Stochastic Simulation of Graph Transformation Systems

Paolo Torrini[1], Reiko Heckel[1], and István Ráth[2]

[1] Department of Computer Science, University of Leicester, pt95 | reiko@mcs.le.ac.uk
[2] Department of Measurement and Information Systems
Budapest University of Technology and Economics, rath@mit.bme.hu

1 Introduction

Stochastic graph transformation systems (SGTS) [1] support integrated modelling of architectural reconfiguration and non-functional aspects such as performance and reliability. In its simplest form a SGTS is a graph transformation system (GTS) where each rule name is associated with a rate of an exponential distribution governing the delay of its application. However, this approach has its limitations. Model checking with explicit states does not scale well to models with large state space. Since performance and reliability properties often depend on the behaviour of large populations of entities (network nodes, processes, services, etc.), this limitation is significant. Also, exponential distributions do not always provide the best abstraction. For example, the time it takes to make a phone call or transmit a message is more likely to follow a normal distribution.

To counter these limitations, *generalised* SGTS [2] allow for general distributions dependent on rule - match pairs (rather than just rule names). Generalised semi-Markov processes provide a semantic model for such systems, supporting stochastic simulation. Rather than model checking, simulations provide a more flexible tradeoff between analysis effort and confidence in the result and so allow to verify soft performance targets in large-scale systems.

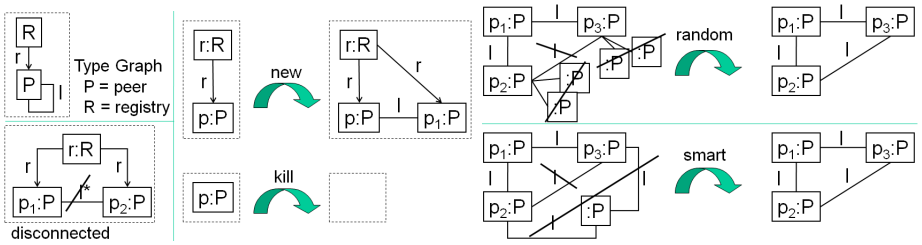
We present a tool called GraSS, for Graph-based Stochastic Simulation, to enable the analysis of such processes. The tool is developed in Java-Eclipse, extending the VIATRA model transformation plugin with a control based on the SSJ library for Stochastic Simulation in Java. The main performance challenge, in finding, at each state of the simulation, all matches for all rules, is alleviated by VIATRA's RETE-style incremental pattern-matching approach [3], which stores precomputed matching information and updates it during transformation. We illustrate and evaluate the application of the tool by the simulation of the original P2P reconfiguration model as well as an improved and scaled-up version.

2 A P2P Network Model

As a test case we use an example of a SGTS modelling reconfigurations in a P2P network [1]. Generating the state space of the model for up to seven peers, in [1] we used stochastic model checking to analyse, e.g., the probability of the

network being fully connected, so that each participant can communicate with every other one.

The GTS below models basic P2P network reconfigurations. Rule *new* on the left adds a new peer, registers it and links it to an existing peer. Rule *kill* deletes a peer with all links attached. Predicate *disconnected* checks if there are two nodes that are not connected by a path of links labelled l.



The two rules on the right create redundant links to increase reliability in case a peer is lost. Rule *random* creates a link between $p2$ and $p3$ unless there is one already or the number of additional connections of either $p2$ or $p3$ is greater than two. Rule *smart* creates a link if there is no two-hop path between $p2$ and $p3$ apart from the one via $p1$. We consider two families of systems, $SGTS_{random,x}$ and $SGTS_{smart,x}$. The former has rules $\{new, kill, random\}$ and rates $\sigma(new) = \sigma(kill) = 1$ and $\sigma(random) = x$. In the latter, *random* is replaced by *smart* with $\sigma(smart) = x$. In both cases x ranges from 1 to 10,000 to test different ratios between basic and redundancy rules.

3 Simulating Stochastic Graph Transformations

In order to define a general interface between the stochastic control component of the simulation and existing graph transformation tools used for executing rules, we define SGTS for a generic notion of graph transformation. Refining [4], a *graph transformation approach* is given by a class of graphs $\mathcal{G}$, a class of rules $\mathcal{R}$, and a $\mathcal{R} \times \mathcal{G}$-indexed family of sets of *rule matches* $\mathcal{M}_{r,G}$ for rule r into graph G. Transformation is defined by a family of partial functions $\Rightarrow_{r,m}: \mathcal{G} \to \mathcal{G}$, such that $\Rightarrow_{r,m}(G)$ is defined if and only if $m \in \mathcal{M}_{r,G}$. This captures the idea that rule application is well-defined and deterministic if m is a match for r in G.

