Quick introduction to PRISM GSMP *(assumes basic knowledge of PRISM)*

Mário Uhrík
November 26, 2018

1 What is PRISM GSMP and what does it offer?

PRISM GSMP is an extension of PRISM model checker that adds modeling and analysis using Generalized Semi-Markov Processes (GSMP). GSMPs may be seen as CTMCs with any number of concurrently active state-changing events at any given time. These events may have any general distribution on time of occurrence, instead of just exponential distribution as is the case of CTMC. Thus, the modeling capability of GSMP is far greater than that of CTMC.

Precisely, PRISM GSMP adds the following:

- Extension of PRISM modeling language that adds support for GSMP. For now, the following types of distributions are supported:
  - Exponential distribution with real rate parameter $\lambda > 0$
  - Dirac distribution with real timeout parameter $t > 0$
  - Uniform distribution with real parameters $b > a > 0$
  - Erlang distribution with real rate and integer shape parameters $\lambda, k > 0$
  - Weibull distribution with real shape and scale parameters $\lambda, k > 0$
- Analysis of continuous-time Markov chains with alarms (ACTMCs), a subset of GSMPs. ACTMC is a GSMP with only up to one non-exponentially distributed event active in any given state. With this restraint, using the method of subordinated Markov chains, PRISM GSMP supports the following kinds of analysis:
  - Steady-state probabilities for ACTMC
  - Steady-state rewards for ACTMC

  - Reachability rewards for ACTMC

- Optimal parameter synthesis for minimizing/maximizing the reachability reward for ACTMC. E.g. given an ACTMC, find the optimal distribution parameter of a particular event such that the reachability reward is maximized. This is implemented using efficient algorithms recently published in the PhD thesis of Ľuboš Korenčiak.

Formal definition of GSMP, syntax and semantics of the language extension can be found at `https://github.com/VojtechRehak/prism-gsmp/blob/master/doc/prism-gsmp-property.pdf`. Example GSMP models written in this language extension can be found at `https://github.com/VojtechRehak/prism-gsmp/tree/master/prism-examples/gsmp`.

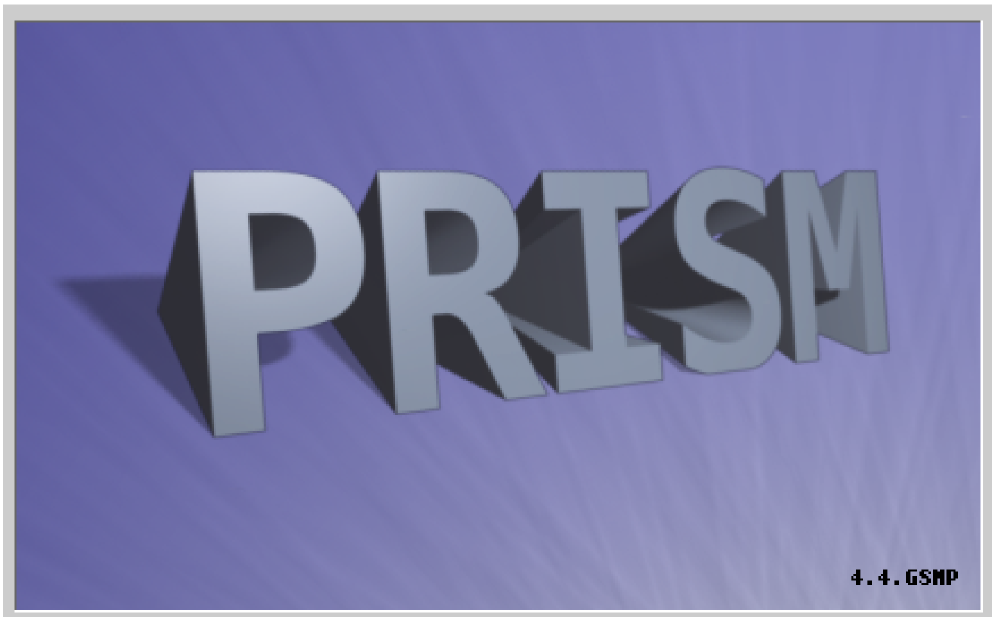
2 Obtaining, installing, and running PRISM GSMP

Installation of PRISM GSMP is identical to the process described in detail on the official PRISM website under **"Building PRISM from source on Windows using Cygwin"** for Windows, or **"Building PRISM from source (non-Windows)"** for other operating systems.

