

SMART:

Simulation and Markovian Analyzer for Reliability and Timing

Gianfranco Ciardo

Andrew S. Miner

Department of Computer Science, College of William and Mary, Williamsburg, VA 23187
 {ciardo,asminer}@cs.wm.edu

Abstract—SMART is a new tool designed to allow various high-level stochastic modeling formalisms (such as stochastic Petri nets and queueing networks) to be described in a uniform environment and solved using a variety of solution techniques, including numerical methods and simulation. Since SMART is intended as a research tool, it is written in a modular way that permits the easy integration of new solution algorithms.

I. SMART LANGUAGE

Models are described to SMART using a strongly-typed, declarative language. The three basic predefined **types** for the objects defined in SMART are:

- **bool**: true or false.
- **int**: integer values.
- **real**: real values (machine-dependent precision).

Composite types can be defined using the concepts of:

- **sets**: collection of homogeneous objects.
- **arrays**: multidimensional data structures of homogeneous objects indexed by the elements of a set.
- **aggregates**: analogous to the Pascal “record”.

A type can be further modified by the following **natures**, which describe stochastic characteristics:

- **const** (the default): a non-stochastic quantity.
- **ph** or **rand**: a random variable.
- **proc**: a stochastic process indexed by time.

Declaration of new functions with parameters is allowed. Functions can be recursive:

```
int fact(int n) := if(n=0,1,n*fact(n-1));
```

Functions can be overloaded, as long as they can be distinguished by the types of their formal parameters. Function calls can pass parameters by position or by name:

```
int raiseto(int base, int exp) := ...;
...
raiseto(2,16);
raiseto(exp:=16, base:=2);
```

Parameters can be assigned default values that are used if the passed parameter is the keyword **default** or if not all of the named parameters are specified:

```
real root(real arg, int n:=2) := ...;
...
root(arg:=64, n:=3);           /* 4 */
root(64, default);           /* 8 */
root(arg:=64, n:=default);    /* 8 */
root(arg:=64);               /* 8 */
```

Arrays are functions declared within a **for** statement. The dimensionality of the array is determined by the enclosing **for** iterators. Conceptually, the indices of the array along each dimension are the elements of a finite set, hence it is legal to define arrays with **real** indices. For example,

```
for (int s in {1..5}, real f in {1..s..0.1}) {
  real res[s][f] := MyModel(s,f).out1;
}
```

stores the results of a parametric study of **MyModel** in array **res**, where the output measure **out1** is computed when the first input parameter ranges from one to five and the second one ranges from one to the value of the first parameter, with a step of one-tenth.

For approximate solutions based on stochastic decomposition [4], fixed-point iterations can be specified using the **converge** statement:

```
converge {
  real x guess 1.0;
  real y := fy(x, y);
  real x := fx(x, y);
}
```

The iterations stop when two subsequent **x** values (and **y** values) differ by less than ϵ in either relative or absolute terms. The values for **x** and **y** are updated either immediately or at the end of each iteration. Both ϵ and the updating criterion are fine-tuned using **option** statements.

The **converge** and **for** statements can be nested arbitrarily within each other.

II. RANDOM VARIABLES

SMART can manipulate discrete and continuous phase-type distributions, which correspond to the **ph int** and **ph real** types. An operation on a **ph** type produces another **ph** type if phase-type distributions are closed under that operation; otherwise, the result is a **rand** type:

```
ph int X := geometric(0.7);
ph int Y := discreteuniform(1,5);
ph int sumXY := X+Y;
ph int prodX := 4*X;
ph int chooseXY := choose(0.4:X, 0.6:Y);
ph int minXY := min(X, Y);
```

However, mixing **ph int** and **ph real**, or performing other operations not guaranteed to result in a phase-type distri-

bution, forces SMART to consider the random variable as generally distributed:

```
rand int diffXY := X-Y;
rand real sumRX := 4.5+X;
rand int prodXY := X*Y;
```

The resulting random variables can then be manipulated only via Montecarlo methods (not yet implemented).

III. MODEL FORMALISMS

The declaration of a model is similar to that of a function. Instead of a return value, a model declaration specifies a block that defines the model. A model definition consists of three parts: declarations, specification, and user measures. Components of the model, such as the places and transitions of a Petri net, are defined in the declaration section. The model is then specified by calling formalism-specific functions. In the case of a Petri net, these are functions that specify such things as the structure of the net, the firing times of the transitions, the initial marking, and so on. Measures are declared as user-defined functions that compute some quantity of interest, such as the expected number of tokens in a given place. Measures are the only part of a model that can be accessed outside of the model definition block. Example model code is shown in Figure 1, the associated Petri net is shown in Figure 2.

