# Modeling Scientific Theories as PRISM Programs

**Taisuke SATO**[1]

**Abstract.** **PRISM** is a new type of symbolic-statistical modeling language which integrates logic programming and learning seamlessly.[2] It is designed for the symbolic-statistical modeling of complex phenomena such as genetics and economics where logical/social rules and uncertainty interact, thus expected to be a valuable tool for scientific discovery.

In this paper, we first give a detailed account of **PRISM** at propositional logic level. Then we concentrate, instead of looking over various fields, on one subject, the inheritance mechanism of blood types. We show with experimental results that various theories of blood type inheritance are described as **PRISM** programs. Finally we suggest possible extensions of **PRISM**. The reader is assumed to be familiar with logic programming [8].

## 1 Introduction

Scientific discovery goes through a cycle of collecting observations, building a hypothesis and verifying it one way or another. Oftentimes, observations are given as numbers and a hypothesis is expressed as a mathematical formula. When observations include non-numeric aspects such as causal relationship among events however, we need a symbol manipulation language to be able to express our hypotheses. We also need to cope with noise or uncertainty in the observations that is inevitable in the real world.

**PRISM** is a formal knowledge representation language for modeling scientific hypotheses about discrete phenomena which are governed by rules and probabilities [18, 19]. It is also a programming language containing both logical elements and probabilistic elements. Syntactically **PRISM** programs are just an extension of logic programs in which the set of facts in a program has a probability distribution.

Semantically though, **PRISM** programs have a probabilistic version of possible worlds semantics (termed *distributional semantics* [18]) defined on a certain probability space (of infinite dimension). Distributional semantics extends a standard fixed point semantics and views a **PRISM** program as specifying an infinite joint distribution for all possible fixed points.[3]

[1] Dept. of Computer Science, Tokyo Institute of Technology, 2-12-1 Ôokayama Meguro-ku Tokyo Japan 152

[2] An overview of **PRISM** was presented in [19]. It was based on the first implementation of **PRISM** and programming examples were chosen from various fields.

[3] In logic programming, the (least) fixed point of a program is the set of all ground atoms provable from the program.

**ECAI 98 Workshop Programme.**

Although **PRISM** has the ability of computing all computable functions, it must be emphasized that it is not just another programming language but a programming language with learning ability, and it is this learning ability that distinguishes **PRISM** from other existing programming languages [19].

When given goals, i.e. conjunctive ground atoms which express our observations together with a program that models them, **PRISM** tries to maximize the probability of the goals, by first identifying the relevant facts through backward reasoning from the goals, and then adjusting the statistical parameters associated with the facts, by a learning algorithm based on the EM algorithm [20]. Although our learning is just parameter learning by MLE (Maximum Likelihood Estimate) applied to computer programs, and already found in, for instance, HMM (hidden Markov model) in speech recognition and their applications to genetic information processing [17, 2], the point is that unlike the case of HMM where the underlying model is restricted to probabilistic automata (type 3 grammars in the Chomsky hierarchy), **PRISM** applies MLE to the whole class of programs (type 0 grammars), and hence it covers not only HMMs but other well-known symbolic-statistical model such as Bayesian network [14] and PCFG (Probabilistic Context Free Grammars) [3].

After getting parameter learning done, we let the program compute, through forward reasoning from the learned distribution and facts, probabilities of atoms that represent phenomena we are interested in. This way **PRISM** provides us with a tool to build a symbolic model of probabilistic phenomena and, after statistical tests, allows us to calculate reliable values of probabilities of their behaviors for the purposes of prediction.

In the rest of this paper, we first explain what **PRISM** is like in detail (this is because the combination of symbolic execution and statistical learning looks new to the reader), then proceed to describing a series of examples of scientific theories taken from the area of blood gene inheritance. We begin by a simple one, an `ABO` blood type program, then proceed to more complex ones, for instance, the interplay of a social structure and the genetic inheritance mechanism found in the Kariera tribe which lived 80 years ago in the west Australia [21]. Finally we suggest future directions such as rule extraction from data which can be made possible by extending **PRISM**. The reader is assumed to be familiar with logic programming [8].

