smart

stochastic model checking analyzer for reliability and timing

the SMART team ◇ smart@cs.ucr.edu

UC RIVERSIDE
UNIVERSITY OF CALIFORNIA

1994 Definition of a general framework for defining interacting models that exchange data

1996 First prototype: stationary solution of GSPNs, sparse matrices, iterative methods

1997 Kronecker solution algorithms, state-space storage requiring few bytes per state

1998 MDD-based state-space generation and storage

1998, 1999 MxD-based numerical solution algorithms

1999 Simulation with general distributions. Numerical solution with DPH distributions

2000 MDD+Kronecker-based approximations [SIGMETRICS 2000]

2001 Saturation-based symbolic state-space generation [TACAS 2001]

2001 Numerical solution for phased-delay SPNs

2001 First release

2002, 2003 Full CTL model checking, EVMDD-based CTL witnesses and counterexamples

2005 Fully general (non-Kronecker-based) symbolic state-space generation

2007 Bounded CTL model checking

Ongoing Full rewrite: performance and generality enhancements, integration of logical and stochastic analysis, unification of decision-diagrams libraries

The SMART package

Initially: SMART: Simulation and Markovian Analyzer for Reliability and Timing

- Stochastic models described as Petri nets with random firing times
- Efficient explicit state space generation
- Numerical transient or stationary solution of continuous-time or discrete-time Markov chains
- Batch means simulation of general discrete-state processes

Now: SMART: Stochastic Model-checking Analyzer for Reliability and Timing

- Implicit (MDD) state-space generation and CTL model checking
- Implicit (Kronecker, MxD, EVMDD) description of the underlying continuous-time Markov chain
- Numerical stationary solution of a Markov regenerative phase-delay Petri nets (PDPNs)
- Regenerative simulation of general PDPNs, with automatic detection of regenerations

What is SMART?

- A package integrating logic and stochastic modeling formalisms into a single environment (at the moment: DTMCs, CTMCs, and SPNs)
- Models expressed in different formalisms can be combined in the same study
- For the analysis of logical behavior:
  - explicit (BFS exploration) and implicit (symbolic MDD Saturation) state-space generation
  - symbolic CTL model checking
- For the study of the stochastic and timing behavior
  - explicit (sparse storage) and implicit (Kronecker) numerical solution approaches
  - numerical solution of semi-regenerative models
  - regenerative discrete-event simulation
- Easy integration of new formalisms and solution algorithms
- Over 100,000 lines of source C++ code

Declaration statements declare functions over some set of arguments

If there are no arguments, the function is constant (different from “non-random”, i.e., deterministic)

The type of the function and of its arguments must be defined

Definition statements declare functions, but also how to compute their value

Expression statements compute and print values

Can also have side-effects, such as redirecting the output or displaying additional information

Option statements modify the behavior of SMART

There are options to control the numerical solution algorithms (such as the precision or the maximum number of iterations), the verbosity level, etc.

Options statements appear on a single line beginning with “#”.

Compound `for` statements define arrays or repeatedly evaluate parametric expressions

Useful to explore how a result is affected by the modeling assumptions
(rate of an event, maximum size of a buffer, etc.)

Compound `converge` statements specify fixed-point iterations

Useful for approximate performance or reliability studies

SMART basic object types

SMART uses a strongly-typed declarative language with the following predefined types:

- `bool`: the values `true` or `false` — `bool c := 3 - 2 > 0;`
- `int`: integers (machine-dependent) — `int i := 12;`
- `bigint`: arbitrary-size integers — `bigint n := 12345678901234567890*2;`
- `real`: floating-point values (machine-dependent) — `real x := sqrt(2.3);`
- `string`: character-array values — `string s := "Monday";`

Composite types can be defined using the concepts of:

- *sets* — `{1..8,10,25,50}`
- *arrays* — `a[3][0.2]`
- *aggregates* — `p:t:3`

An object can be further modified by a stochastic nature:

- `const`: a non-stochastic quantity, the default
- `ph`: a random variable with discrete or continuous phase-type distribution
- `rand`: a random variable with arbitrary distribution
- `proc`: a random variable that depends on the state of a model at a given time

In addition, we can define models in various formalisms:

- `ctmc`: continuous-time Markov chains
- `dtmc`: discrete-time Markov chains
- `spn`: stochastic Petri nets

