simgi - A Stochastic Gillespie Simulator for Molecular Systems

Author: Markus Dittrich
email: haskelladdict at users dot sourceforge dot net
Version: 0.2 (11/24/2009)

Contents

Introduction

simgi is a fairly simple and straightforward stochastic simulator based on Gillspie's[1] direct method. **simgi** is implemented in pure Haskell, command line driven and comes with a flexible simulation description language called Simgi Model Generation Language (SGL). More information is available from the project summary page.

Status

The 0.2 release of **simgi** provides a fully functional simulator which has been tested on several model systems some of which were fairly large. The engine itself is not yet fully optimized for speed. Furthermore, version 0.1's random number generator (RandomGen) has been replaced which a much more powerful and faster implementation of a 64bit Mersenne Twister.

Download

The current release of simgi can be downloaded here.

Compilation

Compilaton of **simgi** requires

- >=ghc-6.10
- >=gmp-4.3
- >=mersenne-random-pure64

To compile the documentation (not required), you will also need

- >=docutils-0.5
- latex, e.g., tetex or texlive

Building of **simgi** can be done either via

- the standard `make, make check, make install`
- or via cabal

Simgi Model Generation Language (SGL)

simgi simulations are described via Simgi Model Generation Language (SGL). The corresponding simulation files typically have an *.sgl* extension, but this is not enforced by the **simgi** simulation engine.
A SGL file consists of zero or more descriptor blocks of the form

```
def <block name>

  <block content>

end
```

The formatting of the input files is very flexible (but see[3]). In particular, neither newlines[4] nor extraneous whitespace matter. Hence, the above SDL block could also be written on a single line. However, it is strongly recommended to stick to a consistent and "visually simple" layout to aid in "comprehending" the underlying model. Also, it is worth to point out that **simgi**'s parser is case sensitive.
Comments can be added to the SGL file and are parsed according to the Haskell language specs

- simple line comments begin with a `--` token and treat everything until the next newline as a comment, including valid SDL commands. Hence, SDL blocks containing line comments need to be separated by newlines in order to be parsed correctly.
- block comments begin with a `{-` token and end with a `-}` token. Everything within a comment block is ignored by the parser and block comments can be nested.

An important concept inside SGL is the one of an *expression statement*. These are enclosed in curly braces and can contain any mathematical expression involving doubles, the simulation time (via the keyword TIME), as well as the values of any variable or molecule count. The values of time, molecule counts and variables are evaluated at runtime and represent the current values during each iteration of the simulation. Presently, the following mathematical functions are supported: `sqrt, exp, log, log2, log10, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, erf, ergc, abs`.
Below is a list of all SGL blocks available for describing simulations. Presently, the order of blocks matters and should be exactly the same in which they are described below. Several SGL blocks are optional and are indicated as such below. This implies in particular that all non-optional blocks are required.
Currently, the SGL specs define the following block types with their respective block commands and block content:
parameter block: *<block name> = parameters*

The purpose of the parameter block is to describe the global simulation parameters. The following parameters are currently supported:

***time* = <*double*>** Maximum simulation time in seconds. Default is 0.0 s.

***outputBuffer* = <*Integer*>** Output will be kept in memory and written to the output file and stdout every *outputBuffer* iterations. Larger values should result in faster simulations but require more system memory. Default is to write output every 10000 iterations.

Note: *outputBuffer* only affects how often output is written to the output file, not how much is being accumulated during a simulation (see outputFreq parameter).

***outputFreq* = <*Integer*>** Frequency with which output is generated and written to the output file. Default is 1000.

***systemVol* = <*double*>** Volume of the simulation system in liters. This is needed to properly compute the reaction rates in molar units. If rates should rather be interpreted as reaction propensities (like in[1]) please set *systemVol = nil*. Default is a system volume of 1.0 liter.

