The PRISM GSMP and Property Language - Syntax and Semantics

May 15, 2018

This document provides the syntax and semantics for the *generalized semi-Markov process* (GSMP) extension of the PRISM language. It also specifies property language extension that allows for distribution-parameter synthesis in a GSMP. For syntax of the PRISM language, i.e., for CTMC, DTMC, PTA, and MDP, see the manual:

- `http://www.cs.bham.ac.uk/~dxp/prism/manual/ThePRISMLanguage`

For semantics of the PRISM language for CTMC, DTMC, and MDP, see:

- `http://www.prismmodelchecker.org/doc/semantics.pdf`

This document extends the above mentioned materials and the reader is expected to be familiar with them.

Our goal is to provide an extension of the PRISM language syntax and semantics such that it will be able to represent a GSMP in the most comprehensive way for current PRISM users. We first provide a definition of a GSMP.

Definition ([1]). *A generalized semi-Markov process (GSMP) is a tuple* $(S, \mathcal{E}, Act, F, Succ, S_{\text{in}})$ *where*

- S *is a finite set of states,*
- $\mathcal{E}$ *is a finite set of* events*,*
- $Act\colon S \to 2^{\mathcal{E}}$ *assigns to each state* s *a nonempty set of active events* $Act(s) \neq \emptyset$ *in* s*, moreover, we use* $St(e)$ *to denote states where* e *is active, i.e.,* $St(e) = \{s \mid e \in Act(s)\}$*,*
- $Succ : \mathcal{E} \to \mathcal{M}(S)$ *is a* successor function *that assigns to an event* e *a probability transition matrix* $St(e) \times S$ *specifying for each state* $s \in St(e)$ *probability distribution over its successors,*
- $F\colon \mathcal{E} \times \mathbb{R} \to [0,1]$ *provides, for each event* $e \in \mathcal{E}$*, a cumulative distribution function (CDF)* $F(e,\cdot)$ *such that* $F(e,0) = 0$*, and*
- $S_{\text{in}} \subseteq S$ *is a nonempty set of* initial states*.*

Syntax	CDF	Argument Restrictions
`exponential(a)`	$F(\tau) = 1 - e^{-\mathtt{a}\tau}$	$\mathtt{a} \in \mathbb{R}_{>0}$
`weibull(a,b)`	$F(\tau) = 1 - e^{-(\tau/\mathtt{a})^{\mathtt{b}}}$	$\mathtt{a}, \mathtt{b} \in \mathbb{R}_{>0}$
`erlang(a,b)`	$F(\tau) = 1 - \sum_{n=0}^{\mathtt{b}-1} \frac{1}{n!} \cdot (\mathtt{a}\tau)^n \cdot e^{-\mathtt{a}\tau}$	$\mathtt{a} \in \mathbb{R}_{>0}$, $\mathtt{b} \in \mathbb{N}$
`dirac(a)`	$F(\tau) = \begin{cases} 0 & \text{for } \tau < \mathtt{a} \\ 1 & \text{for } \tau \geq \mathtt{a} \end{cases}$	$\mathtt{a} \in \mathbb{R}_{>0}$
`uniform(a,b)`	$F(\tau) = \begin{cases} 0 & \text{for } \tau < \mathtt{a} \\ \frac{\tau - \mathtt{a}}{\mathtt{b} - \mathtt{a}} & \text{for } \tau \in [\mathtt{a}, \mathtt{b}) \\ 1 & \text{for } \tau \geq \mathtt{b} \end{cases}$	$\mathtt{a}, \mathtt{b} \in \mathbb{R}_{\geq 0}, \mathtt{a} < \mathtt{b}$

Table 1: Cumulative distribution functions.

GSMP Semantics The execution of a GSMP starts in one of the initial states $s_{\text{in}} \in S_{\text{in}}$ (if $|S_{\text{in}}| > 1$ PRISM checks the model for an arbitrary choice of the initial state).[1] In every state each active event keeps one timer. In s_{in} all active events (given by $Act(s_{\text{in}})$) do not have a timer yet, thus we set a timer for each event e in $Act(s_{\text{in}})$ to a random value according to CDF $F(e, \cdot)$. The event e, which has the minimal timer value, first waits until its timer goes off then it occurs and causes a change of state.[2] The next state s' is chosen randomly with probability $Succ(e)[s_{\text{in}}, s']$. In the new state s', the active events that either just occurred $Act(s') \cap \{e\}$ or are newly active $Act(s') \setminus Act(s_{\text{in}})$ do not have a valid timer value, thus their timer is set the same way as above. The inherited events $Act(s') \cap Act(s_{\text{in}}) \setminus \{e\}$ keep their old timer value, that is decreased by the time spent in s_{in}. Then the execution proceeds in the same manner.

