Experimenting with Stochastic Prolog as a Simulation Language

Enrico Oliva, Luca Gardelli, Mirko Viroli, Andrea Omicini

Alma Mater Studiorum–Università di Bologna
Via Venezia, 52 - 47023 Cesena, Italy
{ enrico.oliva, luca.gardelli, mirko.viroli, andrea.omicini } @unibo.it

Abstract. While simulation is an established tool for scientific analysis, it is recently gaining more interest also in other contexts, such as software engineering. Hence, more and more attention is devoted to the development of suitable simulation languages (and tools), as well as to their exploitation in application development and run-time. As already experienced in the context of general-purpose programming languages, we envision future developments towards expressiveness, with performance issues becoming less and less relevant.
Along this direction, we propose a preliminary stochastic simulation framework developed on top of a logic programming language, called *Stochastic Prolog*: this framework allows us to run simulations directly from Prolog-based specifications. Our objective, in this work, is to put the basis for future research on logic stochastic language used for simulation purpose. In our approach Prolog clauses can be labelled with *rates* modelling temporal/probabilistic aspects. The main advantage of using Prolog is that it is significantly more expressive than other languages typically used in simulation, allowing complex specifications to be more easily encoded. In order to evaluate our framework, we compare it with the stochastic language defined by the PRISM tool, by discussing as case study the *collective sorting* problem, a decentralised sorting strategy for multiagent systems (MAS) inspired by behaviours observed in social insects.

1 Introduction

There is a growing interest in simulation languages and tools, and an increasing use of them throughout the engineering process. This is related to the increasing complexity of today computational systems in both the scientific and the engineering community: complexity implies that it is essentially infeasible to fully predict the behaviour of a system from its design, since small changes in some of the surrounding condition can lead system behaviour to diverging dynamics. Simulation is hence used as a means to preview the behaviour of complex computational systems without resorting to complete implementations, and possibly using such observations to tune the design in the early phases of the engineering process.

Many current simulation languages are based on stochastic extensions of some very low-level language, such as Stochastic π-calculus [1], generic stochastic process algebras for quantitative analysis such as EMPA [2], and verification-oriented tools like PRISM [3]. As the systems to be modelled and simulated tend to grow in complexity, such tools are more and more becoming inadequate. What typically happens is that a designer is forced to define a system behaviour using low-level mechanisms and tricks, much in the same way a programmer in the 70s would have used Assembler for building a complex application. One of the main reasons why this situation is persisting is that performance is still usually considered the primary issue of simulation, rather than expressiveness of the languages and frameworks.

Much in the same way programming languages evolved from low- to high-level, including more and more expressive abstractions and requiring true virtual machines in spite of a significant performance overhead, we envision a similar situation for simulation languages and frameworks. Although we do not deny the importance of performance in simulation, we believe it can and should be handled at the level of the underlying execution engine: the designer is to be freed from this problem, and has to leverage expressive languages for specifying system behaviour. Towards this direction, we believe logic languages such as Prolog can be used as a basis for developing suitable simulation languages, since: (i) they are high-level, characterised by expressive and abstract constructs; (ii) they are still core languages with few constructs, paving the the way towards formal analysis of probabilistic properties; and (iii) they allow to easily exploit the rule-based specification style—which is quite common in the context of distributed systems.

In this article we introduce a preliminary stochastic simulation framework on top of the Prolog language, called *Stochastic Prolog*: to this purpose we follow a probabilistic logic approach labelling clauses with rates. As a first step, we define a base stochastic extension of first order logic in order to follow the framework of Continuous Time Markov Chains CTMC [4]: rates define transition frequency according to an exponential distribution, and in the case of race conditions they implicitly define the probability for an action to be scheduled next.

As a second step, we evaluate our Stochastic extension of Prolog applying it to the case of collective sorting, a decentralised sorting algorithm inspired by *social insects behaviours* [5], and then compare it to the PRISM tool, taken here as a reference language for simulation. In this case study, a multiagent system (MAS) is composed of agents and tuple spaces: agents are in charge of moving tuples from one space to another according to their type, until reaching full ordering [6, 7]. We model a self-organising solution to the collective sorting problem using both our framework and the PRISM tool, then compare the expressiveness of the two solutions, finally showing how the framework proposed here can better scale with system complexity.

