SMART

What is SMART?

- A package integrating logical and stochastic modeling formalisms into a single environment
- Multiple parametric models expressed in different formalisms can be combined in a study
- Easy addition of new formalisms and solution algorithms
- For the study of logical behavior:
  - explicit (BFS exploration) state-space generation [Tools97]
  - implicit (symbolic MDD Saturation) state-space generation [TACAS01,03, CHARME05]
  - Saturation-based CTL symbolic model checking [CAV03]
- For the study of stochastic and timing behavior
  - explicit (sparse storage) numerical solution of CTMCs and DTMCs
  - implicit (Kronecker, MxDs, EV*MDDs) numerical solution of CTMCs [INFORMSJC00,PNPM99]
  - structural-based approximations of large CTMC models [SIGMETRICS00]
  - explicit numerical solution of semi-regenerative models [PNPM01]
  - simulation of arbitrary models
  - regenerative simulation with automatic detection of regeration points [PMCCS03]
  - structural caching to speed up simulation [PMCCS03]

SMART Language

Strongly-typed, computation-on-demand, with five types of basic statements . . .

- **Declaration statements** declare functions over some arguments (including constants)
- **Definition statements** declare functions and specify how to compute their value
- **Model statements** define parametric models (declarations, specifications, measures)
- **Expression statements** print values (may have side-effects)
- **Option statements** modify the behavior of SMART (precision, verbosity level)

. . . and two compound statements that can be arbitrarily nested

- `for` statements define arrays or repeatedly evaluate parametric expressions
  Useful for parametric studies
- `converge` **statements** specify numerical fixed-point iterations
  Useful for approximate performance or reliability studies
  Cannot appear within the declaration of a model

SMART Types

Basic predefined types are available in SMART

- `bool`: the values `true` or `false` — `bool c := 3 - 2 > 0;`
- `int`: integers (machine-dependent) — `int i := -12;`
- `bigint`: arbitrary-size integers — `bigint i := 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

- aggregate: analogous to the Pascal “record” or C “struct” — `p:t:3`
- set: collection of homogeneous objects — `{1..8,10,25,50}`
- array: collection of homogeneous objects indexed by set elements

A type can be further modified a nature describing stochastic characteristics

- `const`: (the default) a non-stochastic quantity
- `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

Predefined formalism types can be used to define stochastic processes

Function declarations

Objects in SMART are functions, possibly recursive, that can be overloaded

```
real pi := 3.14;                         // an argument-less function
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);                                          // prints 125, int
pow(0.5,3);                                     // prints 0.125, real
```

Arrays

Arrays are declared using a `for` statement

An array's dimensionality is determined by the enclosing iterators

Indices along each dimension belong to a finite set

We can define arrays with `real` indices

```
for (int i in {1..5}, real r in {1..i..0.5}) {
  real res[i][r]:= MyModel(i,r).out1;
}
```

`res` is not a "rectangular" array of values:

- `res[1][1.0]`
- `res[2][1.0], res[2][1.5], res[2][2.0]`
- ...
- `res[5][1.0], res[5][1.5], ..., res[5][5.0]`

Numerical fixed-point iterations

The approximate solution of a model is often based on a heuristic decomposition

Submodels are solved in a fixed-point iteration using the `converge` statement

```
converge {
  real x guess 1.0;
  real y guess 2.0;
  real y := ModelA(x, y).measureY;
  real x := ModelB(x, y).measureX;
}
```

The `converge` statement iterates until the values of the variables declared in it (`x` and `y`) differ by less than ϵ in either relative or absolute terms, from one iteration to the next

These variables can be updated either individually or at the end of each iteration

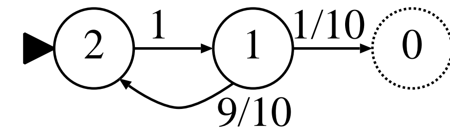
Option statements control both ϵ and the updating criterion