## 2 The basic idea of PRISM

### 2.1 Modeling and forward reasoning

A **PRISM** program $\mathcal{DB}$ is written as $\mathcal{F} \cup \mathcal{R}$ where $\mathcal{F}$ is a set of unit clauses and $\mathcal{R}$ is a set of definite clauses.[4] One salient feature of **PRISM** is that $\mathcal{F}$ is given a probability distribution. We call it a *basic distribution* (of the $\mathcal{DB}$).

The purpose of **PRISM** programs is to represent computational models of our observations by logical statistical means. The $\mathcal{R}$ part is used to describe logical rules working behind the observations whereas the $\mathcal{F}$ part models their uncertainty as a probability distribution. Our assumption is that what we observe is representable as an instance of some clause head in the program.

Suppose, for example, we observe phenomena concerning our health, say a headache and a fever. We know from experience that drinking too much causes a headache while a cold causes a headache and a fever simultaneously. If we use ground atom $H$ for having a headache, $F$ for having a fever, $D$ for drinking too much and $C$ for a cold, respectively, our causal knowledge about headaches and fever is formalized as $\mathcal{DB}$ listed below.[5]

$$\begin{aligned} \mathcal{DB} &= \mathcal{F} \cup \mathcal{R} \\ \mathcal{F} &= \{D, C\} \\ \mathcal{R} &= \{H \Leftarrow D, H \Leftarrow C, F \Leftarrow C\} \end{aligned}$$

We also know that drinking too much has little to do with a cold, logically speaking. We therefore, instead of further seeking for a logical relationship between them, just introduce a joint probability distribution (*basic distribution*) $P_{\mathcal{F}}$ for them.

| $\langle x_1, x_2 \rangle$ | $P_{\mathcal{F}}(D = x_1, C = x_2)$ |
|---|---|
| $\langle 0,0 \rangle$ | 0.72 |
| $\langle 1,0 \rangle$ | 0.18 |
| $\langle 0,1 \rangle$ | 0.08 |
| $\langle 1,1 \rangle$ | 0.02 |

Here we think of ground atoms as random variables taking on 1 when they are true and 0 when they are false, and we use $x_i$ for the value of an atom. The basic distribution is extended to a one for all ground atoms $D, C, H$ and $F$ in the program by taking the iff definition of $\mathcal{DB}$ [8]

$$\text{iff}(\mathcal{DB}) = \{H \Leftrightarrow D \vee C, F \Leftrightarrow C\}$$

and then by calculating the truth values of $H$ and $F$ from those of $D$ and $C$.[6] For example, when $D$ is true and $C$ is false, $H$ is true and $F$ is false. This *forward reasoning* gives rise to the following joint distribution $P_{\mathcal{DB}}$ for $\langle D, C, H, F \rangle$ where $y_i$ denotes the sampled value of $H$ or $F$.

[4] To be precise, the program has three parts, the modeling part, the utility part and the control directives. We are talking here only about the modeling part for simplicity.

[5] We assume that rules in $\mathcal{R}$ are always correct.

[6] The use of the iff definition is justifiable because our semantics, distributional semantics, is a probabilistic generalization of the least model semantics of logic programs in which the iff definition always holds. See [8, 18]

| $\langle x_1, x_2, y_1, y_2 \rangle$ | $P_{\mathcal{DB}}(D = x_1, C = x_2, H = y_1, F = y_2)$ |
|---|---|
| $\langle 0,0,0,0 \rangle$ | 0.72 |
| $\langle 1,0,1,0 \rangle$ | 0.18 |
| $\langle 0,1,1,1 \rangle$ | 0.08 |
| $\langle 1,1,1,1 \rangle$ | 0.02 |
| others | 0.0 |

Once this distribution is obtained, we are able to calculate whatever statistics we want. If we are interested in the joint distribution of $H$ and $F$, it becomes

| $\langle y_1, y_2 \rangle$ | $P_{\mathcal{DB}}(H = y_1, F = y_2)$ |
|---|---|
| $\langle 0,0 \rangle$ | 0.72 |
| $\langle 1,0 \rangle$ | 0.18 |
| $\langle 0,1 \rangle$ | 0.0 |
| $\langle 1,1 \rangle$ | 0.10 |

This tells us for instance the probability of simultaneously having a headache ($H$) and a fever ($F$) is 0.10.