The remainder of the article is structured as follows: in Section 2 we describe the simulation framework; in Section 3 we provide a brief description of the case study and provide specifications both for the PRISM tool and our framework; in Section 4 we discuss related works and finally conclude.

2 A Stochastic Simulation Framework in Prolog

We describe a framework to run stochastic simulations based on Prolog logic language in order to exploit the expressive power of a declarative language. Prolog is very useful because of the uniform representation of code and data, both encoded as first-order logic (FOL) clauses, which makes writing (meta-)interpreters quite easy [8]. The framework allows the modelling of stochastic aspects, in particular according to the CTMC model, an important class of stochastic processes widely used in practice both to determine system performance and to predict system behavior.

As a CTMC is basically an automata where transitions from one state to another are labelled with rates, in our framework it is used to enhance a logic program with rates driving the goal resolution process—the label-based approach is commonly used to introduce stochastic aspects in formal languages, e.g. the Stochastic π Calculus [1] and also in logic programming [9].

The basic idea of our framework is to use stochastic operation towards FOL for simulation purpose. To this end it is necessary to add a couple of features to FOL: 1) clauses annotated by *rate*, 2) stochastic inference relation.

Definition 1. *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.*

Syntactically, in a Stochastic Prolog program the set of labelled clauses are expressed using the following notation: `label:h:-b1,b2,..bn.`. Namely, each stochastic clause is a standard clause with a prefix `label`, which is a frequency value (or rate) `r(X)`, where `X` should be a ground number.

The semantic of a stochastic clause is defined by a possible world semantic of a Herbrand interpretation of the classical underlying first order language. Thus a C clause defined $C = r : h \leftarrow b$ is true in a possible world W denoted as $W \models C$ if and only if C is true in the Herbrand interpretation belonging to W. The values indicated in C clauses are the frequency rates that are used to choose next ground instance $C\theta$ and proceed with the resolution derivation process.

The stochastic inference relation for labelled clauses is inspired by Gillespie's algorithm [10] based on frequency values.

Definition 2. *The 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 $C_k \in \mathcal{C}$

More practically, the operational semantics of stochastic Prolog is expressed by the meta-interpreter in Figure 1. From a Prolog point of view we have introduced a *probabilistic cut*. The normal cut tells the the interpreter to remove a choice point from the stack, eliminating a branch of solution tree and limiting the backtracking. Instead, a *probabilistic cut* tells the meta-interpreter to make a probabilistic choice in the SLD resolution tree, discarding the other possibilities.

Each execution of a program always corresponds to the production of a single simulation trace, formed by a stream of simulation events each being a couple `event (State,Time)`—where `State` is the Prolog goal to be solved yet, and `Time` is the elapsed time. The goal resolution process depends on the kind of predicate involved, briefly: standard predicates are solved as in Prolog, i.e., the top-most clause is selected and others are reconsidered during backtracking; predicates with rates are solved by randomly selected a clause (the higher the rate, more likely they are selected) and causing an elapsed time according to the exponential distribution (see below). The time t of each event in the simulation is calculated only after the execution of a stochastic clause, and proportionally to the total rate with an exponential distribution $t = (1/rtot) * (\ln(1/n_2))$ where n_2 is a new random number $\in [0, 1]$.

As a basic example, let us consider a process in which each time unit a fair coin is tossed a hundred of times, but with a 1% probability of failing, in which case the process is to be stopped. Figure 1 lists the code for simulating this system.

