Experimenting with Stochastic Prolog as a Simulation Language

Enrico Oliva, Luca Gardelli, Mirko Viroli, Andrea Omicini

ALMA MATER STUDIORUM—Università di Bologna, Cesena, Italy

23° Convegno Italiano di Logica Computazionale (CILC)
10th, July 2008
Outline

1. Motivation
2. A Stochastic Simulation Framework in Prolog
3. Prolog Implementation
4. Comparing Stochastic Prolog with PRISM
   - Case Study: Collective Sorting
   - Collective Sorting in PRISM
   - Collective Sorting in Our Framework
5. Conclusions
Complexity in a computational system implies that it is essentially infeasible to fully predict the behaviour from its design in particular

- how small changes in some of the surrounding condition could lead system behaviour to diverging dynamics

Simulation is useful to preview the behaviour of complex computational systems in order to

- tune the design in the early phases of the engineering process
- study the diverging dynamics
Simulation Languages and Tools II

Simulation languages currently are based on stochastic extensions of some very low-level language

- Stochastic Pi-calculus
- generic stochastic process algebras for quantitative analysis such as EMPA
- verification-oriented tools like PRISM

Designer is forced to define a simulation using low-level mechanisms and tricks

- performance vs expressiveness of languages
Logic Languages such as Prolog

We believe logic languages could be used a basis for developing simulation languages,

- they are high-level, characterised by expressive and abstract constructs
- they are core languages with few constructs, paving the way towards formal analysis of probabilistic properties
- they have rule-based specification style
Our Framework

- is based on Prolog logic language
  - FOL representation of code and data
- allows the modelling of stochastic aspects according to the CTMC model
  - CTMC is basically an automata where the transitions are labelled by rate

The idea

The basic idea is to use stochastic operation towards FOL for simulation purpose. To do so, it is necessary to add
- clauses annotated by rate
- stochastic inference relation
Stochastic Logic Program

Definition

A stochastic logic program is a set of clauses of the form

\[ r : h \leftarrow b_1, \ldots, b_n \] and \[ h \leftarrow b_1, \ldots, b_n \]

where \( h \) and \( b_i \) are atomic formula, \( r \) a frequency value.

Labelled Clauses

Labelled clauses are expressed by the syntax

\[ \text{label}:h:-b_1,b_2,\ldots,b_n \]

where the label is a frequency value (or rate) \( r(X) \)

Example

\[ r(49.5) : \text{coin(head)}. \]
\[ r(49.5) : \text{coin(tail)}. \]
Stochastic Inference

Definition

Stochastic inference is expressed by the following algorithm:

1. Find the set $\mathcal{C}$ of labelled clauses whose head $h$ unifies with the current goal $b$
2. Calculate $r_{tot} = \sum_{i=1}^{n} r_i$, where $n$ is the cardinality of $\mathcal{C}$, where $r_i$ is the rate of $C_i \in \mathcal{C}$
3. Generate a random number $n_1 \in [0, 1]$
4. Evaluate the relation $\sum_{i=1}^{k-1} r_i \leq n_1 \cdot r_{tot} \leq \sum_{i=1}^{k} r_i$ in order to find $k$,
5. Next head to consider in the resolution process is the body of $C_k \in \mathcal{C}$

The inference relation is inspired by Gillespie’s algorithm based on frequency values.
From Prolog point of view we have introduced a *Probabilistic Cut*

- to make a probabilistic choice in the SLD resolution tree discarding the other possibilities

**Example**

\[
\begin{align*}
\text{r(49.5)} &: \text{coin(head)}.
\text{r(49.5)} &: \text{coin(tail)}.
\text{r(1)} &: \text{coin(manhole)}.
\text{toss}() &: \text{coin(X)}, \text{continuation(X)}.
\text{continuation(manhole)} &: !.
\text{continuation(_)} &: \text{toss()}.
\end{align*}
\]
Program Execution II

Execution of a program corresponds to the production of a stream of simulation events \( \text{event} \ (\text{State}, \text{Time}) \) where

- **State** is the Prolog goal to be solved yet
- **Time** is the elapsed time calculated by
  \[
  t = \frac{1}{rtot} \times \ln\left(\frac{1}{n_2}\right)
  \]
  where \( n_2 \) is a random number

Example

\[
T = \left[ \ldots, \text{event}(28, \text{coin(tail)}, 0.0102374), \text{event}(29, \text{coin(head)}, 0.0151458), \text{event}(30, \text{coin(head)}, 0.00339425), \text{event}(31, \text{coin(manhole)}, 0.00836014) \right]
\]
Stochastic framework is achieved, quite easily, by using meta-programming techniques in Prolog.

The meta-program, in addition to normal clauses resolution, has to make:

- stochastic choices without backtracking
- calculate the simulation steps and the simulation time

Possible introduction of specification facilities like $p(X)$

- in $p(X): C$ the frequency value of all clauses, match to $C$, sum to 1
- they are solved by a random selection according to its probability and causing no elapsed time
Prolog Implementation II

solve_trace/6 predicate realises most of the selection process

solve_trace(Goal, Time, Trace, Step, NS, TBack) :-
  findall(X : (Goal :- B), clause(X : Goal, B), L),
  % retrieve all clause match with goal
  not(empty_list(L)), !,
  sum_up(L, Tot),
  random(R),
  Tot1 is Tot * R,
  selection(L, Tot1, 0, Rate : (G)),
  % make stochastic selection
  arg(1, G, Goal), arg(2, G, Body),
  label_eval(Tot, Rate, Step, Goal, NS),
  % eval label for step & time
  solve_trace(Body, Time, Trace, NS, NS1, TBack1).
Comparing Stochastic Prolog with PRISM

PRISM as a reference language for the sake of comparison

- it is a paradigmatic case of how programming in such simulation/verification languages hardly scales with the system complexity (number of guards increases)

What we expect using Prolog is that it should be possible to provide a more compact and simpler specification
Outline

1. Motivation
2. A Stochastic Simulation Framework in Prolog
3. Prolog Implementation
4. Comparing Stochastic Prolog with PRISM
   - Case Study: Collective Sorting
   - Collective Sorting in PRISM
   - Collective Sorting in Our Framework
5. Conclusions
The **goal** of the algorithm is to distribute information across the tuple spaces, clustering similar information in the same space.