### 2.2 Backward reasoning and statistical parameters

In the previous example, $P_{\mathcal{F}}$ was given a priori. In reality however, constructing reliable $P_{\mathcal{F}}$ is a major problem. We show that this can be done by statistical learning based on the combination of backward reasoning and the EM algorithm [20] as follows.

First we suppose, to make matters simple that drinking too much and having a cold are statistically independent events. So we write $P_{\mathcal{F}}(D = x_1, C = x_2) = P_{\mathcal{F}}(D = x_1)P_{\mathcal{F}}(C = x_2)$. Second, we introduce statistical parameters $\theta_i$ $(i = 1, 2)$ such that $P_{\mathcal{F}}(D = 1) = \theta_1$ and $P_{\mathcal{F}}(C = 1) = \theta_2$.

Now suppose we have observed, randomly, the state of headache ($H$) and a fever ($F$) three times and have recorded them as $\langle \neg H, \neg F \rangle$, $\langle H, \neg F \rangle$ and $\langle H, F \rangle$. The probability of these observations happening is $P_{\mathcal{DB}}(H = 0, F = 0)P_{\mathcal{DB}}(H = 1, F = 0)P_{\mathcal{DB}}(H = 1, F = 1)$. We express this probability in terms of $\theta_1$ and $\theta_2$ by using backward reasoning.

Take for example $P_{\mathcal{DB}}(H = 0, F = 0)$, the probability of simultaneously having no headache and no fever. Since distributional semantics allows us to logically reduce $H$ to $D \vee C$ and $F$ to $C$ respectively (this is backward reasoning), the probability of $\langle H = 0, F = 0 \rangle$, i.e. that of $\neg H \wedge \neg F$ is equal to the probability of $\neg(D \vee C) \wedge \neg C$ ($\Leftrightarrow \neg D \wedge \neg C$), which is $(1 - \theta_1)(1 - \theta_2)$. Likewise, the probability of $\langle H = 1, F = 0 \rangle$ is $\theta_1(1 - \theta_2)$ and that of $\langle H = 1, F = 1 \rangle$ is $\theta_2$. Accordingly, $P_{\mathcal{DB}}(H = 0, F = 0)P_{\mathcal{DB}}(H = 0, F = 1)P_{\mathcal{DB}}(H = 1, F = 1)$ is calculated as $(1 - \theta_1)\theta_1(1 - \theta_2)^2\theta_2$.

### 2.3 MLE and the EM algorithm

To statistically learn reliable values of $\theta_1$ and $\theta_2$ from the observations, we appeal to the principle of maximizing the probability of what we observed, i.e. MLE (Maximum Likelihood Estimate). So, we maximize $(1 - \theta_1)\theta_1(1 - \theta_2)^2\theta_2$ by choosing appropriate $\theta_1$ and $\theta_2$, in this case $\theta_1 = 1/2$ and

$\theta_2 = 1/3$.

Unfortunately, generally speaking, MLE is a complex optimization problem and there is no easy way to calculate the parameter values giving the optimum, especially when there are a large number of random variables. So when designing **PRISM** , we decided to be satisfied with a local maximum, and to apply the EM algorithm [20] to a class of basic distributions of very simple type (the EM algorithm is an iterative algorithm for MLE which fits particularly well when some data are missing).

Note that we have a kind of *data missing situation* in which we observe only atoms that appear as a clause head but facts (sampled according to the basic distribution $P_{\mathcal{F}}$) are never known, i.e. missing. We thus applied the EM algorithm to this data missing situation and derived a new learning algorithm for our MLE problems [18].