Example 1. Simple Stochastic Prolog Program for simulating a coin toss with possible failures, and a possible trace. The predicate `coin/1` expresses the fact that the probability of tossing to head or tail is 49.5%, and probability of failure is 1% (causing a failure of the process): then since the sum of rates is 100, each tossing will take place at an average 0.01 of elapsed time. Predicate `toss/0` drives the process: after solving `coin/1`, if `manhole` is found the process is stopped, otherwise it is reiterated again.

```
r(49.5):coin(head).
r(49.5):coin(tail).
r(1):coin(manhole).

toss() :- coin(X), continuation(X).
continuation(manhole) :- !.
continuation(_) :- toss().

---------------------------
?- solve_trace(toss(), 100, T).
yes.
T=[...,
event(28, coin(tail), 0.0102374),
event(29, coin(head), 0.0151458),
```

```
event(30, coin(head), 0.00339425),
event(31, coin(manhole), 0.00836014)]
```

Note that the result of the simulation is provided by a meta-interpreter `solve_trace`, accepting as input the initial state of the stochastic program (i.e. the goal to be solved), a maximum elapsed time, and returning as output a sequence of events. In order to ask the solution we hence try demonstrating goal `solve_trace(+Goal,+Time, -Trace)`. The output is a list of events, each with its own elapsed time. Such a trace can easily be passed to a drawing program for generating charts.

2.1 A Prolog Implementation

The first implementation of Stochastic Prolog is easily achieved by exploiting the well-known meta-programming capabilities of Prolog. Our meta-program structure in Figure 1 is similar to the standard Prolog interpreter *solve* [8]. Notice that our meta-interpreter, in addition to normal clauses resolution, has to make stochastic choices without backtracking, and calculate the simulation steps and the simulation time. Currently, all the functionality required by Stochastic Prolog are developed by changing the standard resolution process of Prolog through the meta-interpreter. The resolution is directed by label value and label type. To introduce the label syntax `label:clause`, we define a new operator "`:`" through definition `:-op(500, yfx, ':')`. Rate labels are expressed by using label predicates `r(X)`. Technically, backtracking of labelled clauses is disabled by the meta-interpreter using cuts operator. We can introduce also some specification facilities like `p(X)` label which means that all `p(X):C` unifiable clauses have the frequency value that sum to 1. They are solved by randomly selecting a clause (according to its probability) and causing no elapsed time.

Figure 1 shows the top level predicate of the interpreter, which is called to start the simulation. Last clause of `solve_trace/6` predicate realises most of the selection process: all clauses matching `Goal` and with some label `X` are retrieved combining built-in predicates `findall/3` and `clause/2`. The stochastic choice is made in `selection/4` whose details are not reported for the sake of brevity. The system has been tested with SWI-Prolog platform [11], but it could have been tested with any other standard Prolog interpreter because only ISO standard predicates are used.

3 Comparing Stochastic Prolog with PRISM

In order to evaluate our Stochastic Prolog framework, we consider the case of *collective sorting* as described in [6, 7]. We first describe the problem, then we model a self-organising solution according to the agent paradigm. We provide a specification for both the PRISM tool and our Stochastic Prolog framework.

In this paper, we considered PRISM as a reference language for the sake of comparison, for it is a paradigmatic case of how programming in such simulation/verification languages hardly scale with the system complexity. Since a

```
solve_trace(Goal,Time,Trace):-
  solve_trace(Goal,Time,Trace,0,NS,TBack).
solve_trace(_,Time,_,Time,_,T):-!,write('STOP ').
solve_trace(true,_,_,_,_,_):-!.
solve_trace((Goal1;Goal2),Time,Trace,Step,NS,TBack):-
  !,(solve_trace(Goal1,Time,Trace,Step,NS,TBack);
  solve_trace(Goal2,Time,Trace,Step,NS,TBack)).%Disjuction

solve_trace((Goal1,Goal2),Time,Trace,Step,NS,TBack):-
  !,
  solve_trace(Goal1,Time,Trace,Step,NS,TBack),
  NS < Time,
  solve_trace(Goal2,Time,Trace,NS,NS1,TBack).%Conjuction

solve_trace(Goal,Time,Trace,Step,NS,TBack):-
  builtin(Goal),!,
  call(Goal),NS = Step. %System predicate

solve_trace(Goal,Time,Trace,Step,NS,TBack):-
  clause(Goal,B),
  solve_trace(B,Time,Trace,Step,NS,TBack).%Normal predicate

solve_trace(Goal,Time,Trace,Step,NS,TBack):-
  findall(X:(Goal:-B),clause(X:Goal,B),L),
  not(empty_list(L)),!,
  sum_up(L,Tot),
  random(R),           % 0-1 random number
  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).%Label predicates
```