The design of SMART allows for relatively easy addition of new model formalisms.

IV. DISTRIBUTED VERSION

SMART can distribute work on a network of workstations in two ways: large models can be solved using distributed algorithms, and different models can be solved simultaneously on different machines.

A. Distributed algorithms

Following the ideas in [1], the state space of any discrete-state model can be explored using a distributed algorithm. If N workstations are available, the resulting state space is partitioned in N classes, one per workstation. We are currently implementing the distributed numerical solution of the process, assuming it is a Markov chain. This approach provides the ability of solving large models by exploiting the overall available memory.

B. Concurrent solutions

In many cases, the same model needs to be solved for a large number of parameter combinations. These can be easily specified using the array feature in SMART, which is then able to maintain a pool of jobs to be sent to remote hosts. SMART builds a dependency graph, so that solutions that depend on previous results wait until those results have been computed.

This is by far the most efficient use of multiprocessing, provided each solution is small enough to fit in the main memory of the processor to which it is assigned. The amount of communication is limited to a scheduler sending

```
spn kanban(int n) := {
  /* Declarations */
  place pm1, pback1, pkan1, pout1,
        pm2, pback2, pkan2, pout2,
        pm3, pback3, pkan3, pout3,
        pm4, pback4, pkan4, pout4;
  trans tin1, tredo1, tok1, tback1,
        tin23, tredo2, tok2, tback2,
        tin4, tredo3, tok3, tback3,
        tredo4, tok4, tback4, tout4;
  /* Transition Firing Times */
  firing(tin1:expo(1.0), tredo1:expo(0.36),
         tok1:expo(0.84), tback1:expo(0.3),
         tin23:expo(0.4), tredo2:expo(0.42),
         tok2:expo(0.98), tback2:expo(0.3),
         tin4:expo(0.5), tredo3:expo(0.39),
         tok3:expo(0.91), tback3:expo(0.3),
         tredo4:expo(0.33), tok4:expo(0.77),
         tback4:expo(0.3), tout4:expo(0.9));
  /* Net Structure */
  arcs(pkan1:tin1, tin1:pm1, pm1:tredo1,
       pm1:tok1, tredo1:pback1, tok1:pout1,
       pback1:tback1, tback1:pm1, pout1:tin23,
       tin23:pkan1, pkan2:tin23, tin23:pm2,
       pm2:tredo2, pm2:tok2, tredo2:pback2,
       tok2:pout2, pback2:tback2, tback2:pm2,
       pout2:tin4, tin4:pkan2, pkan3:tin23,
       tin23:pm3, pm3:tredo3, pm3:tok3,
       tredo3:pback3, tok3:pout3,
       pback3:tback3, tback3:pm3, pout3:tin4,
       tin4:pkan3, pkan4:tin4, tin4:pm4,
       pm4:tredo4, pm4:tok4, tredo4:pback4,
       tok4:pout4, pback4:tback4, tback4:pm4,
       pout4:tout4, tout4:pkan4);
  /* Initial Marking */
  init(pkan1:n, pkan2:n, pkan3:n, pkan4:n);
  /* Measures */
  real e1 :=
    avg(tk(pm1)+tk(pback1)+tk(pout1));
  real e4 :=
    avg(tk(pm4)+tk(pback4)+tk(pout4));
};
```

Fig. 1. SMART Code for a Petri net

solution requests to the remote hosts, and receiving back the requested measures.

V. ADVANCED FEATURES

Various experimental implementations are ongoing in the SMART project. These are being used to assess the effectiveness of new algorithms and data-structures.

A. State-space storage

Since the generation and storage of the state space is a major component of any state-space based solution technique, we have investigated more efficient strategies than