This new learning algorithm can apply to *any logic programs* as long as each observation (ground atom) has a finite number of explanations (proofs) from the $\mathcal{DB}$.[7]

Getting back to the observation of $H$ and $F$, if we have a large number of random samples of $H$ and $F$, reliable values of $\theta_1$ and $\theta_2$ are obtainable by applying the MLE (our EM learning algorithm) to the observed samples. And once these reliable values have been obtained, we can calculate probabilities such as $P_{\mathcal{DB}}(H = 1 \mid F = 1)$.

## 2.4 PRISM programming

**PRISM** programming has the following four steps.

**(1) Modeling** We build a logical-statistical model, assuming a basic distribution over facts.
**(2) Learning** Given (random) observations as ground atoms, we adjust parameters associated with the basic distribution by the EM learning algorithm.
**(3) Testing** After learning, we check our model against the observations by statistical tests.
**(4) Calculation (Prediction)** We use our model to calculate various statistics, for instance, for predication.

Various built-in predicates are available for these steps. To write a basic distribution in the modeling step, we use `bsw/3` and `msw/3`. `bsw(coin,2,1)` for instance behaves as a random binary switch named `coin` which is true (resp. false) if and only if sampling `coin` at 2nd time returns 1 (resp. 0). Likewise `msw(<name>,<time>,<value>)` acts as a random multiple switch. These atoms are statistically independent random variables.[8]

**PRISM** associates a statistical parameter with each `bsw` (and `msw`) atom as the probability of the atom being true. Let $\theta$ be such a statistical parameter for `bsw(coin,_,1)`. It means Prob(`bsw(coin,_,1)`) $= \theta$ which also implicitly means Prob(`bsw(coin,_,0)`) $= 1 - \theta$ because only two values, 1 and 0, are allowed in the third argument of the `bsw` atom. Statistical parameters are set manually by a built-in predicate or are adjusted in the learning step by the EM algorithm.

In the learning step, the user is required to prepare teacher data in the form of ground atoms. They must be randomly sampled ones. Presently, two built-in predicates, `learn/0` and `learn/1`, are available. `learn/0` is used when teacher data are stored on a file, while `learn/1` is used when teacher data are supplied as a list of ground atoms. They all execute the EM learning algorithm until saturation by repeatedly adjusting statistical parameters associated with `bsw/3` and `msw/3` atoms in a program to achieve a local maximum of the probability of the conjunction of teacher atoms.

For the testing and calculation steps, `prob/2` and `cprob/3` are available as built-in predicates. `prob(<f>,<p>)` is used to calculate the probability `p` of the positive boolean formula `f`. `cprob(<f-1>,<f-2>,<p>)` is used for the conditional probability `p` of `f-1` when `f-2` is true.

We now look at concrete **PRISM** programs. We show however only the modeling part of them, omitting the utility part and the control directives as they are of secondary importance compared to the modeling part.

# 3 ABO blood type program

Our first example is about `ABO` blood type [5, 6]. Let us explain modeling, sampling and learning in **PRISM** with this example.

We already know that a child inherits genes, one from each parent, and the pair (*genotype*) determines the child's blood type (*phenotype*). In the case of `ABO` blood type, three genes (`a`, `b` and `o`) are involved; if the child inherits `b` from the father and `o` from the mother, his/her blood type is `B` and so on (see **Figure 1**). We also know that the inheritance is by chance. That is, for a randomly sampled child, which gene is inherited from his/her parent is probabilistic, and the inheritance from the father and that from the mother are statistically independent.

## 3.1 Modeling

This inheritance mechanism that models random mating [6] is succinctly described as a **PRISM** program in **Figure 2**.

Practically speaking, **PRISM** programs are just Prolog programs with special built-in predicates. So, the upper case letters `X`,`Y` and `P` are logical variables and the rest are constants. More importantly, **PRISM** execution follows Prolog convention,[9] top-to-bottom, clause-wise, and left-to-right goal-wise. Although it happens that this program is non-recursive, **PRISM** allows for recursion (see an HMM program in [19]).[10]

[7] This restriction seems reasonable. In the real world, it is usual that we don't need to consider infinitely many causes of phenomena in question.