Fig. 1. The meta-interpreter code for predicates `solve_trace/3` and `solve_trace/6`

PRISM specification defines a transition system, we expect the specification to rapidly grow with the system complexity (number of guards increases): this is often the case when dealing with non-trivial algorithms, such as the collective sorting. What we expect using a more expressive language such as Prolog, is that it should be possible to provide a more compact and simpler specification.

3.1 Case Study: Collective Sorting

In this section, we describe a particular decentralised sorting strategy called *collective sorting*: the solution to this problem has been inspired by sorting behaviours displayed by social insects [5]. Collective sorting has potential applications to both distributed software and physical environments, although,

from now on, we refer to a specific computational scenario based on the agent paradigm.

We consider a distributed environment consisting of Linda-like tuple spaces hosting tuples belonging to a limited set of kinds. The goal of the algorithm is to distribute information across the tuple spaces, clustering similar information in the same space while separating different kinds of information—see Figure 2. In particular, we consider here the case where the number of tuple spaces is equal to the number of tuple kinds—indeed, this is quite a general situation, in which tuple kinds are actually groups identified at design-time, after the number of tuple spaces is known.

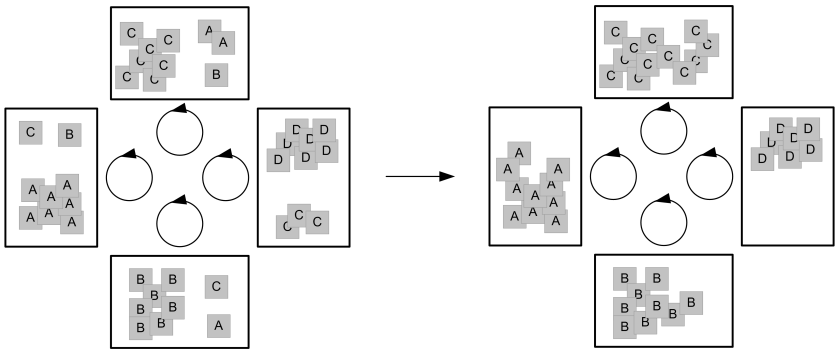

Fig. 2. Dynamics of collective sorting. Given the system state on the left-hand side, the algorithm eventually drive the system to the state displayed on the right-hand side

Each agent is responsible for keeping its own tuple space ordered. It adopts a specific interaction protocol to evaluate the movement of some tuple: it sleeps for a certain time, then wakes up and interact again. Doing so, it will work at a certain rate (or frequency) r—where the time for executing the protocol is supposed to be much smaller than $1/r$. We proposed a solution to the collective sorting problem for tuple-space-based systems, where each time unit an agent performs a local observation (in its own tuple space) and a remote observation (another tuple space chosen randomly), and evaluates whether a tuple has to be moved away based on the outcomes [7]. The possible actions of such an agents over the tuple spaces are hence the following:

- read a tuple "uniformly" over a tuple space—that is, each tuple matching the request has the same probability of being picked (hence bigger groups of a same kind are likely read more often)
- remove a tuple from a tuple space
- insert a tuple in to a tuple space

Note that from a simulation point of view, having n agents each with its own working rate r is equivalent to having just one agent with working rate $n * r$,

each time behaving like one of the n agents chosen probabilistically—each agent has the same probability of being chosen. Hence, we consider the following single 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

This self-organising algorithm is shown to bring the system towards ordering independently of the initial configuration of tuples [7]—the ordering is yet complete only by slightly changing the algorithm as shown in [6], but this issue is of no interest here. The chart displayed in Figure 3 represents the dynamics of collective sorting starting from initial situation TS1(20,20) and TS2(20,20): the algorithm evolves the system towards the final state TS1(40,0) and TS2(0,40). It is worth noting that the dual solution may have occurred as well, and it is not possible to foretell where a specific cluster will appear.