Syntax of GSMPs in PRISM

The syntax of GSMPs extends the CTMC syntax. Hence, we specify only the syntax that was added to the CTMC syntax. Each GSMP model file must begin with the keyword `gsmp`.

Distributions Before the specification of modules, it is possible to specify several variables of type distribution, e.g.,

```
const distribution dist1 = dirac(3*1);
```

Currently, we support five distributions: exponential, Weibull, Erlang, Dirac, and uniform. All the distributions have two arguments of type double except for the Dirac and exponential distributions that have only one. Table 1 provides the syntax of the distribution specifications along with the associated CDF and restrictions on the arguments. The value of an argument can be given by an arithmetic expression of numbers and constants that evaluates to a number.

Events All events are local to a module. To preserve backward compatibility, events with exponential CDF can also be specified in the same way as in the PRISM CTMC models. The remaining

[1] There is also an option to specify an initial distribution on states in the command line version of PRISM. We do not elaborate this option in this document for the sake of readability and since the corresponding semantics is clear.

[2] If there are multiple events with the minimal timer value, one of them is randomly chosen- each with the same probability.

events have assigned identifiers and must be declared immediately after the module name. There are two possibilities of declaring an event:

- `event f = dist1;` and
- `event f = dirac(3.0);`

Both such declarations define an event `f` with a Dirac distributed delay set to the value 3.0. For the event `f`, we specify its set of active states (i.e., $St(\texttt{f})$) and successor function (i.e., $Succ(\texttt{f})[\cdot,\cdot]$) by PRISM commands where the event identifier `f` is in the arrow. E.g.,

```
[L] s=1 --f-> 0.3:(s'=0) + 0.7:(s'=2);
```

specifies that the event `f` is active in all states where `s=1` and whenever it occurs, the next state is derived from the original one by changing variable `s` to `0` with probability 0.3 and to `2` with probability 0.7. The probabilities in each command have to sum to one (similarly to DTMC commands in PRISM). The labels (e.g., `L`) are used for synchronization and to impose impulse rewards as for CTMC.

Moreover, we allow *slave* commands for the synchronization purposes. The slave command is denoted by `--slave->` that indicates that it has no delay and obtains a delay of a (*master*) command that it synchronizes with. E.g.,

```
[L] t=1 --slave-> 0.3:(t'=0) + 0.7:(t'=2);
```

synchronizes with commands with label `L` of other modules.

Semantics

First, we translate all (old-fashioned) CTMC commands (one by one) into GSMP commands. For each CTMC command `[a] g -> r_1:u_1 + ⋯ + r_n:u_n` (where `a` is a synchronization label or an empty string), we exchange the command for

```
const double r_new_sum = r_1 + ⋯ + r_n;
event e_new = exponential(r_new_sum);
[a] g --e_new-> (r_1/r_new_sum):u_1 + ⋯ + (r_n/r_new_sum):u_n;
```

where `r_new_sum` and `e_new` are fresh identifiers for each CTMC command.

An important observation is that we do not define the semantics in a compositional manner, i.e., by first giving the semantics of each module in the system and then combining these results. The reason for this is that guards (and updates) of one module are allowed to refer to the variables of other modules (and indeed global variables). Instead, we define the semantics of a system by translating its set of modules into a single *system module* (in a compositional manner) and then defining the semantics for the system through this single system module.

Constructing the system module

In this section, we describe the process of constructing the system module from its component modules. The composition of the modules is defined by a process-algebraic expression which can include parallel composition of modules, action hiding and action renaming. We now consider each of these in turn. Note that, in this construction process, we require that all updates of all commands have been expanded to explicitly include all local variables of the module and all global variables, even those that do not change.