Essentially, this boils down to the following:

1. Download repository `https://github.com/VojtechRehak/prism-gsmp`.
2. Successfully run *make* in directory *prism-gsmp/prism*.
3. PRISM GSMP can now be launched by running the scripts in *prism-gsmp/prism/bin*. PRISM GSMP fully supports both the command line version (*prism*) and the GUI version (*xprism*).

Verify that it is indeed PRISM GSMP by observing that the version format is *X.Y.GSMP*.



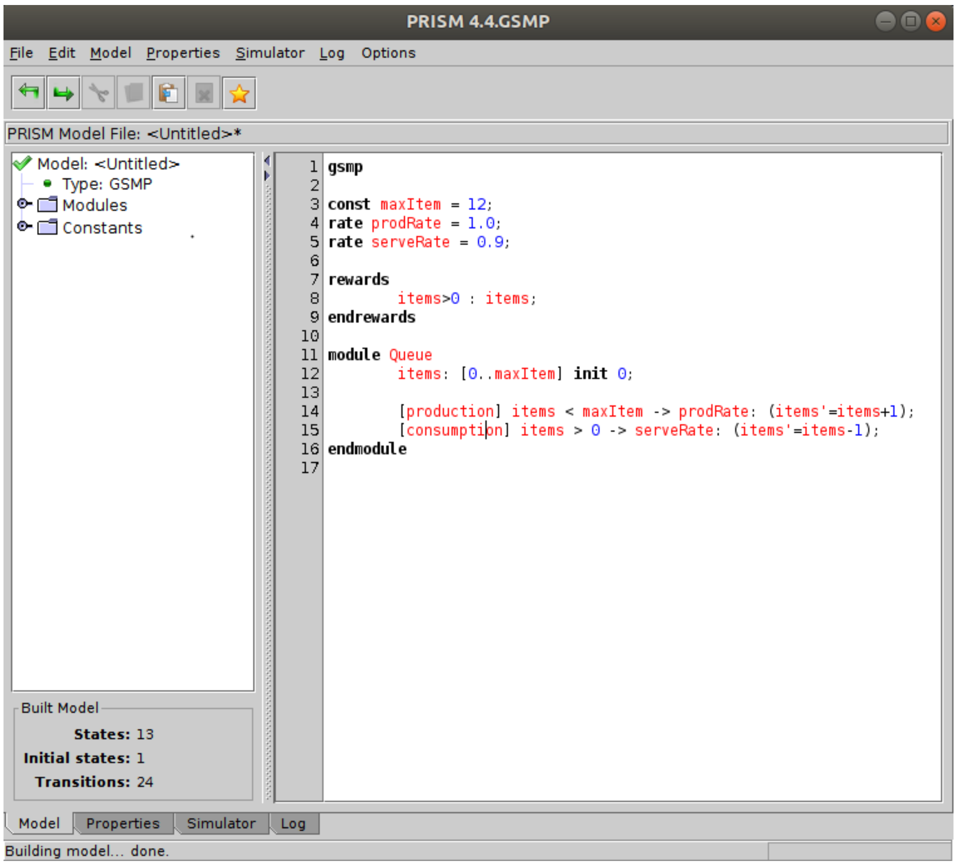
3 GSMP MODELING MINITUTORIAL

First of all, ensure that engine is set to *explicit* all times. PRISM GSMP is only implemented for the *explicit* engine. In PRISM GSMP, *explicit* engine is the default setting.

GSMP modeling language has full backward compatibility to CTMC models, i.e. all valid CTMC models in the PRISM language can be parsed as GSMP. Let us demonstrate on the following CTMC model of an $M/M/1/n$ queue.

```
ctmc

const maxItem = 12;
rate prodRate = 1.0;
rate serveRate = 0.9;

rewards
   items>0 : items;
endrewards

module Queue
   items: [0..maxItem] init 0;

   [production] items < maxItem -> prodRate: (items'=items+1);
   [consumption] items > 0 -> serveRate: (items'=items-1);
endmodule
```

Load this model into PRISM GSMP, but use keyword *gsmp* instead. Build the model.



```
gsmp

const maxItem = 12;
rate prodRate = 1.0;
rate serveRate = 0.9;

rewards
        items>0 : items;
endrewards

module Queue
        items: [0..maxItem] init 0;

        [production] items < maxItem -> prodRate: (items'=items+1);
        [consumption] items > 0 -> serveRate: (items'=items-1);
endmodule
```