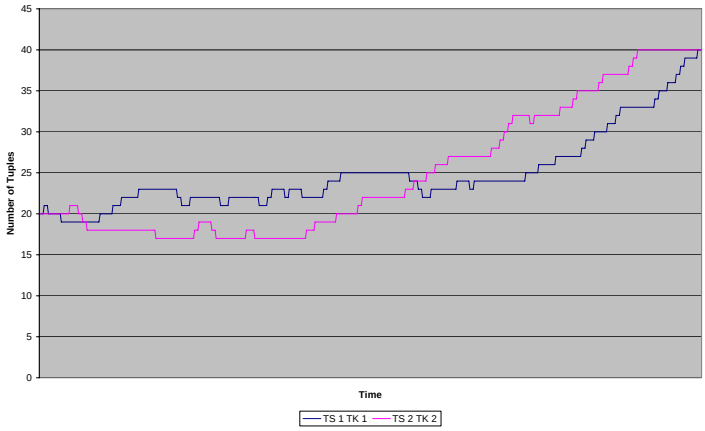

Fig. 3. The dynamics of the collective sorting, specifically tuple kind 1 in tuples space 1 and tuple kind 2 in tuple space 2

As a case to evaluate the simulation framework developed, we provide the specifications for the basic version of collective sorting using both the PRISM language and our framework.

3.2 Collective Sorting in PRISM

Among the many commercial and academic simulation frameworks, we have chosen the PRISM tool [3] as a comparison: other than allowing simulation, it offers a simple specification language and the possibility to perform formal analysis. In general, it is a paradigmatic case of low-level stochastic language.

In PRISM, models are specified using a state-based language based on Reactive Modules and is able to represent either probabilistic, non-deterministic and stochastic systems using, respectively, Discrete-Time Markov Chains (DTMC), Markov Decision Processes (MDP) and Continuous-Time Markov Chains (CTMC). Components of a system are specified using modules and the state is modelled as a set of finite-values variables: furthermore, modules composition and interaction is achieved in a process algebra style.

According to the agent agenda defined in the previous section, in Figure 4 we provide the PRISM commented specifications for collective sorting, considering the basic case of two tuple spaces and two tuple kinds. The first part of the specification consists in constants, variables and formulas: the behaviour of the agent is completely specified within the block `module agent .. endmodule`. Each step in the agent agenda is encoded using one or more transitions, specified using the notation `[] guard -> rate_1 : update_1 + ... _ + rate_n : update_n;`: the guard is a boolean expression that, when verified, leads to one of the next states specified in the updates according to the specified rates. The algorithm is articulated in four steps:

Step 0 choose the source and the destination tuple spaces
Step 1 randomly draw a tuple from source tuple space and observe its kind
Step 2 randomly draw a tuple from destination tuple space and observe its kind
Step 3 depending on the tuple kinds decide whether to move the tuple or not

It is worth noting that several transitions occur with rate value equals `decision`: this is an arbitrary large constant used to model a decisional process, i.e. a process having a duration small in comparison with the other durations of the system. It is easy to recognise that the PRISM language does not produce very compact specifications: indeed, specifications tend to grow combinatorially with the number of tuple spaces n (which directly influences the variables involved and the number of guards). For instance, even with $n > 5$ the specifications grows to several hundreds of rules: it is clear then that such a kind of language can be used only as a low-level one, whereas more expressive languages are instead required.

3.3 Collective Sorting in Stochastic Prolog

According to the agent agenda previously defined, we provide a Stochastic Prolog specification for collective sorting—see Figure 5. While the PRISM specification has been encoded for a specific instance, the prolog works for all instances of collective sorting problem, where the number N of tuple kinds and tuple spaces

can be any greater than 1. This is a clear expressiveness advantage over the PRISM language, where specifications grow very quickly.

