Simulation of Generalized Semi-Markov Processes

Dimuth Kulasinghe
Neeraj Pradhan

A generalized semi-Markov process (GSMP) is a discrete event model that is used to simulate a large class of stochastic systems. The goal of the project is to develop a Haskell library for GSMP simulation that is flexible, robust and requires minimal system specification from the user. We present the first version of the library, the design challenges faced during the development and future work.

I. Introduction to GSMPs

GSMPs offer a very general framework for simulating discrete event systems in which the state evolution is guided by events that are triggered by a stochastic process. [1] Even though state evolution does not have an explicit dependence on the past states, GSMPs are not strictly Markovian or 'memoryless', as there is an implicit dependence on events that may have been triggered by past states.

The key building blocks for GSMPS are states, events and clocks. (Table 1)

- Events associated with a state compete to trigger the next state transition.
- Each event has its own distribution for determining the next state.
- For each new event, a clock is scheduled that indicates the time until the event is scheduled to occur.
- At each state transition, the following events need to scheduled or canceled:
  - Old events that are also active events in the current state: clocks continue to run down.
  - New events that become active in the current state: clocks need to be scheduled.
  - Old events that are not active in the current state: clocks need to be canceled.

Starting from an initial state, we generate sample paths for a given GSMP and calculate point estimates and confidence interval for a quantity of interest. The GSMP needs to be specified in a way that this quantity is expressible in terms of the GSMP state.

II. Algorithm and Library Details

We follow the 'variable time advance' algorithm to simulate the GSMP. A flowchart of the above algorithm is shown in Fig. 1. Broadly the algorithm can be specified as follows:

- Starting from an initial distribution, we generate the set of active events in the state. The clocks for these events are generated from the specified distributions, and scheduled in a priority search queue.
- There is a stochastic transition from the current state to the next state depending on the next triggering event (the minimum element in the priority queue).

GSMP Building Blocks	
S	Finite or countably infinite set of states
E	Finite set of events
E(s)	Set of events scheduled to occur in state s
P(s'; s, e*)	Probability that the new state is s' given that the event e* occurs in state s
F(. ;s',e',s, e*)	The distribution function to set the clock for the new event e' when the event e* triggers a state transition from s to s'
μ	The initial distribution function for the state and clocks.

Table 1. The building blocks of a GSMP provided by the client system specification.

- When the system transitions, we schedule any new events and cancel the inactive events from the priority search queue. We update the estimator with the new value for the quantity of interest.
- The above process is repeated until the termination criterion, as specified by the user, is met.
- We carry out many simulation replications starting from a random seed, so as to obtain the point estimate of our quantity of interest within the desired confidence bounds.

Some of the client-side requirements are listed in Table 2. The user specification for the state-space can be very general, depending on the problem requirements. The event data type also contains a function that is used to specify the distribution of the clock readings, by using inverse transform sampling. [2]

Note that unlike a finite state Markov process, the state-space can possibly be infinite in the case of GSMPs. As such, instead of a transition matrix, clients must provide the specification for state transitions through a transition function.

The important modules for the GSMP simulation are as follows:

1. **Estimator.hs:** Functions to keep track of the current value of the estimator, so that quantities like mean, variance, confidence intervals etc. can be calculated for the given quantity of interest. Numerically stable formulae for these statistical measures are used that require O(1) space so that we do not have to keep all the simulation values in memory.
2. **StateSpace.hs:** Defines the types (interfaces) that client code must use to specify the state-space, events, transitions and active events (see Table 2 for more details).
3. **EventList.hs:** Defines the event list data structure, which uses a priority search queue to schedule events, and other helper functions to schedule events at each state transition.
4. **GSMP.hs:** The heart of the algorithm that specifies how to simulate the system to obtain point estimates for the quantity of interest, to a desired precision.

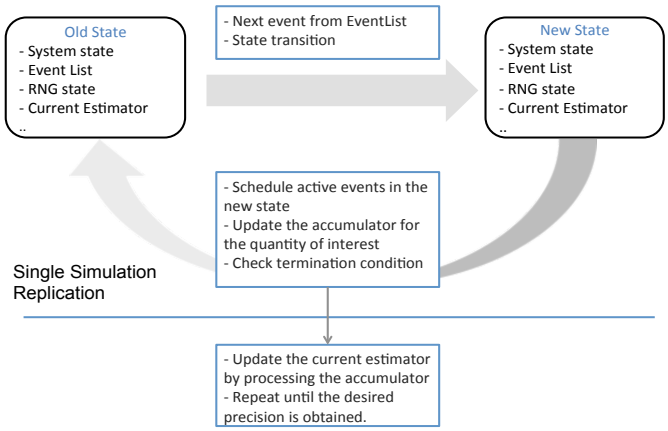

Fig 1. Flowchart of 'variable time advance' algorithm for GSMP simulation.