Built Model
States: 13
Initial states: 1
Transitions: 24

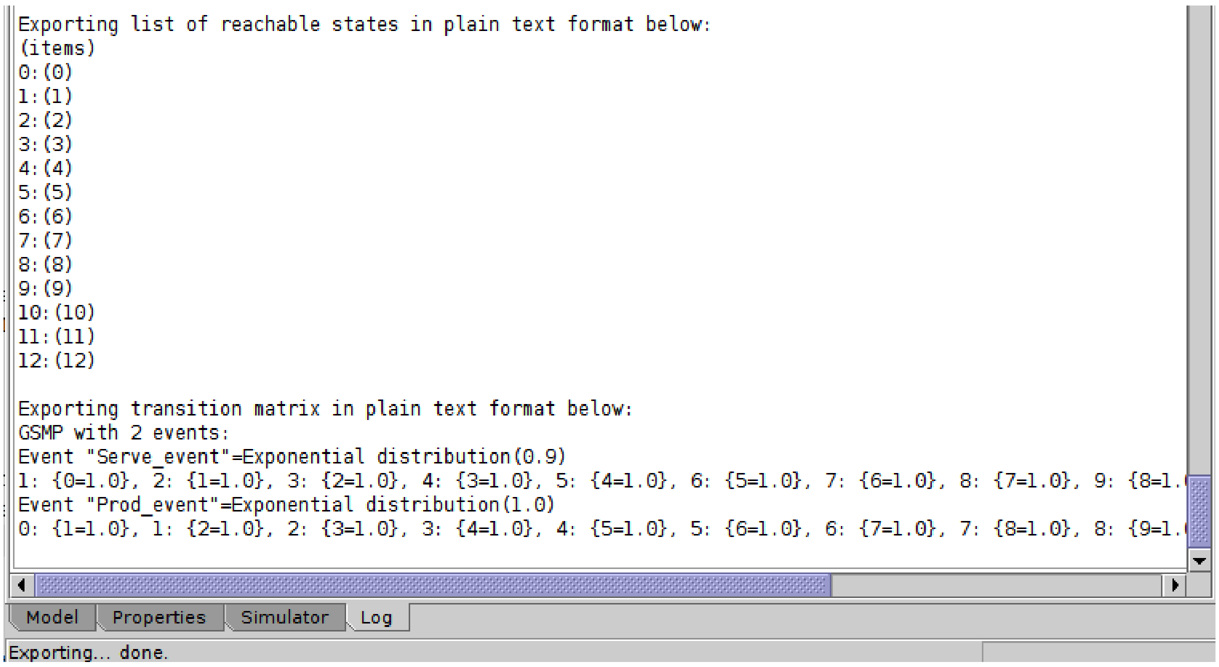
Building model... done.

Let us introduce the new GSMP syntax while keeping the model equivalent. The new GSMP syntax is all about *events* and *distributions*. *Distributions* are assigned to *events*, and *events* are assigned to transition commands. Observe the same model rewritten using the new GSMP syntax.

```
gsmp

const maxItem = 12;
rate prodRate = 1.0;
rate serveRate = 0.9;

rewards
   items>0 : items;
endrewards

module Queue
   event Prod_event = exponential(prodRate);
   event Serve_event = exponential(serveRate);

   items: [0..maxItem] init 0;

   [production] items < maxItem --Prod_event-> (items'=items+1);
   [consumption] items > 0 --Serve_event-> (items'=items-1);
endmodule
```

Module *Queue* now declares two events, *Prod_event* and *Serve_event*. They use the exponential distribution with corresponding rates. Note that the update section of the event transition commands is now a probability distribution that must sum to one.

States and transitions can be exported/viewed as in original PRISM. However, the transitions of a GSMP cannot be described as simply as in the case of a CTMC. GSMP may be perceived as a list of events, where each event has a distribution and transition matrix. This is what PRISM GSMP does when exporting transitions for a GSMP.



```
Exporting list of reachable states in plain text format below:
(items)
0:(0)
1:(1)
2:(2)
3:(3)
4:(4)
5:(5)
6:(6)
7:(7)
8:(8)
9:(9)
10:(10)
11:(11)
12:(12)

Exporting transition matrix in plain text format below:
GSMP with 2 events:
Event "Serve_event"=Exponential distribution(0.9)
1: {0=1.0}, 2: {1=1.0}, 3: {2=1.0}, 4: {3=1.0}, 5: {4=1.0}, 6: {5=1.0}, 7: {6=1.0}, 8: {7=1.0}, 9: {8=1.
Event "Prod_event"=Exponential distribution(1.0)
0: {1=1.0}, 1: {2=1.0}, 2: {3=1.0}, 3: {4=1.0}, 4: {5=1.0}, 5: {6=1.0}, 6: {7=1.0}, 7: {8=1.0}, 8: {9=1.
```