- Number of tuple spaces is equal to number of tuple kinds
- Each agent works at a certain frequency rate $r$
Agent Actions
- read a tuple uniformly over a tuple space
- remove a tuple from a tuple space
- insert a tuple into a tuple space

Agent Protocol
1. choose the source tuple space randomly
2. choose the destination tuple space randomly
3. uniformly read a tuple $S$ from the source tuple space
4. uniformly read a tuple $D$ from the destination tuple space
5. only if the tuple kinds are different, transfer a tuple of kind $D$ from the source to the destination
Collective Sorting is a self-organising algorithm to bring the system towards ordering independently of the initial configuration of tuples.

The algorithm evolves
- from initial situation TS1(20,20) and TS2(20,20)
- to final state TS1(40,0) and TS2(0,40)

We provide the specifications for the basic version of collective sorting using both the PRISM language and our framework.
Models in PRISM are specified using a state-based language based on Reactive Modules and it is able to represent DTMC, MDP and CTMC

- components of a system are specified using modules
- state is modelled as a set of finite-values variables
- modules composition and interaction is achieved in a process algebra style
The program is a sequence of transitions of the form

\[
\text{guard} \rightarrow \text{rate}_1: \text{update}_1 + \ldots + \text{rate}_n: \text{update}_n;
\]

where **Guard** is a boolean expression that leads to one of the next states (updates) according to the rates.

**PRISM solution**

- is not a very compact specifications tend to grow combinatorially with the number of tuple spaces \( n \)
- is encoded for a specific problem instance

**Example**

\( n > 5 \) specifications grow to several hundreds of rules
Stochastic Prolog specification works for all instance of the collective sort problem there is
- one rule with a rate (with head r(1):state(M)) for agent working rate
- four probabilistic selections
  - two for choosing source and destination tuple spaces \( \text{ts}(S) \) and \( \text{ts}(D) \)
  - two for reading tuples in them \( \text{cell}(S,KS,NKS),\text{cell}(D,KD,NKD) \)
Collective Sorting in Our Framework II

Example

A possible initial system with $n = 2$ is expressed by the following facts

\[\begin{align*}
p(80) &: \text{cell}(1,1,80). \\
p(50) &: \text{cell}(1,2,50). \\
p(30) &: \text{cell}(2,1,30). \\
p(20) &: \text{cell}(2,2,20). \\
p(1) &: \text{ts}(1). \\
p(1) &: \text{ts}(2). \end{align*}\]

- **Fact** $\text{cell}(\text{TS}, \text{TK}, N)$ stands for $N$ tuples of kind $\text{TK}$ residing in tuple space $\text{TS}$
- The state of the system is represented by a weighted matrix of cells
Collective Sorting $N \times N$
Specification in Our Framework
transfer(S,D,KS,KD) :-
    retract(p(N1):cell(S,KS,NKS)),
    retract(p(N2):cell(D,KD,NKD)),
    (N1 == 0,Nout1 is 0,Nout2 is N2;
    N1 =\= 0,exec(N1,N2,Nout1,Nout2)),
    assert(p(Nout1):cell(S,KS,Nout1)),
    assert(p(Nout2):cell(D,KD,Nout2)).
exec(TkS,TkD,TkSa,TkDa) :-
    TkDa is TkD + 1,
    (TkSa is TkS - 1).
r(1):state(M):- ts(S),state0(S).
state0(S) :- ts(D),state1(S,D).
state1(S,S) :- !,state0(S).
state1(S,D) :-
    cell(S,KS,_),
    cell(D,KD,_),
    state2(S,D,KS,KD).
state2(S,D,K,K) :- !,start.
state2(S,D,KS,KD) :-
    transfer(S,D,KS,KD),
    findall(R:(cell(X,Y,N)),R:(cell(X,Y,N)),M1),
    state(M1).
start:- ts(S),state0(S).
Main Predicates

- `transfer/4` tests whether the transfer is possible or not and it accordingly executes.
- `state/1`, `state1/2`, `state2/4` represent states during protocol execution (agent agenda); the `M` parameter contains the matrix with all probability values.
- `start/0` starts the simulation process.

To run the simulation we invoke the meta-goal:
```
solve_trace(start, 100, Trace)
```
Conclusions

- we propose a preliminary stochastic framework based on Prolog that allows to perform stochastic simulation directly from Prolog specifications
- we compare the specification of the collective sorting problem written in Stochastic Prolog with the same one expressed with the PRISM tool
- more thorough study is now required, from this preliminary study it is necessary to evaluate performance vs expressiveness
Future work

- study the semantic of the language
- extend the language in several way i.e. in order to support the concurrence operator
- translate from Stochastic Prolog specification to PRISM specification
- bring to the level of Prolog engine the stochastic features now implemented by meta-interpreter exploiting the flexibility of tuProlog
Thank you!

Questions?