A possible initial system with $n = 2$ is expressed by the following facts with a probability label:

```
p(80):cell(1,1,80).
p(50):cell(1,2,50).
p(30):cell(2,1,30).
p(20):cell(2,2,20).
p(1):ts(1).
p(1):ts(2).
```

Fact `cell(TS,TK,N)` stands for N tuples of kind TK residing in tuple space TS: we hence represent the system configuration with a matrix of weights. This choice ensures that while reading a tuple on a tuple space, it is more likely to get one of a bigger group. Moreover, we have two facts of the kind `ts(X)` with same probability, used to pick a tuple space probabilistically. These facts are considered as input state for the simulation, and could by changed throughout via assertion and retraction as usual in Prolog.

In the general sense the overall system could be considered like an instance of a CTMC. In the whole specification there is only one rule with a rate (with head `r(1):state(M)`), responsible for setting the agent working rate to 1, and four probabilistic selections: two for choosing source and destination tuple spaces `ts(S)` and `ts(D)`, and two for reading tuples in them `cell(S,KS,NKS),cell(D,KD,NKD)`.

There are three main predicates that are involved in the problem specification: `transfer/4`, `state/1` and `start/0`. The predicate `transfer(+S,+D,+KS,+KD)` tests whether the transfer is possible or not and it accordingly executes the transfer—it is worth noting that this modifies the rates of facts. There, `S` and `D` are respectively tuple space source and destination identifiers, `KS` and `KD` are tuple kind of source and tuple kind of destination. Predicates `state/1`, `state0/1`, `state1/2`, `state2/4` represent states during protocol execution, and implement the CTMC: The `M` parameter contains the current state of system i.e. the matrix with all probability values, which is reified as argument for being tracked in the simulation trace. `start` predicate is called to start the simulation process.

In order to run a simulation, we invoke the meta-goal `solve_trace(start, 100, Trace)`, which produces as output the trace of the next 100 system states with corresponding elapsed time, e.g. as follows:

```
event(44, state([24:cell(2, 2), 50:cell(1, 2),
                 26:cell(2, 1), 80:cell(1, 1)], 3), 0.767398)
event(45, state([50:cell(1, 2), 26:cell(2, 1),
                 23:cell(2, 2), 81:cell(1, 1)], 3), 1.54405)
event(46, state([50:cell(1, 2), 26:cell(2, 1),
                 80:cell(1, 1), 24:cell(2, 2)], 3), 0.455535)
event(47, state([80:cell(1, 1), 24:cell(2, 2),
```

```
                49:cell(1, 2), 27:cell(2, 1)], 3), 0.834995)
event(48, state([24:cell(2, 2), 49:cell(1, 2),
                26:cell(2, 1), 81:cell(1, 1)], 3), 0.119659)
```

In order to verify the generality of the Prolog specifications we evaluated the collective sorting in the instance with 3 tuple spaces and hence a 3×3 matrix, as in Example 2.

Example 2. collective sorting simulation with $N = 3$

```
p(80):cell(1,1,80). p(50):cell(1,2,50). p(10):cell(1,3,10).
p(30):cell(2,1,30). p(20):cell(2,2,20). p(20):cell(2,3,20).
p(70):cell(3,1,70). p(60):cell(3,2,60). p(10):cell(3,3,10).
p(1):ts(1). p(1):ts(2). p(1):ts(3).

?- solve_trace(start,100,T).
...
event(97, state([
p(11):cell(1,3,11), p(55):cell(3,2,55), p(8):cell(3,3,8),
p(32):cell(2,1,32), p(75):cell(1,1,75), p(22):cell(2,2,22),
p(27):cell(2,3,27), p(51):cell(1,2,51), p(69):cell(3,1,69)]),
1.85357)

event(98, state([
p(11):cell(1,3,11), p(55):cell(3,2,55), p(8):cell(3,3,8),
p(75):cell(1,1,75), p(22):cell(2,2,22), p(27):cell(2,3,27),
p(69):cell(3,1,69), p(50):cell(1,2,50), p(33):cell(2,1,33)]),
0.959323)

event(99, state([
p(11):cell(1,3,11), p(55):cell(3,2,55), p(8):cell(3,3,8),
p(22):cell(2,2,22), p(69):cell(3,1,69), p(50):cell(1,2,50),
p(33):cell(2,1,33), p(26):cell(2,3,26), p(76):cell(1,1,76)]),
0.366512)
STOP
```