For a set of rules R, $\mathcal{E}_R$ is the set of *events*, i.e., compatible pairs $\langle r, m \rangle$. $\mathcal{S} = \langle R, G_0, F \rangle$ is a *stochastic graph transformation system* with set of rules R, initial graph G_0, and $F : \mathcal{E}_R \to (\mathbf{R} \to [0, 1])$ assigning each event a continuous distribution function such that $F(e)(0) = 0$.

We encode SGTS into generalised semi-Markov schemes (GSMS), a generalisation of Markov chains associated with generalised semi-Markov processes [5]. Here transitions are independent of past states, but unlike Markov chains they may depend on the time spent in the current one, i.e., interevent times may be non-exponentially distributed. Formally, a GSMS is a structure

$$\mathcal{P} = \langle\, S,\ E,\ act : S \to \wp(E),\ trans : S \times E \to S,\ \delta : E \to (\mathcal{R} \to [0, 1]),\ init : S \,\rangle$$

where S is a set of states (given by all graphs reachable in $\mathcal{S}$), E is a set of events (the rule matches $\mathcal{E}_R$), $init$ is the initial state (graph G_0), act gives the set of events (rule matches) enabled in a state (graph), $trans$ is the transition function (given by $trans(G, \langle r, m\rangle) = \Rightarrow_{r,m} (G)$), and δ defines the probability distribution for each event (given by F).

The simulation component uses VIATRA as a graph transformation tool to implement the elements of the GSMS that depend on the representation of states and events, notably $S, E, act, trans, init$, i.e., GTSs are represented as a VIATRA models. Definitions of distributions F are loaded from an XML file. Based on this data, a GSMS simulation in GraSS consists of the following steps

1. Initialisation — the simulation time T is initialised to 0 and the set of the enabled matches (active events) is obtained from the graph transformation engine. For each active event, a scheduling time t_e is computed by a random number generator (RNG) based on the probability distribution assigned to the event. Timed events are collected as a list ordered by time (state list).
2. At each simulation step
   (a) the first element $k = (e, t)$ is removed from the state list
   (b) the simulation time is increased to t
   (c) the event e is executed by the graph transformation engine
   (d) the new state list s' is computed, by querying the engine, removing all the elements that have been disabled, adding to the list an event for each newly enabled match m with time $t = T + d$, where d is provided by the RNG depending on $F(m)$, and reordering the list with respect to time

GT rules with empty postconditions are used as probes — statistics about occurrence of precodition patterns are computed as SSJ tally class reports, giving average values over runs. One can specify the number of runs per experiment (esp. useful to reduce the biasing effect of runs truncated by deletion of all elements) and their max depth (either by number of steps or simulation time).

4 Evaluation

In order to validate the correctness and scalability of the tool we run a number of experiments based on the P2P model of Section 2. We do not expect to replicate exactly the results reported in [1] because (1) we remove the restriction to 7 nodes that was used to guarantee a finite (and manageable) state space; (2) unlike in [1] where states and transitions were presented up to isomorphism, our simulation deals with concrete graphs and transitions. A detailed comparison of the underlying mathematical models is beyond the scope of this paper, but it appears that, since the Markov chain is constructed from a more abstract transition system in [1], the two are not in stochastic bisimulation. Thus, evaluating the same properties on both models may lead to different results. As in [1] we run experiments with 10 different models, 5 versions each of using *random* and *smart* rules, with rates ranging through $x \in \{1, 10, 100, 1000, 10000\}$.

We perform 5 runs each with a simulation time bound of 10s for each experiment — i.e. no run exceeds 10s regardless of the number of steps. The table below gives the output of an experiment, indicating the version of the model (1st column) followed by the percentage of disconnected states encountered, the average number of steps performed per run, the average maximal extension of the network, and the average time taken for each run.

