minimal and maximal size is specified in the options.

3.5 Running the optimisation

To start the evolver on a single machine, run

```
sbmlevolver [<arguments>]
```

from the command line. By default, arguments are read from file `options.txt`. If a different options file is wanted, this has to be given directly after the call. After this, any options can be given, overriding the options specified in the optionsfile.

During the run, information about the fitness-development of the population is displayed on the screen. There are two ways to finish the program: either the pre-specified fitness, number of iterations or runtime is reached, or the user sends the signal SIGUSR1 to the process (`kill -SIGUSR1 <pid>`). After finishing, the program stores the population of the last generation in separate SBML files of the format `result_<id>_<fitness>.model.xml`. With these files, a run can be restarted from the exact point where it terminated.

3.6 Looking at the results

Results of the evolution run are given in two ways: First, running statistics about the population are displayed, such as the best and average fitnesses, and mean numbers of species and reactions. Using the `output_file` option, this can be stored into a file and used for later evaluation. Second, the final population is stored as SBML models, which can be investigated using standard Systems Biology software tools (see list on www.sbml.org). Since the SBML_odeSolver has to be installed anyway, this program (and its associated tools such as the ParameterScanner) can be used to simulate, visualise and analyse the models.

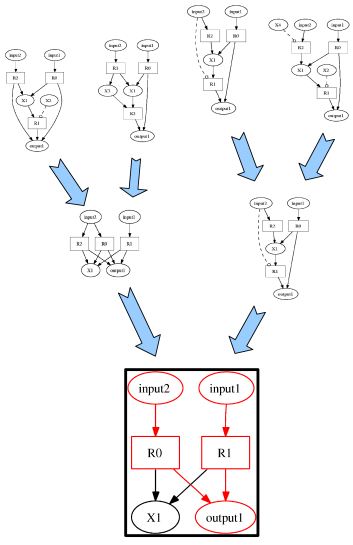

Figure 1: Snapshots from the evolution of an addition-network, including the final result at the very bottom. Node X1 is not needed for the computation and could be regarded as “junk”.

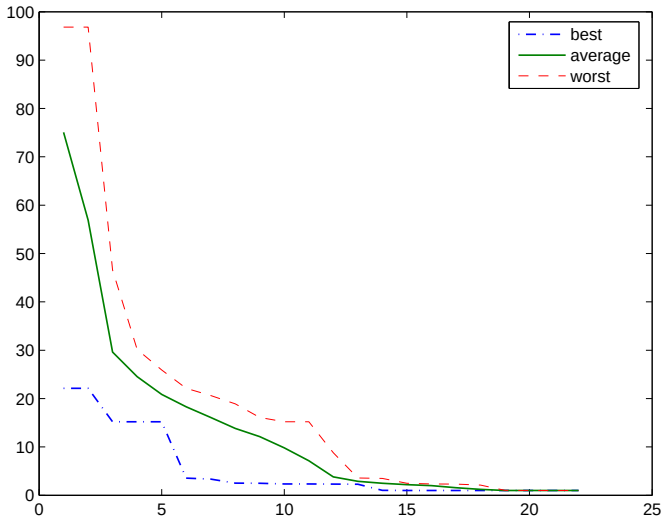

Figure 2: Development of the best, average and worst fitness during a typical run.

Figure 1 shows a series of snapshots from the evolution of a network capable of adding two numbers, while the corresponding fitness development is shown in figure 2. If the evolution gets stuck in a (supposedly local) optimum, one can stop and restart it with a different set of parameters, using the final population of the first run as an initial population for the second.

3.7 The fitness evaluator subprogram

If one just wants to test single models, the SBMLevolver has to be called as

```
sbmlevolver --get-fitness <name-of-model-file>
```

which calculates the fitness of the SBML model with respect to a given objective function. It mainly utilises the `FitnessEvaluator` and the `ObjectiveFunction` classes of the main program. The selection of an appropriate objective function works in exactly the same way as for the main program, using the text file `objectivefunction.txt`. The program will simulate the model, compare its behaviour to the one specified in the objective function, and return the calculated fitness value.

4 Example: Evolving an addition-network

In this example, the evolution of a simple network, capable of the addition of two positive real numbers, is described. The two numbers will be encoded as the initial concentrations of two input species *input1* and *input2*, while the output of the calculation will be found in the concentration of species *output* at time 10.0.

As a first step, we have to prepare the building-blocks for the evolver. This is done in the file `buildingblocks.txt` (figure 3). As you can see, we only use mass-action kinetics here. We also specify boundaries for the parameters, which helps to avoid numerical problems in the integration when the time-scales become too separated.

The second item we have to think about is the objective function (shown in figure 4). We choose to use four fitness cases, taking 0 and 10 as values for both *input1* and *input2* (more should be used, but this is not done here due to space constraints). We will measure the concentration of *output* at four timesteps of 2.0 seconds each, ignoring the first one (offset). Since the input is only to be specified at $t = 0$, we put a '*'

```
# Maximum number of species
5
# Maximum number of reactions
5

# Lower and upper bound for species initial concentrations
0
10

# Available reaction types (format: numRe, numPr, numMo, formula,
# numPara, [paraId, lb, ub]*)
# Some possible reactions with rate 0 are included to ease moving
# in search space
0 -> Product0 : 0 0
0 | Modifier0 -> Product0 : 0 0
0 -> Product0 + Product1 : 0 0
0 | Modifier0 -> Product0 + Product1 : 0 0
Reactant0 -> 0 : 0 0
Reactant0 | Modifier0 -> 0 : 0 0
Reactant0 | 0 -> Product0 : k*Reactant0 1 k 0 5
Reactant0 | Modifier0 -> Product0 : k*Reactant0*Modifier0 1 k 0 5
Reactant0 -> Product0 + Product1 : k*Reactant0 1 k 0 5
Reactant0 | Modifier0 -> Product0 + Product1 : k*Reactant0*Modifier0 1 k 0 5
Reactant0 + Reactant1 -> 0 : 0 0
Reactant0 + Reactant1 -> Product0 : k*Reactant0*Reactant1 1 k 0 5
Reactant0 + Reactant1 -> Product0 + Product1 : k*Reactant0*Reactant1 1 k 0 5
Reactant0 <-> Product0 : k*(Reactant0-Product0) 1 k 0 100 T
```