Important User Supplied Types and Functions
data StateSpace s Defines the statespace for the GSMP. 's' can be an Int, Vector, tuple or any other type that are instances of some common type classes.
data Event a An event with IDs given by the type parameter 'a', associated with the given GSMP. Also stores the distribution function for the clock setting.
Active Events :: StateSpace s -> [Event a] The set of active events in a given state, whose clocks either need to be scheduled or allowed to run down.
Canceled Events :: StateSpace s -> [a] The set of inactive events in a given state, which need to be canceled.
Transition :: StateSpace s -> a -> Dist (StateSpace s) Given a state and a triggering event, the resulting distribution of the next possible states.
State Quantity :: StateSpace s -> Time -> Acc -> Acc The quantity of interest that is updated according to the current state and time of transition and stored in the accumulator.

Table 2. Client specified functions and types used by the library for GSMP simulation

III. Design Challenges

We faced many challenges during the design of the simulation library, some of which are enumerated below.

1. **Stateful Computations:** The GSMP simulation is a stateful process, where at any given time, we need to store multiple states – the current state of the simulation, event-list, random number state and the current estimate of the quantity of interest. Passing around these states to multiple functions can be highly unwieldy. However, by using lenses inside the state monad, we could not only hide this plumbing within a record data structure, but also be able to modify the different components of the record with ease.

   We however, discovered, that performance tuning and discovering memory leaks could be complicated if there are multiple functions that pass around an internal state, as in our simulation.

2. **Imposing Constraints on Type Values:** We used GADTs to impose class constraints in data declarations and smart constructors to constrain the values of a certain data type (e.g. the Probability data type in our module). However, there are certain static constraints that were not possible to incorporate easily in the library API, as Haskell does not have a system of dependent types:

   - Restricting the domain of the state-space as specified by the client so that all resulting client functions (transition functions, active states etc.) are assured to give valid state transitions. We currently rely on the client to handle this by wrapping the returned values inside a Maybe data type.
   - The transition function when given a certain state and event combination where the event is inactive in the state must report an invalid transition. Again, we are relying on the client to provide the correct specification.

3. **Stochasticity:** We have many functions in the code that need to use randomly generated values. The only way to have a pure function that results in stochastic values, violates the principle of referential transparency, is to pass around the random number generator state with the function and update its state as we continue to use the random number stream. Much of this plumbing is currently being taken care of in the state monad.

   Another issue is that most of the faster random numbers reside either in the ST or IO monad like mwc-random and mersenne-random. Both these RNGs, however, do not provide the functionality for splitting the random number generator, which can be extremely useful when multiple functions need to use streams of random numbers. More importantly, care needs to be taken to use these RNGs without suffering from monadic creep. In the end, we decided to stick with the default System.Random module to preserve purity and simplicity.

IV. Further Work

Further work is required to increase the robustness and usefulness of the library. As it stands now, simulation can be fairly slow and memory consuming. Part of the challenge is also that we do not have control over the way clients provide inputs to the simulation, sometimes resulting in inefficient specifications.

We plan to work on the following improvements to the library:

- Faster random number generation by moving away from the default System.Random module. As mentioned, this may result in moving part of the code to the ST monad.
- Incorporating more checks for the specifications provided by the user, to

ensure the validity of system specification in terms of transition functions and active events. Currently, very few checks are done and we rely on clients to provide a correct specification.

- Incorporating report functionality so that users can see the distribution of simulation values, instead of only being provided with the final point estimate and confidence bounds. This will likely require using the Writer monad to log simulation values generated.

V. Conclusion

Simulation is a stateful process, and the particular algorithm that we used fits very well in the object-oriented paradigm. While Haskell offers features like lenses and monadic state that can be used to write code for stateful applications, we think that a more functional reformulation of the simulation algorithm is desirable.

Ideally, we would like to leverage Haskell's powerful type system to infer the validity of client supplied simulation specification as far as possible, and to use the lazy nature of the language to refine our estimator for the quantity of interest until we reach a desired precision. Our algorithm currently needs to work against the functional and lazy nature of Haskell, because it is inherently an object-oriented algorithm.

Functional formulation for simpler probabilistic computations has been suggested. [3] However, in case of GSMPs, this may not be an easy problem because the infinite state-space of a GSMP and its non-Markovian character makes it difficult to construct a more functional formulation in terms of a transition matrix.

VI. References

[1] P. J. Haas, "Stochastic Petri Nets: Modelling, Stability, Simulation", Springer-Verlag

[2] Luc Devroye, "Non-Uniform Random Variate Generation", Springer-Verlag

[3] M. Erwig, S. Kollmansberger, "Functional Pearls: Probabilistic Functional Programming in Haskell", http://web.engr.oregonstate.edu/~erwig/papers/PFP_JFP06.pdf