[8] To be exact, if their `names` are different, they are independent. If they have the same `name` but different `times`, they are independent but identically distributed.

[9] **PRISM** has three execution modes: sampling mode, answer with probability mode and answer with formula mode. We are focusing on the sampling mode here.

[10] Recursion introduces an infinite number of random variables, even in such a simple case as modeling a Bernoulli sequence,

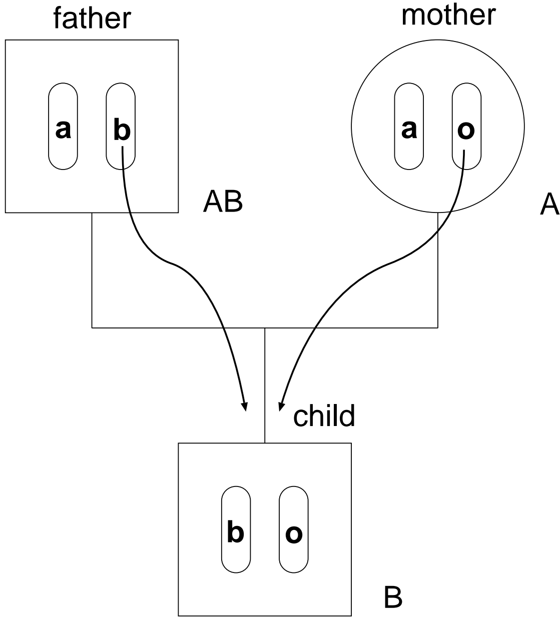


**Figure 1.** ABO **blood type model**

```
blood_type(a):- (gtype(a,a); gtype(a,o); gtype(o,a)).
blood_type(b):- (gtype(b,b); gtype(b,o); gtype(o,b)).
blood_type(o):- gtype(o,o).
blood_type(ab):- (gtype(a,b); gtype(b,a)).

gtype(X,Y):- gene(father,X),gene(mother,Y).

gene(P,a):- msw(gene,P,a).
gene(P,b):- msw(gene,P,b).
gene(P,o):- msw(gene,P,o).
```

**Figure 2.** ABO **blood type program**

The declarative content of this program seems self-evident. We have only to note that `blood_type(a)` says a child has blood type `a`, `gtype(X,Y)` says a child has the genotype `<X,Y>` and `gene(P,a)` says gene `a` is inherited from `P`. The last three clauses use `msw` atoms. In the current case, `msw(gene,P,V)` stands for a statistically independent and identically distributed ternary random switch named `gene` such that `V` takes on one of {`a, b, o`} as a sampled value at `P`th time. If `P = father` and `V = a`, it means the child happened to inherit gene `a` from `father`.

## 3.2 Sampling execution

Putting declarative content aside, there is a little intricacy in operational semantics. Although **PRISM** has three execution modes, we pay attention only to the sampling execution.

which necessitates the construction of a probability space with an infinite dimension [18].

In general, the sampling execution of a **PRISM** program $\mathcal{DB} = \mathcal{F} \cup \mathcal{R}$ is stated as the sampling of $\mathcal{DB}$ as a (countably infinite) random vector defined by distributional semantics [18].[11] However, since our random variables are just the namesake of ground atoms, we state a sampling process in terms of ground atoms for the sake of intuitiveness.