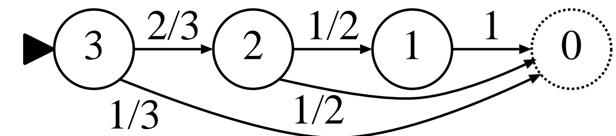
Random variables

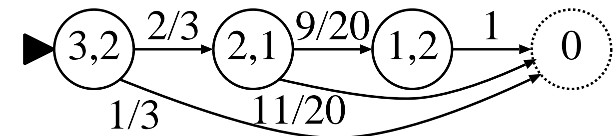
SMART can manipulate discrete and continuous phase-type distributions

Internally, SMART uses an absorbing DTMC or CTMC to represent a `ph int` or `ph real`

```
ph int  X := 2*geom(0.1);
ph int  Y := equilikely(1,3);
ph int  A := min(X,Y);
ph int  B := 3*X+Y;
ph int  C := choose(0.4:X,0.6:4*Y);
ph real x := expo(3.2);
ph real y := erlang(4,5);
ph real a := min(3*x,y);
```







To evaluate `avg(A)`, SMART builds the DTMCs for `X`, `Y`, and `A` and computes the MTTA of `A`'s

Mixing `ph int` and `ph real` or performing certain operations results in general distributions

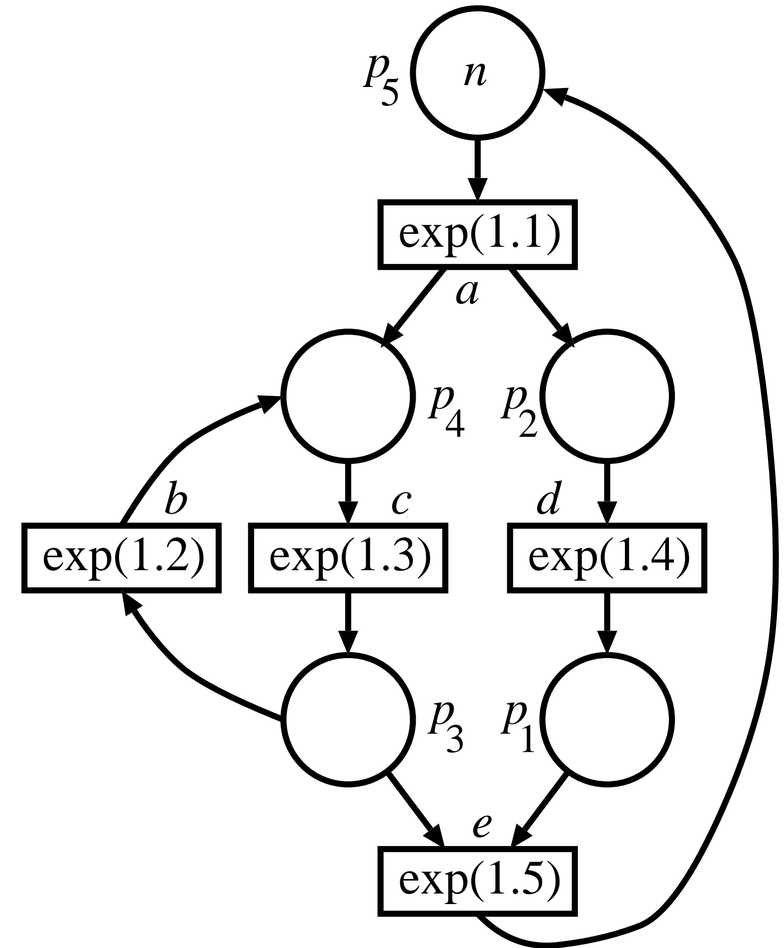
```
rand int  D := X-Y;
rand int  E := X*Y;
rand real F := x+X;
```

