GraTS: graph transformation-based stochastic simulation — documentation

Paolo Torrini, Istvan Rath

December 11, 2011

1 Introduction

GraTS implements a framework to run discrete-event stochastic simulations of semi-Markov processes based on generalised stochastic graph transformation. Graph transformation rules are used to define observable events that lead from one state to another (action rules), as well as quantitative observations over states (probe rules). GraTS enables the execution of structured simulation experiments made of multiple batches of bounded runs, allowing for variations in sensitive parameters that may include maximum length. It can compute statistics about action rule application, timing, probe rule values and covariance matrix, based on different forms of aggregation of the observations throughout an experiment.

GraTS is implemented in Java as a functionality of VIATRA2, a rule-based language interpreter based on abstract state machines and incremental pattern matching, available as Eclipse plugin. GraTS relies on the implementation of graph transformation given by VIATRA2, taking advantage of incremental pattern matching to deal efficiently with semi-Markov processes. GraTS uses the SSJ libraries to deal with random number generation, distribution functions, confidence intervals and other statistics. GraTS also allows for the visualisation of individual runs, based on graph visualisation as provided in Eclipse by Zest.

1.1 Models

Models are generalised graph transformation systems. They can represent discrete event systems, where each state is a graph of given type, and each event is given as a transformation rule matched to a subgraph of the current state. Each event has a sort, and it is associated to a random variable that represents the scheduled delay as specified by a continuous cumulative distribution function (cdf) assigned to the sort.

Each model is put in as a VIATRA2 graph transformation system together with a cdf assignment to event sorts. By default, the sort of an event is the same as the associated rule name — but this can be modified, by considering attributes and therefore making cdf assignment depend on local values.

Statistics are computed by using SSJ methods. In particular, we use the Report method to compute average, maximum, minimum and standard deviation (report values) for each observation which is stored as an instance of the Tally class. Statistics may refer to the following possible observations.

1. for each state, number of matches for each probe rule
2. for each run, application and timing for each action rules (i.e. number of applications divided by the run length, and average delay)
3. variables obtained by aggregation of report values based on batch (set of runs with same parameters), slice (set of runs with varying parameters) and experiment (set of batches)
4. confidence intervals for mean and variance
5. correlation between pairs of observations

The underlying stochastic model is that of a semi-Markov process allowing for exponential and lognormal distributions (lognormal instead of normal, as all our variables represent time delays and thus we need only positive values).

2 Input requirements

Running a simulation requires the following

1. a GTS
   (a) the metamodel
   (b) the initial model
   (c) the transformation rules, as a *vtcl* file

   The model should be contained in a dedicated element of the model space
2. setting the execution parameters — these include
   (a) external parameters, as passed by the model space
   (b) internal parameters, as passed by an XML file (parameters.xml)
3. setting the stochastic structure, as defined in an XML file (stochastic input file) — this includes
   (a) stochastic control types for the transformation rules, which define how to use the rules with respect to transformation and probing
   (b) event sorts associated to action rules — as subsets of rule matches
   (c) assignments of cumulative distributions functions to the action rules and (optional) to the associated event sorts
   (d) stochastic variation control values (optional)

3 Simulation experiments

A simulation experiment consists of a set of runs, each specified by a given stochastic assignments, and executed up to a given maximum length. Each experiment is structured as a sequence of variations, each variation defined by a batch of simulation run. All the runs of the same batch share the same stochastic parameters. Runs from different batches can vary, either in stochastic assignment or in maximum length.

Under the hierarchical aggregation option, statistics are printed for each run, batch and experiment. At each level, observations are printed out as comma-separated values, statistics as JSS reports and covariance analysis output. Under the 2D-linear aggregation option, statistics are aggregated for batches (horizontally) and slices (vertically) after execution. Aggregation is exhaustive for runs of the same batch, arbitrary for slices. A slice is defined as a set of runs, each from a different batch, so to have one for each variation, and never the same in two different slices.