SMART objects are functions

Objects defined in SMART are functions, possibly recursive, and can be overloaded

```
real pi := 3.14;
bool close(real a, real b) := abs(a-b) < 0.00001;
int  pow(int b, int e)      := cond(e==1,b,b*pow(b,e-1));
real pow(real b, int e)     := cond(e==1,b,b*pow(b,e-1));
pow(5,3);                // computes and prints an integer, 125
pow(5.0,3);                  // computes and prints a real, 125
```

Arrays are declared with (possibly nested) for-loops ⇒ useful for large, repetitive structures

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

Facilities for fixed-point iterations are built-in ⇒ useful for numerical approximations

```
converge {
  real x guess 1.0;
  real y := f(x);
  real x := g(x,y);
}
```

Phase-type distributions

Discrete or continuous phase-type random variables can be managed numerically

The internal representation is an absorbing discrete-time or continuous-time Markov chain

Combining `ph` types produces a `ph` type if the distributions are closed under that operation

```
ph   int  X := geometric(0.7) + 2 * equilikely(0,5);
ph   int  T := min( 3 * X, 20 );
ph  real  a := erlang(4,5);
ph  real  b := min( 3 * a, expo(3.2) );
```

Mixing `ph int` and `ph real` results in a generally-distributed random variable

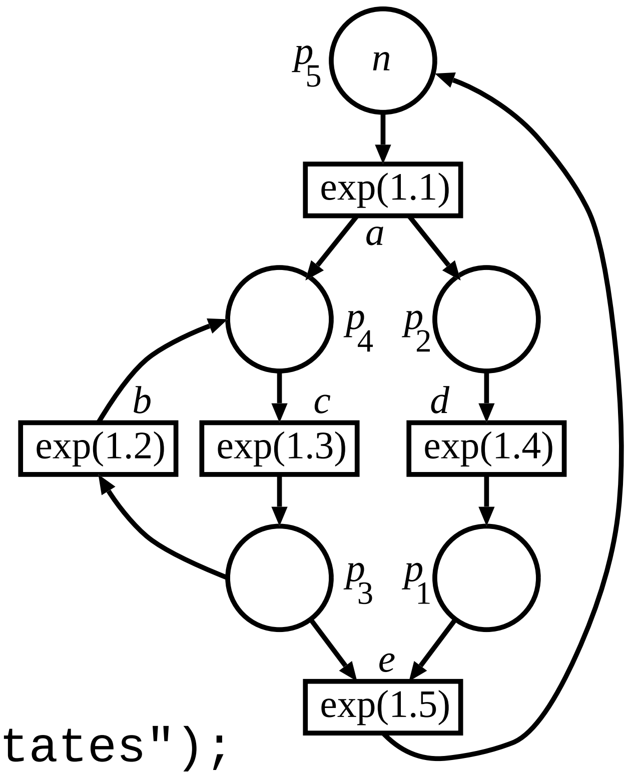
```
rand int  D := X - T;
rand real R := b + X;
```

A `rand` object can be manipulated only via Montecarlo methods (under development)

Example of model formalism: an SPN

```
spn net(int n) := {
  place p5, p4, p3, p2, p1; init(p5:n);
  trans a, b, c, d, e;
  arcs(p5:a,a:p4,a:p2,p4:c,c:p3,p3:b,
       b:p4,p2:d,d:p1,p1:e,p3:e,e:p5);
  firing(a:expo(1.1),b:expo(1.2),
       c:expo(1.3),d:expo(1.4),e:expo(1.5));
  bigint cnt := num_states(false);
  real speed := avg_ss(rate(a));
};

for (int n in {2..4}) {
  print("For n=",n," there are ",net(n).cnt," states");
  print(" and the throughput is ",net(n).speed,"\n");
}
```