Modeling formalisms

A model is declared using formalism-specific types and functions

The type of underlying stochastic process is determined by the distributions in the model

```
spn Net(int n) := {
  place p5, p4, p3, p2, p1;
  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));
  init(p5:n);
  bigint n_s   := num_states(false);
  bigint n_a   := num_arcs(false);
  real speed   := avg_ss(rate(a));
  real e1      := avg_ss(tk(p1));
  real e2      := avg_ss(tk(p2));
  real e3      := avg_ss(tk(p3));
  real e4      := avg_ss(tk(p4));
  real e5      := avg_ss(tk(p5));
};
```



Measures are user-defined functions that specify some constant quantity of interest

Model measures

Only measures are accessible outside of the model definition block

The statements

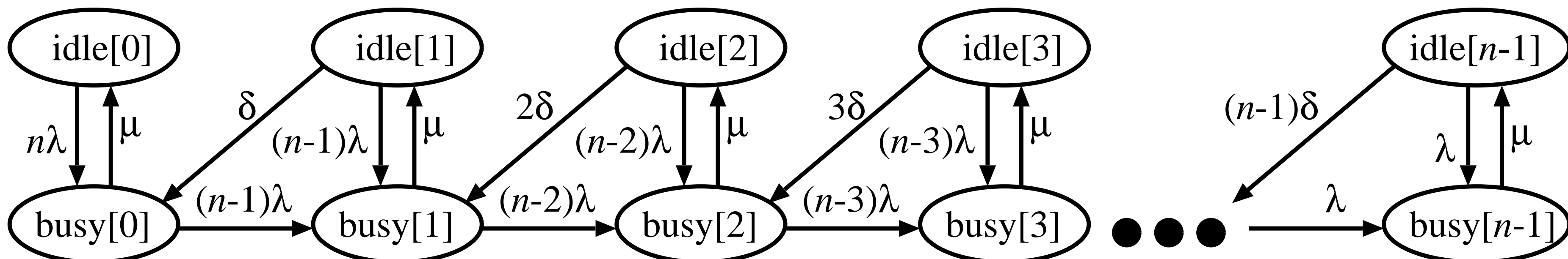
```
for (int n in {1..4}) {
  print("n=",n,": ",Net(n).n_s:2," states, ",
   Net(n).n_a:3," arcs, throughput = ",Net(n).speed,"\n");
}
print("n=1: E[tk(p5,p4,p3,p2,p1)] = (",Net(1).e5,",",
  Net(1).e4,",",Net(1).e3,",",Net(1).e2,",",Net(1).e1,")\n");
```

produce the output

```
n=1:  5 states,   8 arcs, throughput = 0.292547
n=2: 14 states,  34 arcs, throughput = 0.456948
n=3: 30 states,  88 arcs, throughput = 0.553456
n=4: 55 states, 180 arcs, throughput = 0.612828
n=1: E[tk(p5,p4,p3,p2,p1)] = (0.265952,0.469362,0.264686,0.2089
```

Arrays of objects


```
ctmc Lan(int n, real lambda, real mu, real delta) := {
  for (int i in {0..n-1}) {
    state idle[i], busy[i];  // "i" counts workstations in backoff
  }
  init (idle[0]:1.0);
  for (int i in {0..n-1}) {
    arcs(idle[i]:busy[i]:(n-i)*lambda, busy[i]:idle[i]:mu);
  }
  for (int i in {1..n-1}) {
    arcs(idle[i]:busy[i-1]:i*delta, busy[i-1]:busy[i]:(n-i)*lambda);
    real aux[i] := prob_ss(in_state(idle[i])|in_state(busy[i]));
    real a[i]   := cond(i>1,a[i-1]+i*aux[i],aux[1]);
  }
  real p := prob_ss(in_state(idle[0])|in_state(busy[0]));
};
real mu := 1.0; real delta := 0.1; real lambda := 0.04;
for (int n in {2..20}) {
  print("Prob[no backoff|n=",n,"]=",Lan(n,lambda,mu,delta).p,"\n");
  print("Avg[backoff|n=",n,"]=",Lan(n,lambda,mu,delta).a[n-1],"\n");
}
```