4 Rules and events

Each transformation rule should be assigned a type - these are represented alphabetically as ordered strings made of

1. 'A' for applicable rule (action)
2. 'S' for sensitive action
3. 'O' for observable action
4. 'P' for probe rule

such that if either 'O' or 'S' occurs, also 'A' does. In other terms, the letters are primitives, 'S' and 'O' are subtypes of 'A', and each string represents an intersection.

Event sorts are represented as integers, with 0 as default. In the basic implementation, based on class *ETuple1*, the default event type is the only one. Shifting to *ETuple2*, it is possible to introduce more event sorts, to be associated to rule matches on the basis of designated attributes.

For each action rule, each event sort should be assigned a *cdf*, as follows

1. exponential distribution
   type: *exp* (String)
   numeric parameter: *rate* (Double precision)
2. lognormal distribution
   type: *norm* (String)
   numeric parameters: *mean*, *variance* (Double precision)

Action rules are used to be scheduled for execution, on the basis of the *cdf* associated to their event types. In the case of sensitive rules, the *cdf* can vary through different batches. This can happen automatically — by multiplying the initial parameters by a power of a base value up to a limit, or else sequentially by each of the variation values given as input manually in the stochastic input file. If rules are observable, the simulator returns statistics about their application (listed as 'APP:rulename'), and about the delay with which they have been applied over each run (listed as 'OBS:rulename'), both printed to the batch report files.

If a rule is a probe rule, the simulator will collect the number of its matches at each step. These statistics can be printed to the batch report files by defining probes that refer to the rules.

5 Stochastic input file

The stochastic input file should include

1. a set of stochastic assignments
2. probe definitions
3. a set of variation values

A rule set (tag *ruleset*) is given by a stochastic assignment together with a set of probe definitions — more precisely

1. an assignment of control types to transformation rules (tag *rule*) from the *vtcl* file
2. an assignment of *cdfs* to the event sorts (tag *event*) associated with action rules in the rule set
3. a list of probe definitions

Each probe (tag *probe*) is defined in terms of an operator (tag *op*) which may either be a probe rule name, or else be an operation applied to arguments (tag *arg*) which are probe rule names. The operations that are currently supported are *sum* (sum of two arguments), *div* (division of first argument by the second), and *divsq* (division of first argument by the square of the second). Notice that the position of the argument is given explicitly as an integer value (tag *pos*). The simulator adds automatically to the probe list *TIME*, recording the application times (in terms of simulation time — independently of any real time interpretation), and *DELAY*, recording the delays associated to the applied matches. No user-given probe name should be "TIME", "DELAY", or start with either "OBS:" or "APP:".

Each variation value (tag *variation*) is given as a double precision value. *allvars* and *probeset* are optional tags.

6 Output

There are three kinds of output that are printed out. Observation samples are printed out as lists of comma-separated values. Statistics over samples are computed and printed out as SSJ reports. These include average, maximum, minimum and standard deviation, and moreover, the student-T and normal confidence intervals for the mean, and the Chi-2 confidence interval for the variance, with respect to a given confidence level. In the case of hierarchical aggregation, output is provided at the level of runs, batches and experiment. In the case of 2D-linear aggregation analysis, CSV output is given at the level of runs, and reports are given at the level of batches and slices. General information about the experiment settings and run-time values in milliseconds are printed out to a separate file. Moreover, different levels of debugging information are printed to the Eclipse standard output.

7 Execution parameters

7.1 External parameters

The external parameters are stored in the *StoSimPars* element of the model space. *StoSimPars* need to be created as a child of the root.