Parallel composition

Although there are three types of parallel composition, we need only consider the case $M_1||[A]||M_2$, since $M_1|||M_2$ is equivalent to $M_1||[\emptyset]||M_2$ and $M_1||M_2$ is equivalent to $M_1||[A_1 \cap A_2]||M_2$ where A_i is the set of actions that appear in module M_i. The commands of the module $M = M_1||[A]||M_2$ are constructed according to the following rules:

1. for each command `[] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M_1,
   copy the command to the commands of M;
2. for each command `[] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M_2,
   copy the command to the commands of M;
3. for each `a` $\notin A$ and command `[a] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M_1,
   copy the command to the commands of M;
4. for each `a` $\notin A$ and command `[a] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M_2,
   copy the command to the commands of M;
5. for each `a` $\in A$, command `[a] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M_1 and command `[a] h --f-> r_1:v_1 + ⋯ + r_m:v_m;` of M_2
   - if `e` = `slave` or `f` = `slave`, **add** to the commands of M

     ```
     [a] g & h --ef-> p_1*r_1 : u_1 & v_1  +  ⋯  +  p_n*r_1 : u_n & v_1  +
                      p_1*r_2 : u_1 & v_2  +  ⋯  +  p_n*r_2 : u_n & v_2  +
                         ⋮          ⋮                  ⋮          ⋮
                      p_1*r_m : u_1 & v_m  +  ⋯  +  p_n*r_m : u_n & v_m  ;
     ```

     where `ef` is the non-slave event of `e` and `f`, or it is `slave` if both `e` and `f` are `slave`.
   - if `e` $\neq$ `slave` and `f` $\neq$ `slave`,
     - if the setting flag `ExpSyncBackwardCompatible` is set to true and both `e` and `f` are exponentially distributed, we **add** to the commands of M

       ```
       [a] g & h --ef-> p_1*r_1 : u_1 & v_1  +  ⋯  +  p_n*r_1 : u_n & v_1  +
                        p_1*r_2 : u_1 & v_2  +  ⋯  +  p_n*r_2 : u_n & v_2  +
                           ⋮          ⋮                  ⋮          ⋮
                        p_1*r_m : u_1 & v_m  +  ⋯  +  p_n*r_m : u_n & v_m  ;
       ```

       where `ef` is a newly created exponentially distributed event with the rate equal to the multiplication of the rates of `e` and `f`.
     - otherwise, return an **error** stating that synchronization between two non-slave events is not allowed (note that this case is not possible if `ExpSyncBackwardCompatible` is set to true and both the events are exponentially distributed).

Action hiding

The commands of $M = M'/A$ are constructed according to the following rules:

1. for each command `[] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M',
   copy the command to the commands of M;
2. for each `a` $\notin A$ and command `[a] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M',
   copy the command to the commands of M;
3. for each `a` $\in A$ and command `[a] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M' where `e` $\neq$ `slave` (i.e., `e` is a master event),
   add `[] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` to the commands of M;
4. for each `a` $\in A$ and command `[a] g --slave-> p_1:u_1 + ⋯ + p_n:u_n;` of M',
   print a **warning** stating that a non-mastered slave event was ignored.

Action renaming

The commands of $M = M'\{a_1 \leftarrow b_1, \ldots, a_m \leftarrow b_m\}$ are constructed as follows:

1. for each command `[] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M',
   copy the command to the commands of M;
2. for each `a` $\notin \{a_1, \ldots, a_m\}$ and command `[a] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M',
   copy the command to the commands of M;
3. for each `a` $\in \{a_1, \ldots, a_m\}$ and command `[a] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` of M',
   add `[b] g --e-> p_1:u_1 + ⋯ + p_n:u_n;` to the commands of M where `b` is the label of $\{b_1, \ldots, b_m\}$ corresponding to `a`.

The Semantics of the System Module

In this section, we give the semantics of a system module, constructed according to the rules described in the previous section. We suppose that:

- C is the multiset of commands generated by the rules above;
- $V = \{v_1, \ldots, v_m\}$ is the set of variables, both local and global, that appear in the system description; and
- $\mathcal{E}$ is the set of all declared events.

We construct the state space of the system as follows. A state is a tuple $(x_1, \ldots, x_m)$ where x_i is a value for the variable v_i. The set of all states S is therefore the set of all possible valuations of the variables in V. The set of initial states can be specified in one of the following two ways: either by giving an initial value for each variable, or by giving a predicate over variables (using the `init...endinit` construct). In the former case, $S_{\text{in}} \stackrel{\text{def}}{=} \{s\}$ where $s = (x_1, \ldots, x_m)$ and x_i is a the initial value of the variable v_i (recall that, if the initial value of a variable is left unspecified, it is taken to be the minimum value of the variables range). In the latter case, S_{in} is the subset of states S which satisfy the predicate specified in the `init...endinit` construct.