Computation on demand and reuse of computation

Consider an `spn` model `M` with

- transient measures `tm1` and `tm2`
- steady-state measures `sm1` and `sm2`
- and parameters
  - `n` affecting the state space
  - `l` affecting a transition rate
  - `r` appearing only in the expression for `sm1`

No computation takes places until it is required to perform an output (grouping heuristic)

```
M.sm1(n:=3,l:=0.2,r:=1.8);
```

the state space and the CTMC are built, the CTMC is solved, `M.sm1` and `M.sm2` are computed

```
M.sm1(n:=3,l:=0.2,r:=1.9);
```

only `M.sm1` is re-computed (a weighted sum of the steady-state probability vector entries)

```
M.sm1(n:=3,l:=0.4,r:=1.8);
```

the CTMC is re-built and re-solved, `M.sm1` and `sm2` are re-computed

```
M.sm1(n:=4,l:=0.2,r:=1.8);
```

NO COMPUTATION CAN BE REUSED

State-space generation and storage: model partition

SMART uses the `#StateStorage` option to choose between

- explicit techniques that store each state individually (AVL, SPLAY, HASHING)
  - no restrictions on the model
  - time and memory at least linear in the number of reachable states
- implicit techniques that employ MDDs to symbolically store sets of states (MDD_LOCAL_PREGEN, MDD_SATURATION_PREGEN, MDD_SATURATION)
  - require a partition of the model into submodels
  - normally much more efficient

A PN partition is specified by giving class indices (contiguous, positive integers) to places:

```
partition(2:p); partition(1:r); partition(1:t, 2:q, 1:s);
```

or by simply enumerating (without index information) the places in each class

```
partition(p:q, r:s:t);
```

State-space generation and storage: dining philosopers


```
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];
  }
  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");
```

Number of philosophers	States $\|\mathcal{S}\|$	MDD Nodes		Mem. (bytes)		CPU (secs)
		Final	Peak	Final	Peak	
100	4.97×10^{62}	197	246	30,732	38,376	0.04
1,000	9.18×10^{626}	1,997	2,496	311,532	389,376	0.45
10,000	4.26×10^{6269}	19,997	24,496	3,119,532	3,821,376	314.13

CTL model checking: operations

An object of type `stateset`, a set of global model states, is stored as an MDD

All MDDs for a model instance are stored in one MDD forest with shared nodes for efficiency

- **Atom builders:**
  - `nostates`, returns the empty set
  - `initialstate`, returns the initial state or states of the model
  - `reachable`, returns the set of reachable states in the model
  - `potential(`e`)`, returns the states of $\widehat{\mathcal{X}}$ satisfying condition e
- **Set operators:**
  - `union(`$\mathcal{P}, \mathcal{Q}$`)`, returns $\mathcal{P} \cup \mathcal{Q}$
  - `intersection(`$\mathcal{P}, \mathcal{Q}$`)`, returns $\mathcal{P} \cap \mathcal{Q}$
  - `complement(`$\mathcal{P}$`)`, returns $\widehat{\mathcal{X}} \setminus \mathcal{P}$
  - `difference(`$\mathcal{P}, \mathcal{Q}$`)`, returns $\mathcal{P} \setminus \mathcal{Q}$
  - `includes(`$\mathcal{P}, \mathcal{Q}$`)`, returns `true` iff $\mathcal{P} \supseteq \mathcal{Q}$
  - `eq(`$\mathcal{P}, \mathcal{Q}$`)`, returns `true` iff $\mathcal{P} = \mathcal{Q}$

CTL model checking: operations (cont.)