The parameters are

1. *Machine* — the VIATRA *machineFQN* parameter, that can be derived from the *namespace* and *machine* declarations included in the VTCL file. E.g. given the declarations

   ```
   namespace n; machine m;
   ```

   the value of *machineFQN* is then *n.m*

2. *ModelPath* — the qualified name of the model space element which that is the target for the simulation. The dedicated model element should be a child of *ModelPath* (and therefore, *ModelPath* is also the relative path to the dedicated model element). Notice that at initialisation the simulator will copy the internal parameters and the stochastic parameters from the XML files to subelements of *ModelPath* (*parameters* and *distributions*, respectively) — unless *extInputOPtion* is set to *false*

3. *ioPath* — the absolute path to the folder that contains the specification of the GTS

4. *ioInputFolder* — the subfolder of *ioPath* where the input XML files are to be found

5. *extInputOption* - Boolean value (as a string) which decides whether the internal parameters are to be read from the XML file or to be found already in the model space

7.2 Internal parameters

If not already in the model space, these parameters should be found in an existing XML file with name *parameters.xml* located in *ioPath.ioInputFolder*. They can be divided into five groups.

7.2.1 IO parameters

1. *CDF_input* (string) — the name of the stochastic input file, to be found in *ioPath.ioInputFolder*
2. *Output_Folder* (string) — the name for the subfolder of *ioPath* where the output log files are printed out
3. *Output_Files* (string) — name prefix for the log files where the output is printed out

7.2.2 Run parameters

1. *Rule_Set* (string) — the name of the rule set (among those included in the stochastic input file)
2. *Batch_Size* (integer as string) — number of runs for each batch
3. *Time_Opt* (Boolean as string) — if true, each run is limited by a maximum simulation time, otherwise it is limited by a maximum number of steps (depth)
4. *Time_Limit* (integer as string) — maximum number of steps (converted to double precision and used if *Time_Opt* is set to *true*)
5. *Depth_Limit* (integer as string) — maximum number of steps (used if *Time_Opt* is set to *false*)

7.2.3 Variation parameters

1. *Variations* (integer as string) — number of the variations, i.e. of the batches executed for the experiment (gives also the size of each slice)
2. *Variation_Type* (character as string, *N*|*S*, *S* as default) — set to *N*, the variations are in the maximum length of the run (increasing by a factor of 10 for each batch); otherwise, the variations are in stochastic assignment of the sensitive rules (i.e. in 'speed'); notice that if set to speed with no sensitive rules, there is no change in the batch settings
3. *Factor_Up_Opt* (Boolean as string) — only used with speed variations; if set to *true*, the speed variations are obtained by multiplying the initial stochastic parameters of the sensitive rules by a power of *Factor_Up_Base* (integer as string); if set to *false*, the variation values are read from the

stochastic input file (where they are expected to match the variation number)

7.2.4 Output parameters

1. *Hierarchical_Aggregation* (string) — set to *full* prints out the full hierarchical aggregation analysis, which includes covariance and correlation, set to *short* prints out the full analysis for the first two levels (runs and batches) and a selection of the experiment level output

2. *2D_Linear_Aggregation* (string) — set to *basic* prints out only a general analysis on time and depth, set to *batch* prints out batch reports, set to *slice* prints out slice reports, set to *full* prints out the full 2D-linear aggregation analysis

3. *Debug_Level* (integer as string) — sets the amount of debug information printed out to the Eclipse console. It can be set to 0 (no debug except at initialisation), 1 (prints out the applied rules), 2 (prints out info about matches - very slow), 3 (prints out more internal representations)

4. *Confidence_Level* (double precision as string - positive and less than 1) for the confidence intervals

7.2.5 Extra parameters

1. *Debug_Level* (integer as string) — sets the amount of debug information printed out to the Eclipse console. It can be set to 0 (no debug except at initialisation), 1 (prints out the applied rules), or 2 (prints out info about matches - extremely slow)

2. *Covariance_Opt* (Boolean as string) — sets whether the covariance analysis is carried out

3. *Feedback_Opt* (Boolean as string) — if set to *true*, the simulator will look at each step for an element named as *Clock_Name* in the model, and set its value to that of the current simulation time

4. *Clock_Name* (string) — used with *Feedback_Opt*

5. *Extra_Attribue* (string) — when the simulator sources are recompiled with *ETuple2* rather than default *ETuple1*, it is possible to select an attribute *X* with integer values, such that for each match, the simulation will use the values of the instances of *X* included in the match to determine the event sort using *Extra_Operator*