4 Conclusion

In this article, we propose a stochastic framework based on Prolog that allows to perform stochastic simulation directly from Prolog specifications. The proposed extension, called Stochastic Prolog, follows the same approach used in other languages such as Stochastic π-calculus [1]. Clauses are labelled either with rates or probabilities, respectively modelling stochastic and probabilistic aspects: specifically rates define action duration according to an exponential distribution, while next event scheduling is based on the Gillespie's algorithm [10].

In the Logic Programming literature some concepts related to stochastic programming have already been introduced, but to the best of our knowledge they are not meant to target stochastic simulation. Muggleton [9] defines *stochastic logic programming*, where a stochastic logic program P is a set of labelled clause $p : C$ with probability $p \in [0, 1]$, and where for each symbol q in P the probability label for all clauses with q head sum to 1. The meaning of the label in Muggleton is probabilistic but it is not related to time or rate of execution. Therefore, our work is apparently the first one putting together timing and probabilistic aspects into Prolog, and then also the first to experiment it in the context of simulation of complex computational systems.

In order to evaluate our simulation framework we compare the specification of the collective sorting problem written in Stochastic Prolog with the same one expressed with the PRISM tool. Thus, we show that Stochastic Prolog specifications are more compact even with the simplest problem instances: while Stochastic Prolog specifications are not affected by the instance size, PRISM specifications tend to quickly grow up to hundreds of transition rules even with the simple case of $N = 4$—4 tuple spaces and 4 tuple kinds. Although a more thorough study is now required, from this preliminary study it is clear that our approach generally provides for much more expressiveness, and could hence be used as an effective specification tool for system designers that need to simulate complex applications.

Further explorations will first involve a semantic study of the language, coding more case studies to better evaluate it, then extend the language in several ways, e.g. supporting concurrency operators as commonly found in process algebras.

References

1. Priami, C.: Stochastic π-calculus. The Computer Journal **38**(7) (1995) 578–589
2. Aldini, A., Bernardo, M., Gorrieri, R., Roccetti, M.: Comparing the QoS of Internet audio mechanisms via formal methods. ACM Trans. Model. Comput. Simul. **11**(1) (2001) 1–42
3. University of Birmingham: The PRISM probabilistic model checker. Version 3.1.1 and documentation available online at http://www.prismmodelchecker.org (September 2007)
4. Kulkarni, V.G.: Modeling and analysis of stochastic systems. Chapman & Hall, Ltd., London, UK, UK (1995)
5. Deneubourg, J., Goss, S., Franks, N., , C. Detrain, A.S.F., Chrétien, L.: The dynamics of collective sorting: Robot-like ants and ant-like robots. In Meyer, J.A., Wilson, S.W., eds.: From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior. Classics. MIT Press, Cambridge, MA, USA (February 1991) 356–363
6. Viroli, M., Casadei, M., Gardelli, L.: A self-organising solution to the collective sort problem in distributed tuple spaces. In: Proceedings of the 2007 ACM Symposium on Applied Computing (SAC 2007), Seoul, Korea, ACM (11–15March 2007) 354–359 Special Track on Coordination Models and Languages.