- **Temporal logic operators (future and past CTL operators):**
  - EX($\mathcal{P}$) and EXbar($\mathcal{P}$), AX($\mathcal{P}$) and AXbar($\mathcal{P}$)
  - EF($\mathcal{P}$) and EFbar($\mathcal{P}$), AF($\mathcal{P}$) and AFbar($\mathcal{P}$)
  - EG($\mathcal{P}$) and EGbar($\mathcal{P}$), AG($\mathcal{P}$) and AGbar($\mathcal{P}$)
  - EU($\mathcal{P}, \mathcal{Q}$) and EUbar($\mathcal{P}, \mathcal{Q}$), AU($\mathcal{P}, \mathcal{Q}$) and AUbar($\mathcal{P}, \mathcal{Q}$)
- **Execution trace output:**
  - EFtrace($\mathcal{R}, \mathcal{P}$), prints a witness for EF($\mathcal{P}$) starting from a state in $\mathcal{R}$
  - EGtrace($\mathcal{R}, \mathcal{P}$), prints a witness for EG($\mathcal{P}$) starting from a state in $\mathcal{R}$
  - EUtrace($\mathcal{R}, \mathcal{P}, \mathcal{Q}$), prints a witness for EU($\mathcal{P}, \mathcal{Q}$) starting from a state in $\mathcal{R}$
  - dist($\mathcal{P}, \mathcal{Q}$), returns the length of a shortest path from $\mathcal{P}$ to $\mathcal{Q}$
- **Utility functions:**
  - card($\mathcal{P}$), returns the number of states in $\mathcal{P}$ (as a bigint)
  - printset($\mathcal{P}$), prints the states in $\mathcal{P}$ (up to a given maximum)

SMART uses EV$^+$MDDs for efficient witness generation. . .

. . . and Saturation for efficient CTL model checking

Markov and non-Markov models

SMART provides standard numerical solutions to Markov models

- power method (DTMCs) and uniformization (CTMCs) for transient analysis
- iterative methods (Jacobi, Gauss-Seidel, SOR) for steady-state analysis

When all firing times are `ph int`, the underlying stochastic process is a DTMC

When all firing times are `ph real`, the underlying process is a CTMC

Mixing `ph int` and `ph real` timing delays in a model complicates matters

- an `spn` model may still be Markovian if the `ph int` transitions are synchronized
- SMART can
  - determine whether the resulting stochastic process is semi-regenerative
  - build an embedded DTMC and many subordinate CTMCs
  - solve for steady-state measures
- SMART employs a heuristic embedding with elimination

If the model is not semi-regenerative, SMART implements regenerative simulation for steady-state analysis, based on identifying hidden regenerative regions within the underlying GSSMC

Kronecker encoding of the Markov chain matrix

Limited to models where all events have `expo` or constant zero distributions

Quite effective in reducing the memory requirements

Like MDD methods, requires a user-specified partition

`#MarkovStorage` option:

- `SPARSE`
- `POTENTIAL_KRONECKER`
- `KRONECKER`
- `MATRIX_DIAGRAM_KRONECKER`

N	States $\lvert\mathcal{S}\rvert$	Nonzeros $\eta(\mathbf{R})$	Matrix diagram			Kronecker					Sparse	
			Mem. (bytes)	GS its	GS sec/it	Mem. (bytes)	GS its	GS sec/it	JOR its	JOR sec/it	GS its	GS sec/it
5	2,546,432	24,460,016	21,667	139	73	9,486	214	148	527	74	214	19
6	11,261,376	115,708,992	32,702	185	336	14,106	289	723	713	359	—	—
7	41,644,800	450,455,040	46,678	238	1,290	20,388	374	2,923	—	—	—	—

Kanban model

Where is SMART?

The executable is accessible on `eon`, feel free to copy but do not distribute

```
eon:~#38 ls ~ciardo/smart*
/home/csprofs/ciardo/smart2-i386-linux*
/home/csprofs/ciardo/smart-i386-linux*
/home/csprofs/ciardo/smart-intel-macOSX*
/home/csprofs/ciardo/smart-powerpc-macOSX*
/home/csprofs/ciardo/smart-windows.exe*
```

The manual and examples are available at
`http://www.cs.ucr.edu/~ciardo/SMART/index.html`

SMART is being constantly upgraded and modified, the manual might not always be up-to-date :-(