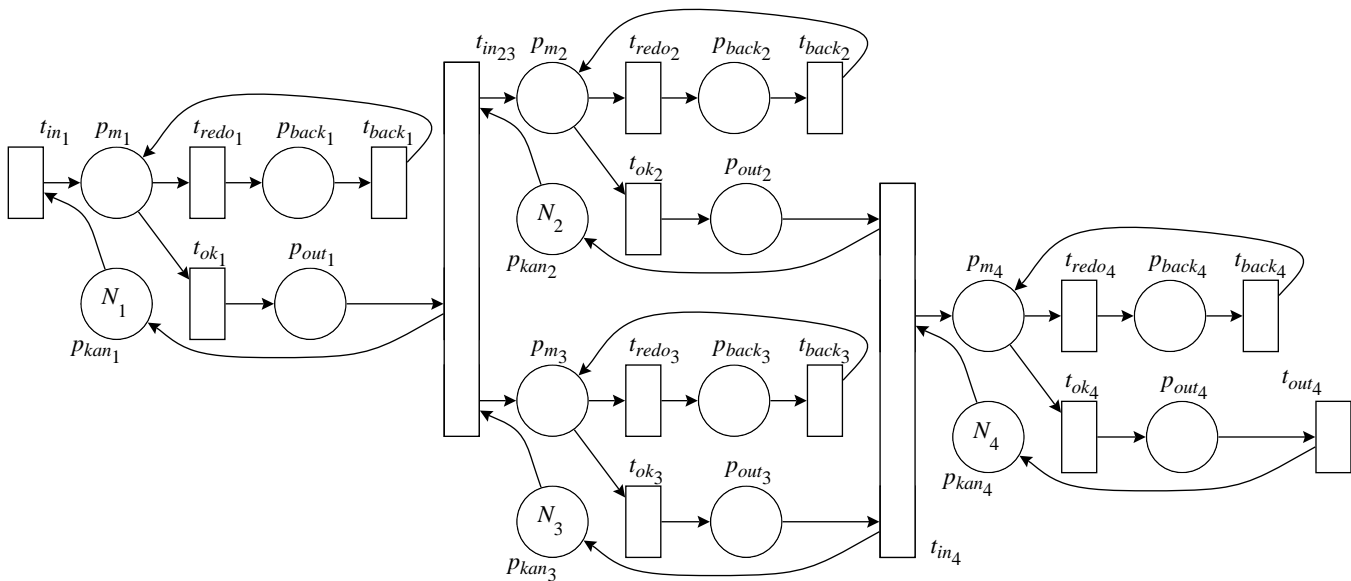


Fig. 2. Petri net described by the SMART code in Fig. 1

the straightforward breadth-first approach using a single search tree for the currently known portion of the state space. We have implemented in SMART a general multi-level method, which allows to store large state space with (asymptotically) a small constant amount of memory per reachable state (e.g., one byte), regardless of the complexity of the model [2]. An additional advantage is that the execution time is reduced as well.

We are also investigating the use of binary-decision diagrams (BDDs) for the same purpose [7].

B. Kronecker algebra

We have also begun an implementation of general solution methods based on Kronecker operators for the storage of the transition rate matrix. Unlike approaches based on the potential state space [5], [8], [9], we use the actual state space, which is often much smaller [3].

The state-space storage technique mentioned in the previous section is employed here to reduce the computational overhead.

C. Numerical solvers

In addition to the more traditional numerical solution algorithms (Power, Gauss-Seidel), we are exploring multigrid-like methods [6] and the automatic computation of optimal relaxation parameters.

VI. ACKNOWLEDGEMENTS

This research was partially supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480; by a joint STTR project with Genoa Software Systems, Inc., for the Army Research Office; and by a matching grant from the Virginia Center for Innovative Technology.

REFERENCES

- [1] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state-space generation of discrete-state stochastic models. *INFORMS J. Comp.* To appear.
- [2] G. Ciardo and A. S. Miner. Storage alternatives for large structured state spaces. In *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, St. Malo, France, June 1997. To appear.
- [3] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, Hampton, VA, 1996. Submitted for publication.
- [4] G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Perf. Eval.*, 18(1):37–59, 1993.
- [5] S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In R. Valette, editor, *Application and Theory of Petri Nets 1994, Lecture Notes in Computer Science 815 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets, Zaragoza, Spain)*, pages 258–277. Springer-Verlag, June 1994.
- [6] G. Horton and S. T. Leutenegger. A multi-level solution algorithm for steady state Markov chains. In *Proc. 1994 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 191–200, Nashville, TN, May 1994.
- [7] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In R. Valette, editor, *Application and Theory of Petri Nets 1994, Lecture Notes in Computer Science 815 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets, Zaragoza, Spain)*, pages 416–435. Springer-Verlag, June 1994.
- [8] B. Plateau, J.-M. Fourneau, and K. H. Lee. PEPS: a package for solving complex Markov models of parallel systems. In R. Puigjaner, editor, *Proc. 4th Int. Conf. Modelling Techniques and Tools*, pages 341–360, 1988.
- [9] B. Plateau and W. J. Stewart. Stochastic automata networks: product forms and iterative solutions. *Rapports de Recherche 2939*, INRIA, July 1996.