Model: P2P	Disconnected	Number of steps	Max number of peers	Runtime
random:1	0.46	33	6	5
random:10	0.62	71	8	8
random:100	0.55	86	8	7
random:1000	0.89	284	20	10
random:10,000	0.46	116	8	9
smart:1	1.33	18	5	1
smart:10	0.01	90	8	4
smart:100	0.00	3561	48	10
smart:1000	0.00	998	24	10
smart:10,000	0.00	62	8	3

Such results confirm the inverse dependency observed in [1] between the rate of the *smart* rule and the probability of being disconnected, whereas for the *random* rule an increased rate does not lead to any significant change in reliability — as confirmed by the average number of disconnections modulo square of node number (not shown). The performance (number of simulation steps per sec) is limited by the complexity of pattern *disconnect* which, in a network of n peers, checks for (non-) existence of n^2 paths. This can be hard due to transitive closure. As a simpler reliability measure, the proportion of peers with at least two connections (hence less vulnerable to loss of connectivity) can do. A simulation of 5 runs with a time limit of 10s has always been carried out in less than a minute. Reliance on incremental pattern matching means model size only affects simulation up to number of RNG calls, whereas increase in number and complexity of the rules can add to the cost of graph transformation, too.

References

1. Heckel, R.: Stochastic analysis of graph transformation systems: A case study in P2P networks. In Van, H.D., Wirsing, M., eds.: ICTAC'05. Volume 3722 of LNCS., Springer (2005) 53–69
2. Khan, A., Torrini, P., Heckel, R.: Model-based simulation of VoIP network reconfigurations using graph transformation systems. In Corradini, A., Tuosto, E., eds.: ICGT'08 - Doctoral Symposium. Volume 16 of El. Com. EASST. (2009) 1–20
3. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the Viatra model transformation system. In: GRaMoT '08, ACM (2008) 25–32
4. Kreowski, H.J., Kuske, S.: On the interleaving semantics of transformation units - a step into GRACE. In: 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073, Springer (1996) 89 – 106
5. D'Argenio, P.R., Katoen, J.P.: A theory of stochastic systems part I: Stochastic automata. Inf. Comput. **203**(1) (2005) 1–38

A VIATRA Model of the P2P Network

We present the VIATRA implementation of our graph transformation rules shown in Section 2. The first pattern implements a check for a bidirectional association.

```
pattern connected(N1,N2) = {
               SN(N1);
               SN(N2);
               SN.overlay(Ov1,N1,N2);
               SN.overlay(Ov2,N2,N1);
        }
```

The following is a recursive pattern checking for the existence of a path between $N1$ and $N2$. Rule *disconnected* is used as statistical probe, to check whether there are two disconnected nodes. This rule is never applied, the simulation just checks whether it is applicable (and counts its matches).

```
pattern pathEx(N1,N2) = {
                SN(N1);
                SN(N2);
                find connected(N1,N2);
   }
       or
       {
               SN(N1);
              SN(N2);
              SN(N0);
              find connected(N1,N0);
              find pathEx(N0,N2);
       }

pattern noPathEx(N1,N2) =
{
SN(N1);
                SN(N2);
                neg find pathEx(N1, N2);
}

gtrule disconnected() =
        {
            precondition pattern lhs(N1, N2) =
            {
             SN(N1);
             SN(N2);
             RS(R1);
```

```
            RS.reg(Rs1,R1,N1);
            RS.reg(Rs2,R1,N2);
            find noPathEx(N1, N2);
           }

    action {
    println("Found a disconnection: "+fqn(N1)+fqn(N2));
    }
        }
```

Here follows the rule for creating new peers, registering them, and connecting them to existing peers in the network

```
gtrule newNode() =  {
             precondition pattern lhs(N1,R1) =  {
                 SN(N1);
                 RS(R1);
                 RS.reg(Rs1,R1,N1);
             }

             action {
                let Ov1=undef, Ov2=undef, Rs2=undef, N2=undef in seq {
                    new(SN(N2));
                    new(SN.overlay(Ov1,N1,N2));
                    new(SN.overlay(Ov2,N2,N1));
                    new(RS.reg(Rs2,R1,N2));
                    println("created node: "+fqn(N2));
                }
             }
       }
```

. . . and the one for deleting a peer with all its connections.

```
gtrule killNode() =     {
            precondition pattern lhs(N1) =  {
                SN(N1);
            }

            action {
                delete(N1);
                println("node to be deleted: "+fqn(N1));
            }
        }
```