We now consider the semantics for a single command of the system module. From this point, we ignore any action-labels assigned to commands in C; these were required only for the process-algebraic construction and can now be safely discarded. Moreover, for the same reason we ignore also slave commands, i.e., commands that use `--slave->`. Similarly to the item 4 of action hiding, we can print a **warning** stating that a non-mastered slave event was ignored. Hence, each command c of C is of the form:

`[] g --e-> p_1:u_1 + ⋯ + p_n:u_n;`.

Since the guard `g` is a predicate over the variables in V and each state of the system is a valuation of these variables, `g` defines a subset of the global state space S where the command c is applicable. We denote this set of states as $S_c = \{s \in S \mid s \models \texttt{g}\}$.

Each update `u_j` of c corresponds to a transition that the system can make when it is in a state $s \in S_c$. The transition is defined by giving the new value of each variable as an expression (possibly using the values of the current state s). Hence, we can think of `u_j` as a function $u_j \colon S_c \to S$. If `u_j` is $(v'_1 = expr_1) \wedge \cdots \wedge (v'_m = expr_m)$, then for each state $s \in S_c$:

$$u_j(s) = (expr_1(s), \ldots, expr_m(s))$$

where $expr_i(s)$ is the expression $expr_i$ evaluated in the state s, for each $i = 1, \ldots, m$.

For each command $c \in C$ and state $s \in S_c$, we define a function $\mu_{c,s} \colon S \to \mathbb{R}_{\geq 0}$ where for each $t \in S$ and the command c of the form `[] g --e-> p_1:u_1 + ⋯ + p_n:u_n;`

$$\mu_{c,s}(t) \stackrel{\text{def}}{=} \sum_{\substack{1 \leq j \leq n \\ \wedge\ u_j(s) = t}} \texttt{p\_j}\ .$$

For all $s \notin S_c$ and $t \in S$, we set $\mu_{c,s}(t) = 0$. Note that, for GSMPs, the syntactic constraints placed on the constants `p_j` mean that the function $\mu_{c,s}$ is actually a probability distribution over S.

GSMP Semantics

We already defined the state space S, the set of events $\mathcal{E}$ and the initial states S_{in}. We will define the remaining structures Act, $Succ$, and F separately for each event.

Let us fix an event $e \in \mathcal{E}$. Let C_e be the set of all commands activated by the event e. We define the function St as $St(e) \stackrel{\text{def}}{=} \sum_{c \in C_e}$. Then for each $s \in S$ we set $Act(s) \stackrel{\text{def}}{=} \{e \in \mathcal{E} | s \in St(e)\}$. The CDF $F(e, \cdot)$ is determined by the distribution associated during the event declaration. Finally, we define the probability transition matrix $Succ(e)$ of event e as follows. For each state $s \in St(e)$ and state $t \in S$ we set

$$Succ(e)[s,t] \stackrel{\text{def}}{=} \frac{\sum_{c \in C_e} \mu_{c,s}(t)}{\sum_{c \in C_e \wedge t' \in S} \mu_{c,s}(t')}.$$

The normalisation is required since $\sum_{c \in C_e \wedge t' \in S} \mu_{c,s}(t')$ can sum to more than one, through either the nondeterminism introduced through the parallel composition of modules or local nondeterminism in a module (i.e., overlapping guards). This normalisation can be considered as replacing any nondeterministic choice between a set of transitions with a uniform (probabilistic) choice between the transitions.

Properties for Distribution-Parameter Synthesis

We extend the PRISM properties working with rewards by specification of a list of parameters to be the subject of optimization. Currently, we support both minimizing and maximizing two objectives: reachability reward and steady-state reward. We express them as follows:

```
Rmax=?  [ F (prop) ] {(ev1,1,0..10), (ev2,1,2..123), (ev3,2,1.2..72.6)}
```

This states that we are looking for optimal values for the first distribution parameters of events `ev1` and `ev2`, and the second parameter of distribution of `ev3` in order to maximize the expected accumulated reward before reaching a state satisfying `prop`. For the parameters, we allow only values within the intervals `0..10`, `2..123`, and `1.2..72.6`, i.e., in the last case we require values ≥ 1.2 and ≤ 72.6. Similarly,

```
R{"rew_name"}min=? [ S ] {(ev1,1,0..100)}
```

expresses that we would like to find the optimal value within interval `0..100` for the first distribution parameter of event `ev1` minimizing the steady-state reward using reward structure `rew_name`.

References

[1] P.J. Haas. *Stochastic Petri Nets: Modelling, Stability, Simulation.* Springer, 2010.