Let us convert the model into a $G/G/1/n$ queue, exploiting the capabilities of the GSMP formalism. The following model uses the *Dirac* distribution with timeout 5 for the arrival event, and *Weibull* distribution with scale 12 and shape 2 for the service event.

```
gsmp

const maxItem = 12;

rewards
   items>0 : items;
endrewards

module Queue
   event Prod_event = dirac(5);
   event Serve_event = weibull(12,2.0);

   items: [0..maxItem] init 0;

   [production] items < maxItem --Prod_event-> (items'=items+1);
   [consumption] items > 0 --Serve_event-> (items'=items-1);
endmodule
```

PRISM GSMP performs many unprecedented semantic checks during model parsing and construction. For example, trying to build the above model with invalid Weibull shape ≤ 0 fails. Failure of these semantic checks generally produces informative errors designed to quickly point the user to what's wrong. Likewise, PRISM GSMP supports previous PRISM language features such as parallel composition of modules, action hiding and action renaming. The most problematic is synchronization of GSMP event commands, which has been implemented with redefined semantics. This is all defined in the aforementioned syntax/semantics document.

4 Running ACTMC analysis

Analysis is restricted to ACTMCs, a subclass of GSMPs. ACTMC is a GSMP with only up to one non-exponential event active in any state. Previously displayed $G/G/1/n$ queue is not an ACTMC, because it has states with two active non-exponential events. However, making either of the events exponential, e.g. a $G/M/1/n$ queue would be a valid ACTMC. PRISM GSMP automatically checks whether a built GSMP is an ACTMC.

```
gsmp

const maxItem = 12;

rewards
   items>0 : items;
endrewards

module Queue
   event Prod_event = dirac(5);
   event Serve_event = exponential(0.25);

   items: [0..maxItem] init 0;

   [production] items < maxItem --Prod_event-> (items'=items+1);
   [consumption] items > 0 --Serve_event-> (items'=items-1);
endmodule
```

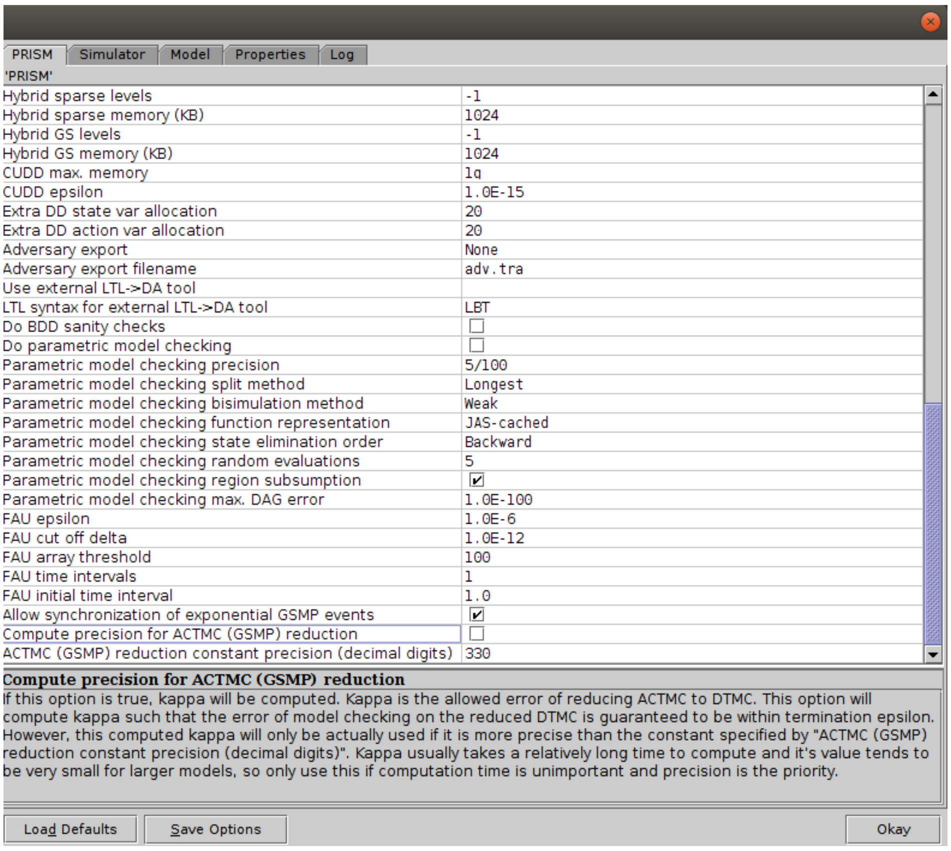
Supported analysis methods thus far are steady-state probabilities, steady-state rewards, and reachability rewards. These can be launched in all standard ways, as in original PRISM for CTMC, for example:

- Steady-state probabilities computation can be launched from the GUI in *Model->Compute->Steady-state probabilities*
- Steady-state rewards can be computed by verifying property *R=? [S]*
- Reachability rewards can be computed by verifying property *R=? [F (prop)]*

PRISM GSMP performs the above analysis methods by reducing the input ACTMC to a DTMC that has on average in the long run equivalent behavior to the input ACTMC. This reduction is done with arbitrarily small allowed error *kappa*. PRISM GSMP adds a couple custom settings related to ACTMC analysis. These are:

- "*Compute precision for ACTMC (GSMP) reduction*"
- "*ACTMC (GSMP) reduction constant precision (decimal digits)*"

They are explained in their respective tooltips. Their default values are designed to offer a reasonable balance between performance and precision. Users of PRISM GSMP should be aware that not only *termination epsilon* is important, but *kappa* is important as well. Both of them can greatly influence the performance and precision of the computation.



'PRISM'

Hybrid sparse levels	-1
Hybrid sparse memory (KB)	1024
Hybrid GS levels	-1
Hybrid GS memory (KB)	1024
CUDD max. memory	1g
CUDD epsilon	1.0E-15
Extra DD state var allocation	20
Extra DD action var allocation	20
Adversary export	None
Adversary export filename	adv.tra
Use external LTL->DA tool	
LTL syntax for external LTL->DA tool	LBT
Do BDD sanity checks	☐
Do parametric model checking	☐
Parametric model checking precision	5/100
Parametric model checking split method	Longest
Parametric model checking bisimulation method	Weak
Parametric model checking function representation	JAS-cached
Parametric model checking state elimination order	Backward
Parametric model checking random evaluations	5
Parametric model checking region subsumption	✓
Parametric model checking max. DAG error	1.0E-100
FAU epsilon	1.0E-6
FAU cut off delta	1.0E-12
FAU array threshold	100
FAU time intervals	1
FAU initial time interval	1.0
Allow synchronization of exponential GSMP events	✓
Compute precision for ACTMC (GSMP) reduction	☐
ACTMC (GSMP) reduction constant precision (decimal digits)	330

Compute precision for ACTMC (GSMP) reduction
If this option is true, kappa will be computed. Kappa is the allowed error of reducing ACTMC to DTMC. This option will compute kappa such that the error of model checking on the reduced DTMC is guaranteed to be within termination epsilon. However, this computed kappa will only be actually used if it is more precise than the constant specified by "ACTMC (GSMP) reduction constant precision (decimal digits)". Kappa usually takes a relatively long time to compute and it's value tends to be very small for larger models, so only use this if computation time is unimportant and precision is the priority.

PRISM GSMP also respects all standard settings such as *termination epsilon*, and *linear equations method*.

Note that in PRISM GSMP, ACTMC reduction is what usually takes vast majority (>99%) of run-time during analysis. Performance of the ACTMC reduction is linearly dependent on number of states with non-exponential events active in them. Such states are processed using subordinated Markov chains. Apart from used *kappa*, processing of a particular state is most visibly dependent on the distribution type and parameters of the event active in it. PRISM GSMP deals with each type of distribution differently, and because of this, their performance varies greatly. Some distributions are also naturally more "vulnerable" to particular parameters. Here are some rough performance expectations for an average PC, using the below $G/M/1/n$ model as a benchmark, with respective distributions used on arrivals.

```
gsmp

const maxItem = 12;

rewards
   items>0 : items;
endrewards

module Queue
   event Prod_event = dirac(5);
   //event Prod_event = uniform(5,5.1);
   //event Prod_event = erlang(20/5,20);
   //event Prod_event = exponential(0.25);
   //event Prod_event = weibull(4, 1);
   event Serve_event = exponential(0.25);

   items: [0..maxItem] init 0;

   [production] items < maxItem --Prod_event-> (items'=items+1);
   [consumption] items > 0 --Serve_event-> (items'=items-1);
endmodule
```

1. *Dirac distribution* processing is by far the fastest. The above $G/M/1/n$ model can be analyzed in a small fraction of a second. Increasing the timeout parameter increases the computation time marginally.