The sampling process starts with that of $P_{\mathcal{F}}$, the basic distribution over $\mathcal{F}$. In the blood type example, $\mathcal{F}$ consists of `msw(gene,_,a)`, `msw(gene,_,b)` and `msw(gene,_,o)`. **PRISM** implicitly associates two statistic parameters, $\theta_1$ and $\theta_2$ as follows. They are initialized before computation by a built-in predicate.

| atom | prob. |
|---|---|
| `msw(gene,_,a)` | $\theta_1$ |
| `msw(gene,_,b)` | $\theta_2$ |
| `msw(gene,_,o)` | $1 - \theta_1 - \theta_2$ |

## 3.3 Top-down execution as sampling

A sampling of $P_{\mathcal{F}}$ determines a set of true ground atoms $\mathcal{F}' \subset \mathcal{F}$, and the least model of $\mathcal{F}' \cup \mathcal{R}$ determines all truth values of ground atoms appearing in $\mathcal{DB}$. Thus, all ground atoms are sampled.

This sampling process suggests bottom-up computation; first sample all `bws/3` or `msw/3` ground atoms (even if there are infinitely many of them) and then with the set of true atoms, compute the least model. This is however not very amenable to Prolog on top of which **PRISM** is currently implemented. So while adopting Prolog's top-down execution order, we let **PRISM** pretend that the sampling from $P_{\mathcal{F}}$ has been done before any computation starts and results are stored somewhere for retrieval.

With this in mind, let us trace a computation process starting from a call to `gtype(b,o)`. Like Prolog, the call unique invokes the clause for `gtype/2` with instantiations `X=b` and `Y=o`, and a call to subgoal `gene(father,b)` occurs which then invokes `gene/2` clauses in turn from top-to-bottom. When the topmost clause `gene(P,a):- msw(gene,P,a)` is invoked with `P` instantiated to `father`, a call to `msw(gene,_,_)` takes place for the first time. **PRISM** samples the truth value of the atom, or more precisely, it samples a random variable whose name is `gene` and whose value is one of `a`, `b` or `o`. If the sampled value is `b`, **PRISM** memorizes `msw(gene,father,b)` as the result of the sampling at the time `father`.

This sampling determines `msw(gene,father,a)` to be false as side effect (any later call to it will fail). So the second clause `gene(P,b):- msw(gene,P,b)` is invoked with `P = father`. The call to `msw(gene,father,b)` ensues and succeeds because **PRISM** remembers that `msw(gene,father,b)` was true on the first sampling. This success completes the goal `gene(father,b)`.

[11] In the formal setting, we always consider $\mathcal{DB}$ as representing the infinite set of ground instantiations of clauses in $\mathcal{DB}$.

**PRISM** then tackles the second subgoal gene(mother,o). This subgoal eventually causes a second sampling of the random variable gene, because time stamp mother is different from father used before. If this sampling returns o, gene(mother,o) will be exclusively true, resulting in the success of the call to gtype(b,o). Else gene(mother,o) will fail and so will gtype(b,o). This way we are able to sample the truth value of ground atom gtype(b,o). When goals contains variables, unification occurs as usual, and when goals succeed, we will have variable bindings (answer substitutions) just like Prolog.

### 3.4 Learning and testing

Suppose that we randomly observed people's blood types and obtained the following table.[12]

| atom | count | ratio |
|---|---|---|
| blood_type(a) | 195 | 0.390 |
| blood_type(b) | 97 | 0.194 |
| blood_type(o) | 159 | 0.318 |
| blood_type(ab) | 49 | 0.098 |
| (total) | 500 | 1.000 |

**Table 1. Sample observations**

For the ABO blood type program to be a good scientific model, the probability distribution specified by the program should be close to the observed distribution. Hence, we adjust statistical parameters associated with msw/3 that determine the probabilities of gene(P,a), gene(P,o) and gene(P,b) so that the resulting distribution gets closer to the above table.

The adjustment is carried out by statistical learning based on the EM algorithm [20, 18]. We run the command learn/1 with the list of ground atoms that represent our observations as follows.

```
| ?- learn([blood_type(a),blood_type(a),
           blood_type(o),blood_type(b),...]).
```

learn/1 starts the EM learning algorithm, and iteratively adjusts parameters associated with msw/3 to maximize the probability of blood_type(a) $\wedge$ blood_type(a) $\wedge$ blood_type(o) $\wedge$ blood_type(b) ... After 11 iterations, the learning stopped. **Table 2** shows learned parameter values.