***outputFile* = <*quoted string*>:** Name of the output file. This is the only required parameter in the parameter section. If not given, the simulation will terminate.

variable block: *<block name> = variables*

This block consist of a list of pairs of the form

```
<String> = variable expression
```

where `<String>` is the variable name, and `variable expression` is either a *Double* or an *expression statement* as defined above.

molecule block: *<block name> = molecules*

This block consist of a list of pairs of the form

```
<String> = <Integer>
```

giving the name of each molecule and the number of molecules present initially. For example, the following molecule definition block defines molecules `A` and `B` with initial numbers of 100 and 200, respectively

```
def molecules
  A = 100
  B = 200
end
```

NOTE: Please do not use any of the predefined mathematical functions or internal variables (currently only TIME) as molecule names since this will lead to undefined behaviour.

reaction block: *<block name> = reactions*

This block describes the reactions between molecules defined in the molecule block. Reactions are specified via

```
reactants -> product  | rate expression |
```

Here, `reactants` and `products` are of the form

```
<Integer> <String> + <Integer> <String> + .....
```

In this expression, `<String>` is a molecule name as defined in the molecule block and `<Integer>` an optional integer specifying the stoichiometry. If `<Integer>` is not explicitly given, it is assumed to be 1.

The `rate expression` can either be a fixed value of type *Double* or an *expression statement* as defined above.

Below is an example reaction block for the two molecules `A` and `B` defined above:

```
define reactions
  2A + B -> A  | 10.0e-5 |
  B      -> A  | {2.0e-5 * A * exp(-0.5*TIME)} |
end
```

In the first reaction, 2 `A` molecules react with one `B` to yield another `A` at a rate of 10.0e-5 1/(Mol s). The second reaction describes a decay of `B` back to `A` at a rate that is computed based on the instantaneous number of `A` molecules present and which decays exponentially with simulation time.

Internally, rate expressions are converted into a compute stack in RPN format which is evaluated at run-time. Even though this procedure is fairly efficient, there is some numerical overhead incurred at each iteration and the use of complicated rate expressions should therefore be avoided if possible.

event block: *<block name> = events*

An event block allows one to specify events which will occur during the simulation. Each event consists of a *trigger expression* and an associated set of *action exprssions.* Events are specified via

```
{ <trigger expression> } => { <action expression> }
```

Here, trigger expression“ is of the form

```
<trigger primitive> [ <boolean operator> <trigger primitive>]
```

with `<trigger primitive>` defined by

```
<expression statement> <relational operator> <expression statement>
```

Each `<trigger primitive>` contains two *expression statements* as defined above and a `<relational operator>` which can be any of `>=`, `<=`, `==`, `>`, and `<`. Hence, each `<trigger primitive>` evaluates to either *true* or *false.*

Several `<trigger primitives>` can be chained together via the `<boolean operators>` `&&` and `||` to yield a final boolean value of *true* or *false.*

If the `<trigger expression>` evaluates to true during an iteration, the associated `<action expressions>` is executed during the same timestep.

`<action expression>` consists of a semi-colon separated list of assignments

```
<String> = <expression> [; <String> = <expression>]
```

where `<String>` is a molecule or variable name and `<expression>` either a *Double* or an *expression statement.*

NOTE: Since molecule counts are integer values assignments to molecule counts in `<action expression>` will be converted to an integer value via *floor.*

output block: *<block name> = output*

This block consists of a simple list of variable and molecule names that will be streamed to the output file in the same order:

```
[ name1, name2, name3, .... ]
```

Example Input Files

Below are several example input files detailing the use of SGL:

- Lotka-Volterra Model
- Brusselator Model
- Oregonator Model

These are also available in the *Models/* sub-directory in the source tree.

Bugs

Please report all bugs and feature requests to <haskelladdict at users dot sourceforge dot net>.

References

[1]Daniel T. Gillespie (1977). "Exact Stochastic Simulation of Coupled Chemical Reactions". The Journal of Physical Chemistry 81 (25): 2340-2361

[2]http://hackage.haskell.org/packages/archive/random/1.0.0.1/doc/html/System-Random#globalrng.html

[3]Since **simgi** currently is an alpha version there may be fairly drastic changes to the SDL specs in future releases until the first beta release.

[4]An exception to this rule are line comments starting with `--` which ingnore everything until the next newline.

[5]Rate expressions can contain any arithmetic expression involving the standard operators "+", "-", "*", "/", "^" (exponentiation), and the mathematical functions `sqrt, exp, log, log2, log10, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, acosh, atanh, erf, erfc, abs`.