7. Casadei, M., Gardelli, L., Viroli, M.: Simulating emergent properties of coordination in Maude: the collective sort case. Electronic Notes in Theoretical Computer Science **175**(2) (June 2007) 59–80 5th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2006).
8. Sterling, L., Shapiro, E.: The Art of Prolog: Advanced Programming Techniques. 2nd edn. Logic Programming. MIT Press, Cambridge, MA, USA (1994)
9. Muggleton, S.: Stochastic logic programs. In De Raedt, L., ed.: Proceedings of the 5th International Workshop on Inductive Logic Programming, Department of Computer Science, Katholieke Universiteit Leuven (1995) 29
10. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. The Journal of Physical Chemistry **81**(25) (1977) 2340–2361
11. SWI-Prolog: Version 5.6.40. Documentation available online at http://www.swi-prolog.org (September 2007)

```
ctmc
const int MAX=40;
const int decision = 1000000;
global t1k1 : [0..MAX] init 20;
global t1k2 : [0..MAX] init 20;
formula chooset1k1 = t1k1/content1;
formula chooset1k2 = t1k2/content1;
formula content1 = t1k1+t1k2;
global t2k1 : [0..MAX] init 20;
global t2k2 : [0..MAX] init 20;
formula chooset2k1 = t2k1/content2;
formula chooset2k2 = t2k2/content2;
formula content2 = t2k1+t2k2;

module agent
//Variables encoding agent internal state
step : [0..3] init 0;
ts : [1..2];
td : [1..2];
s  : [1..2];
d  : [1..2];
//Randomly choose source and destination tuple spaces
[] step = 0 & content1 != 0 & content2 != 0 ->
    decision : (ts' = 1) & (td' = 2) & (step'=1) +
    decision : (ts' = 2) & (td'=1) &(step'=1);
// Choose source tuple kind
[] step = 1 & ts = 1 & content1 != 0 & content2 != 0 ->
    chooset1k1 * decision : (s' = 1) & (step' = 2) +
    chooset1k2 * decision : (s' = 2) & (step' = 2);
[] step = 1 & ts = 2 & content1 != 0 & content2 != 0 ->
    chooset2k1 * decision : (s' = 1) & (step' = 2) +
    chooset2k2 * decision : (s' = 2) & (step' = 2);
// Choose destination tuple kind
[] step = 2 & td = 1 & content1 != 0 & content2 != 0 ->
    chooset1k1 * decision : (d' = 1) & (step' = 3) +
    chooset1k2 * decision : (d' = 2) & (step' = 3);
[] step = 2 & td = 2 & content1 != 0 & content2 != 0 ->
    chooset2k1 * decision : (d' = 1) & (step' = 3) +
    chooset2k2 * decision : (d' = 2) & (step' = 3);
[] step = 3 & s = d ->
    decision : (step'=0);
//Tuple space source 1, Tuple space destination 2
[] step = 3 & ts = 1 & td= 2 & s = 1 & d = 2 &
   t1k2 > 0 & t2k2 < MAX ->
    1.0 : (step'=0) & (t1k2' = t1k2 - 1) & (t2k2'=t2k2 + 1);
[] step = 3 & ts = 1 & td= 2 & s = 1 & d = 2 &
   t1k2 = 0 ->
    1.0 : (step'=0);
[] step = 3 & ts = 1 & td= 2 & s = 2 & d = 1 &
   t1k1 > 0 & t2k1 < MAX ->
    1.0 : (step'=0) & (t1k1' = t1k1 - 1) & (t2k1'=t2k1 + 1);
[] step = 3 & ts = 1 & td= 2 & s = 2 & d = 1 &
   t1k1 = 0 ->
    1.0 : (step'=0);
//Tuple space source 2, Tuple space destination 1
[] step = 3 & ts = 2 & td= 1 & s = 1 & d = 2 &
   t2k2 > 0 & t1k2 < MAX ->
    1.0 : (step'=0) & (t2k2' = t2k2 - 1) & (t1k2'=t1k2 + 1);
[] step = 3 & ts = 2 & td= 1 & s = 1 & d = 2 &
   t2k2 = 0 -> 1.0 : (step'=0);
[] step = 3 & ts = 2 & td= 1 & s = 2 & d = 1 &
   t2k1 > 0 & t1k1 < MAX ->
    1.0 : (step'=0) & (t2k1' = t2k1 - 1) & (t1k1'=t1k1 + 1);
[] step = 3 & ts = 2 & td= 1 & s = 2 & d = 1 &
   t2k1 = 0 -> 1.0 : (step'=0);
endmodule
```

Fig. 4. Collective sorting specifications in the PRISM language

```
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).
```

Fig. 5. Collective sorting $N \times N$ specification in Stochastic Prolog