The *random* rule creates redundant links up to a limit of 3 links per peer.

```
gtrule randomConnect() =    {
```

```
            precondition pattern lhs(N0,N1,N2) =   {
                SN(N0);
                SN(N1);
                SN(N2);
                find connected(N0,N1);
                find connected(N0,N2);
                neg find connected(N1,N2);
                neg find two(N1);
                neg find two(N2);
            }


            action {
                let Ov1 = undef in new(SN.overlay(Ov1,N1,N2));
                let Ov2 = undef in new(SN.overlay(Ov2,N2,N1));
                println("added connection between: "
                           +fqn(N1)+fqn(N2));
            }
        }
```

The *smart* rule is more cautious, creating a link only if there is but a single two-hop path between the two peers. The auxiliary pattern *twoConnect* checks if there are two.

```
pattern twoConnected(N1,N2) =  {
               SN(N1);
               SN(N2);
               find connected(N1,N0);
               find connected(N2,N0);
               find connected(N1,N3);
               find connected(N2,N3);
               check (N0!=N3);
        }


gtrule smartConnect() = {
            precondition pattern lhs(N0,N1,N2) =   {
                SN(N0);
                SN(N1);
                SN(N2);
                find connected(N0,N1);
                find connected(N0,N2);
                neg find connected(N1,N2);
                neg find twoConnected(N1,N2);
            }
```

```
        action {
            let Ov1 = undef in new(SN.overlay(Ov1,N1,N2));
            let Ov2 = undef in new(SN.overlay(Ov2,N2,N1));
            println("added connection between: "
                       +fqn(N1)+fqn(N2));
        }
    }
```

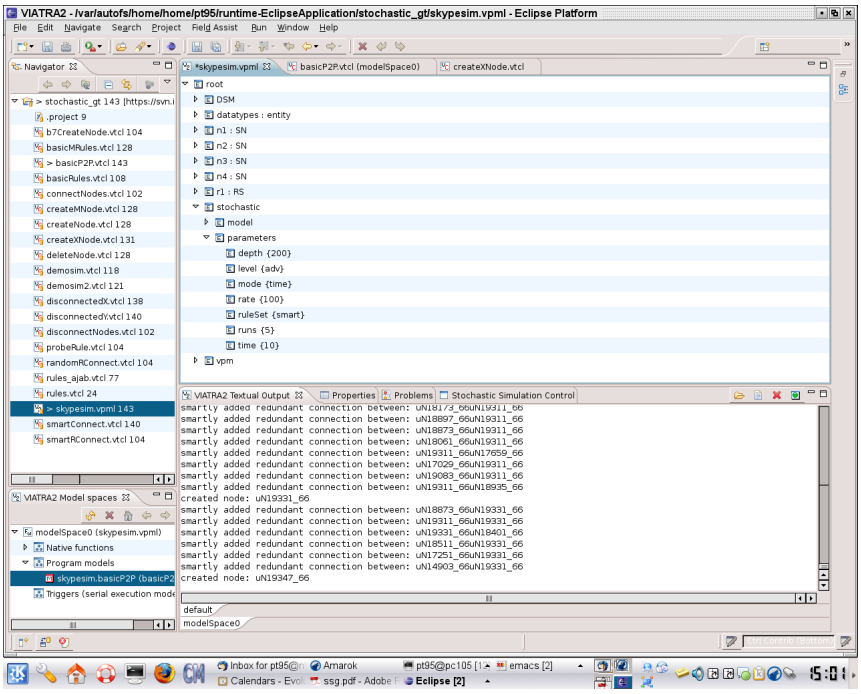
B Eclipse Plugin

Like VIATRA itself, the simulation tool is implemented as an Eclipse plugin which is invoked from within Eclipse itself. There is no separate user interface, all input and output is via files and VIATRA models.