6. *Extra_Operator* (string) — either *min* or *max* (the latter by default)

8 Installation from SVN

Requirements: Java 1.6.x, Eclipse, VIATRA2. The SVN repository is located at

```
http://viatra.inf.mit.bme.hu/svn
```

Check out everything from

```
https://viatra.inf.mit.bme.hu/svn/core/trunk/
```

```
https://viatra.inf.mit.bme.hu/svn/stochsim/trunk/
```

Simply delete what gives conflict. In special cases, one might also need addons such as ViatraDSM (`https://viatra.inf.mit.bme.hu/svn/dsm/trunk/`), or others — these are easily found inside the main SVN hierarchy by name. Run configurations and switch to the VIATRA perspective in the run-time framework. Simulation project samples can then be checked out from

```
https://viatra.inf.mit.bme.hu/svn/transformations/
    collab/trunk/stochastic_gt
```

9 Running the simulation

Launch a run-time framework, open the VIATRA perspective, load the model and the rules. The model space should include the external control parameters. The model should be in the dedicated element, and there should be a folder with the XML input files as specified by the parameters.

Check that the VIATRA textual output is not buffered (green toggle button to the top right of the window). Right-click on the model icon in the model space window, and select *Run stochastic simulation*.

10 Visualisation

It is possible to visualise the simulation, while running it in default automatic mode, or else while executing a single run interactively, step-wise. It is also possible to customise the visualisation (by filtering elements and changing their look).

The following steps are needed in order to set the visualisation environment.

1. activate the visualisation for the dedicated model element subtree by right-clicking on it and selecting *Add subtree*
2. you have the option of using a domain-specific visualiser module, by selecting the drop-down menu of the VIATRA visualisation view (small triangle towards the right on the menu bar of the *_view_*, picking *Layout*,

and finally the *stoch-sim* layout). You can revert to the standard layout using the same menu. More details on domain-specific visualisation can be found at

`http://viatra.inf.mit.bme.hu/publications/traceviz`

If you are interested in customizing the visualisation layout yourself, you can do it by changing the class:

`https://viatra.inf.mit.bme.hu/svn/stochsim/`

`trunk/org.eclipse.viatra2.gui.stochsim/src/org/`

`eclipse/viatra2/gui/stochsim/viz/StochSimFilter.java`

3. to start the interactive mode, right-click on the model icon and choose the *setup interactive stochsim* option. The Interactive stochsim control view will pop up, with the corresponding machine selected.

11 Command-line version

It is possible to build a version of the simulator that can be run by command-line. In order to obtain it, the following steps should be taken (only with Linux and Eclipse-Galileo)

1. open the sscons2.product file in console.stochsim, and in the editor locate the option *Exporting/link* to *Eclipse Product Export Wizard*

2. in the opening dialog, 2a. enter a name for the root directory if it isn't already entered (e.g. *stochsim*) 2b. select a destination directory where the binaries will be deployed 2c. export options: only the *allow for binary cycles* option should be checked (not sure whether this always shows) 4. click finish 5. if everything goes well, in the directory specified a so-called headless version will be deployed along with an executable that can be run e.g. like this:

   `./stochsim LOC/mymodel.vpml LOC/myrules.vtcl mymodel.myrules`

   The three mandatory parameters are:

   1. VPML file (FQN), 2. VTCL file (FQN), 3. the *machineFQN* parameter, to be passed to the simulator (this will also reset the *StoSimPars.Machine* external parameter).

   The *StoSimPars.extInputOption* will be reset to true — therefore, it will read the internal parameters from the XML file. The remaining external control parameters are those set in the model space when you run the export wizard. Alternatively, one can pass them as additional parameters from command line, in the following order:

4. *ioInputFolder* 5. *ioPath* 6. *modelPath*

In practice, only the first one can be useful, if you keep different XML files in different locations.