2. *Uniform distribution* processing is slower than in the case of Dirac distribution. However, it is optimized by precomputing the behavior before a using the Dirac distribution. Therefore, using parameters a and b that are close to each other is almost as fast as the Dirac distribution. Analyzing the model with parameters (5, 5.1) takes a few seconds, whereas (5, 120) may take a few minutes.

3. *Erlang distribution* processing is slower than in the case of Dirac distribution. Analysis with parameter values close to 1 takes several seconds. Using parameter values of (100, 20) may take up to a minute.

4. *Exponential distribution* is generally implemented as a special case of Erlang distribution with $k = 1$. In this particular model, using the exponential distribution explicitly causes PRISM GSMP to recognize it as CTMC, leading to performance roughly the same as in original PRISM.

5. *Weibull distribution* processing is *significantly* slower than in the case of Dirac distribution. Analysis with (4, 1) takes a few minutes. Increasing the parameters to (12, 2) causes analysis to take around 10 minutes. Very poor performance is observed when $k < 1$, e.g. analysis with parameters (1, 0.5) may take several hours.

Users of PRISM GSMP should be mindful of these performance limitations.

5 ACTMC OPTIMAL PARAMETER SYNTHESIS

Optimal parameter synthesis is an experimental feature, absent in original PRISM. The objective of optimal parameter synthesis is to maximize (or minimize) steady-state rewards or reachability rewards given a numeric range of possible parameters of an event distribution. The output of optimal parameter synthesis is optimal value for the chosen event distribution parameter such that the given metric is maximized (or minimized). For non-trivial cases, this problem can be very difficult to solve.

Consider some system that tends to become degraded, or even break down after some randomly long uptime. Repair of a broken system is rather time-consuming. It is possible to revert this aging by restarting the system after some rigidly set uptime t, i.e. performing periodic rejuvenation of the system. Restarts take much less time and are preferable to full repairs. Note that using too large t will be obviously ineffective, and that using too small t may actually harm the total uptime of the system by performing restarts too often. Finding optimal value for t such that the total uptime is maximized is a good example of a non-trivial parameter synthesis problem. PRISM GSMP can solve this problem efficiently using recently published algorithms. This particular example can be modeled and analyzed by PRISM GSMP.

5.1 RUNNING ACTMC OPTIMAL PARAMETER SYNTHESIS

For now, only minimizing or maximizing reachability rewards has been implemented, and only Dirac distributed events may be optimized. This is done by verifying properties:

- *Rmin=? [F (target=1)] {(eventName, 1, 0.0000001..50)}* for minimizing reachability reward, or
- *Rmax=? [F (target=1)] {(eventName, 1, 0.0000001..50)}* for maximizing reachability reward.

Note the set of triples at the end of the property.

- Identifier *eventName* indicates particular event that is to be the subject of optimization.
- Integer *1* indicates index of the event distribution parameter that is to be the subject of optimization. Using *1* stands for the first parameter, using *2* for the second parameter.
- Real number range $n..m$ representing a closed interval $[n, m]$ from which the optimal event distribution parameter values are to be selected.

Due to the complexity of used algorithm, computing optimal parameter synthesis takes a few orders of magnitude longer than regular analysis. Nevertheless, PRISM GSMP can compute optimal parameter synthesis for simple models within a few minutes on an average PC. When finished, the optimal parameter value, and reachability reward (or steady-state reward) computed for this optimal value is printed into the log.

Note that all settings that influence the precision and performance of the analysis also work for optimal parameter synthesis.

6 About PRISM GSMP

PRISM GSMP is a free open source project started at Masaryk University in Brno, Czech Republic in late 2017. It is developed by Mário Uhrík under supervision of Vojtěch Řehák and Ľuboš Korenčiak.

You are welcome to contact us (for example, with bug reports or any questions) either through our GitHub page at `https://github.com/VojtechRehak/prism-gsmp`, or by writing an email to `433501@mail.muni.cz`.