The `for` loop outside the model produces the output

```
For n=2 there are 14 states and the throughput is 0.456948
For n=3 there are 30 states and the throughput is 0.553456
For n=4 there are 55 states and the throughput is 0.612828
```

```
spn phils(int N) := {
  for (int i in {0..N-1}) {
    place                idle[i],waitL[i],waitR[i],hasL[i],hasR[i],fork[i];
    partition(1+div(i,2):idle[i]:waitL[i]:waitR[i]:hasL[i]:hasR[i]:fork[i]);
    init(idle[i]:1, fork[i]:1);
    trans  Go[i],        GetL[i],        GetR[i],        Stop[i];
    firing(Go[i]:expo(1),GetL[i]:expo(1),GetR[i]:expo(1),Stop[i]:expo(1));
  }
  for (int i in {0..N-1}) {
    arcs(idle[i]:Go[i], Go[i]:waitL[i], Go[i]:waitR[i],
         waitL[i]:GetL[i], waitR[i]:GetR[i],
         fork[i]:GetL[i], fork[mod(i+1,N)]:GetR[i],
         GetL[i]:hasL[i], GetR[i]:hasR[i],
         hasL[i]:Stop[i], hasR[i]:Stop[i],
         Stop[i]:idle[i], Stop[i]:fork[i], Stop[i]:fork[mod(i+1, N)]);
  }
  bigint n_s := num_states(false);
};
# StateStorage MDD_SATURATION
print("The model has ", phils(read_int("N")).n_s, " states.\n");
```

Advanced features: State-space generation and storage

Using multiway decision diagrams (MDDs), SMART can generate extremely large state spaces:

Number of Philosophers	States $\|\mathcal{S}\|$	MDD Nodes		Memory (bytes)		CPU (secs)
		Final	Peak	Final	Peak	
100	4.97×10^{62}	197	246	30,732	38,376	0.04
300	1.23×10^{188}	597	746	93,132	116,376	0.13
1,000	9.18×10^{626}	1,997	2,496	311,532	389,376	0.45
3,000	7.74×10^{1880}	5,997	7,496	935,532	1,169,376	1.34

Symbolic CTL model checking queries are available in SMART:

```
stateset Reach := forward(initialstate);    // reachable
stateset NotAbs:= prev(potential(true));    // with successors
stateset Abs    := difference(Reach,NotAbs); // deadlocked
bool     dead   := neq(Abs,nostates);

stateset Good   := potential(e1);
stateset Bad    := potential(e2); // before reaching bad...
stateset Safe   := AU(Bad,Good);  // ...we are always good
stateset Stable:= EG(Good);       // there is an infinite good run
```

Advanced features: MxD-based numerical solutions

Using explicit data structures:

- If the SPN has an underlying DTMC or a CTMC
  - ☐ power method or uniformization for transient analysis
  - ☐ iterative methods (Jacobi, Gauss-Seidel, SOR) for stationary analysis
- If *synchronized* `ph int` are mixed with `ph real` the process is semi-regenerative
  - ☐ embedded DTMC + subordinated CTMCs solved to compute overall stationary measures

Using implicit data structures:

- The transition rate matrix for the underlying CTMC can be encoded
  - ☐ with matrix diagrams (MxDs), a generalization of Kronecker operators
- stationary solution is available using
  - ☐ iterative methods (Jacobi, Gauss-Seidel, SOR)

With implicit methods, can solve at least one order of magnitude larger models

Advanced features: An approximation scheme

- With MDDs for $\mathcal{S}$ and MxDs for $\mathbf{R}$, SMART can encode huge CTMCs
- The solution vector is the memory bottleneck
- SMART provides an approximation technique that uses the complete knowledge of $\mathcal{S}$ and $\mathbf{R}$
- K approximate aggregations based on the structure of the MDD representing $\mathcal{S}$
- A fixed-point iteration is used to break cyclic dependencies

Example: a Kanban model

N	$\|\mathcal{S}\|$	Worst relative error		CPU (sec)
		Average number of tokens	Transition throughput	
4	4.54×10^{5}	2.846%	-0.016%	0.47
5	2.55×10^{6}	2.557%	-0.074%	0.84
6	1.13×10^{7}	2.262%	-0.099%	1.38
7	4.16×10^{7}	2.032%	-0.097%	2.19
30	4.99×10^{13}	unknown	unknown	462.48
66	1.99×10^{17}	unknown	unknown	13,424.50

Planned development for SMART

MDD-based state-space generation in SMART is arguably the best for the class of targeted models

MxD-based exact and approximate CTMC solution in SMART are quite advanced

Evolve these capabilities into an integrated tool that can perform symbolic stochastic model checking

Many of the functionalities in SMART use and manipulate similar classes of decision diagrams

Modularize the SMART code, releasing a geneal MDD library