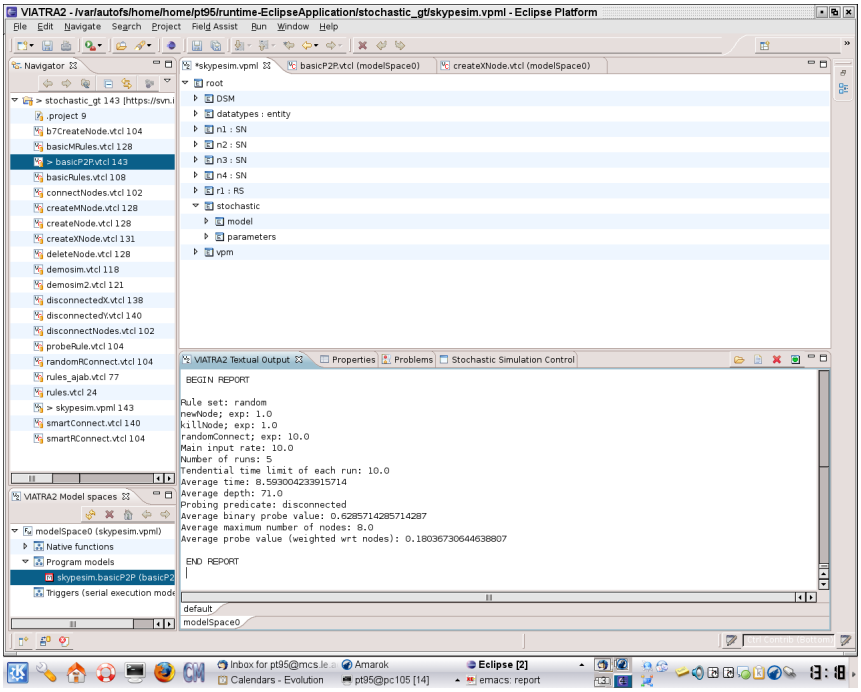
Probability distributions are loaded from an external source, which allows to specify the type of distribution (exponential, normal, etc.) as well as its parameters (rate, mean and variance, respectively).

The demonstration will show the model, the uploading of distributions sources and feature parameters, start a simulation and show the output on screen as well as collected in a log file. To give an indication, below we include a screen of the invocation of the simulation as well as a sample log.

The snapshots below shows a VIATRA screen with the initial model. Model entities are also used to pass parameters to the simulation — notably, number of runs, limit in either time or number of steps, the name of the rule set (with associated stochastic parameters), and a distinguished rate value. A run is started by clicking on the rule icon in the model space window (left below).



The second snapshot shows the output of a system run in the VIATRA textual output window, where also the printout from rule application appears at runtime.


VIATRA2 - /var/autofs/home/home/pt95/runtime-EclipseApplication/stochastic_gt/skypesim.vpml - Eclipse Platform
File Edit Navigate Search Project Field Assist Run Window Help
Navigator
> stochastic_gt 143 [https://svn.i
.project 9
b7CreateNode.vtcl 104
basicMRules.vtcl 128
> basicP2P.vtcl 143
basicRules.vtcl 108
connectNodes.vtcl 102
createMNode.vtcl 128
createNode.vtcl 128
createXNode.vtcl 131
deleteNode.vtcl 128
demosim.vtcl 118
demosim2.vtcl 121
disconnectedX.vtcl 138
disconnectedY.vtcl 140
disconnectNodes.vtcl 102
probeRule.vtcl 104
randomRConnect.vtcl 104
rules_ajab.vtcl 77
rules.vtcl 24
> skypesim.vpml 143
smartConnect.vtcl 140
smartRConnect.vtcl 104
VIATRA2 Model spaces
modelSpace0 (skypesim.vpml)
Native functions
Program models
skypesim.basicP2P (basicP2
Triggers (serial execution mode
*skypesim.vpml
basicP2P.vtcl (modelSpace0)
createXNode.vtcl (modelSpace0)
root
DSM
datatypes : entity
n1 : SN
n2 : SN
n3 : SN
n4 : SN
r1 : RS
stochastic
model
parameters
vpm
VIATRA2 Textual Output
Properties
Problems
Stochastic Simulation Control
BEGIN REPORT
Rule set: random
newNode; exp: 1.0
killNode; exp: 1.0
randomConnect; exp: 10.0
Main input rate: 10.0
Number of runs: 5
Tendential time limit of each run: 10.0
Average time: 8.593004233915714
Average depth: 71.0
Probing predicate: disconnected
Average binary probe value: 0.6285714285714287
Average maximum number of nodes: 8.0
Average probe value (weighted wrt nodes): 0.18036730644638807
END REPORT
default
modelSpace0
Inbox for pt95@mcs.le
Amarok
Eclipse [2]
Calendars - Evolution
pt95@pc105 [14]
emacs: report