Figure 3: An example for file `buildingblocks.txt`

in front of its concentration. We do not want to use Akaike's Information Criterion to modify the fitness (`noAIC`), and we want to have a mass-conserving and small network (penalties for non-massconservance and size).

After the general setup of the run has been determined in the two textfiles, we have to decide about the actual parameters of the evolutionary algorithm, given in file `options.txt` (see figure 5). In this case, we choose to use a population size of 20, producing 80 offspring in a overlapping fashion, such that selection acts on 100 individuals. The population is then refilled by mutations and crossover on the survivors. All survivors are selected by elitist selection. Every tenth offspring should be produced by crossover between survivors. This is a very strong selection pressure, so it will only be successful in simple cases such as this example. In each fitness evaluation, the model parameters are be improved in a (1,5)-ES for 15 generations. Since we use default filenames for all files, we can call the SBMLevolver without command-line arguments.

The output on the screen shows best and average fitness as well two diversity measures and the mean number of species and reactions:

```
Turn    Best      Avg       Diff1     Diff2     Species   Reactions
    1   6.431398 26.97341 7.960000 5.640000 3.200000 4.800000
    2   0.084014 8.160019 9.840000 6.480000 3.800000 6.600000
    3   0.002412 4.835043 13.98000 7.720000 4.600000 8.900000
    4   0.002412 3.037237 16.78000 8.660000 5.300000 11.400000
    5   0.001200 0.912607 16.54000 9.440000 5.800000 12.200000
    6   0.000758 0.007115 9.980000 9.180000 5.900000 12.100000
    7   0.000246 0.001053 12.52000 9.240000 6.000000 13.700000
    8   0.000098 0.000412 15.76000 9.340000 5.900000 15.700000
    9   0.000088 0.000248 17.84000 9.620000 5.700000 16.400000
   10   0.000057 0.000136 20.82000 10.46000 6.300000 17.600000
   11   0.000035 0.000107 25.52000 11.18000 6.700000 20.000000
   12   0.000023 0.000070 21.20000 11.20000 7.400000 18.100000
```

```
io-table
# Addition network. Input1(t=0) + Input2(t=0) = Output(t=inf)
# Determine whether Akaike's Information Criterion should be used
noAIC
# Given the penalty for non-massconservance
10000
# Give penalty for size
10
# No. of cases, inputs, outputs, timesteps, offset, and timestep
4
2
1
5
2
2.0

# Input species
input1
input2
# Now give the names of the output species
output

# Initial input concentrations
# Starting with * sets the concentration only at the first timestep
# Only one number means the concentration is constant the whole time
# Case 0
* 0
* 0
# Case 1
* 0
* 10
# Case 2
* 10
* 0
# Case 3
* 10
* 10

# Now the output data comes
# Case 0
0
# Case 1
10
# Case 2
10
# Case 3
20
```

Figure 4: An example for file `objectivefunction.txt`

```
mu              20
lambda          80
cross_prob      0.1
output_file     run.log
stop_fitness    0.00001
stop_turn       100
pf_mu           1
pf_lambda       5
pf_num_gen      15
```

Figure 5: An example for file `options.txt`

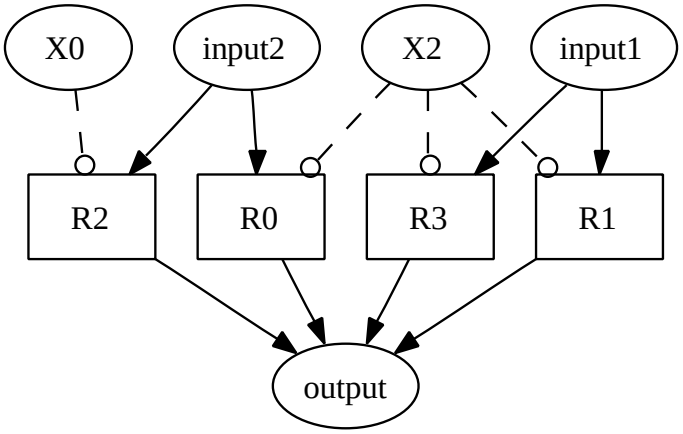

Figure 6: A network resulting from the evolution of an addition. Although it looks more complicated than a hand-designed network, it perfectly performs its task.

```
   13  0.000010 0.000053 24.00000 11.74000 8.100000 19.500000
   14  0.000010 0.000047 29.48000 13.62000 8.700000 23.700000
   15  0.000003 0.000035 33.38000 14.88000 9.000000 27.600000
Run finished: reached desired fitness
```

The best model after this run can be seen in figure 6. By playing with the penalties and AIC, we would hope to find a more elegant solution that focuses on the bare necessities of the problem.

References

[1] A. Finney and M. Hucka. *Systems biology markup language: Level 2 and beyond*. Biochem Soc Trans, 31(Pt 6):1472–1473, Dec 2003.