Finally, the distribution of blood_type/1 specified by these parameters is computed by prob/2 built-in predicate, giving **Table 3**.

Since the **Table 3** is just a theoretically computed distribution, we need to check by the $\chi$-square method how well it fits what we observed, i.e. **Table 1**. By calculation, we get $\chi^2 = 0.6545$. For degrees of freedom 1 (we have two parameters $\theta_1$ and $\theta_2$), this happens with probability $> 0.4$. So **Table 3** cannot be rejected, which indirectly supports, together with **Table 2**, the correctness of ABO blood type program that produced it.

[12] This data set is generated by a program. It reflects the blood type distribution in Japan. The ratio is almost A:B:O:AB = 4:2:3:1.

| atom | parameter | learned value |
|---|---|---|
| msw(gene,_,a) | $\theta_1$ | 0.2835 |
| msw(gene,_,b) | $\theta_2$ | 0.1580 |
| msw(gene,_,o) | $1-\theta_1-\theta_2$ | 0.5585 |
| (total) | | 1.0000 |

**Table 2. Learned parameters**

| atom | computed prob. |
|---|---|
| blood_type(a) | 0.3970 |
| blood_type(b) | 0.2015 |
| blood_type(o) | 0.3119 |
| blood_type() | 0.0896 |
| (total) | 1.0000 |

**Table 3. Computed probabilities**

## 4 Blood type model for children of cousin parents

When parents are cousins, we have to take the fact into account to build a proper model of genetic inheritance of blood genes. This is because a child has the chance of inheriting copies of the same gene from one of the grand-grand parents (see **Figure 3**). The probability of its happening is calculated as 1/16 and is called a coefficient of inbreeding [5].

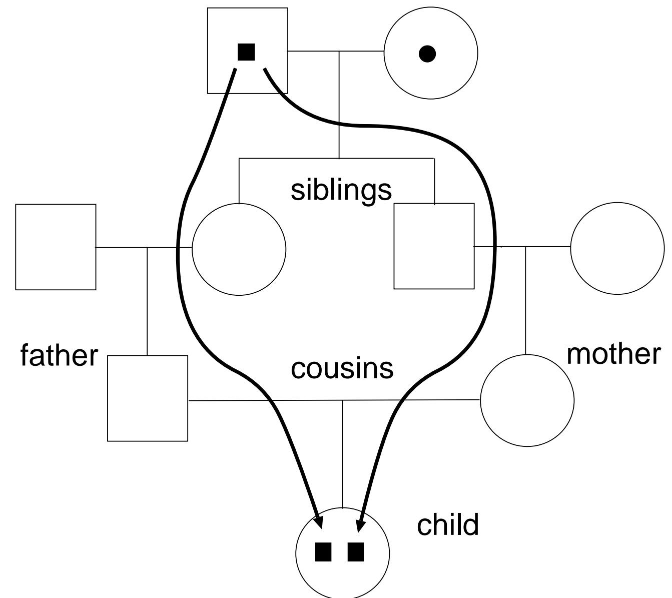


**Figure 3. Cousin model**

**Figure 4** is part of a program that models the blood type inheritance mechanism for children of cousin parents. child_of_cousin(a) says that a child of cousin parents has a blood gene a and cousin(X,Y) says randomly chosen two

cousins have gene X and Y.

inherit(from(XP),pair(sibling_parent,other),1) describes gene inheritance that occurs between the cousin and the cousin's parents. XP denotes the cousin's parent who is the source of gene X. The atom says as the result of gene inheritance at 1st trial, XP becomes, with prob. 1/2, either a sibling parent or the other non-sibling parent. siblings(X,Y) says randomly chosen two siblings have gene X and Y respectively.

We assume that as far as non-sibling parents are concerned, the distribution of their blood types are determined by random mating. In other words, we use the ABO program shown in **Figure 2** as a program to determine the blood type distribution of non-sibling parents.

```
child_of_cousin(a):-
  (cousin(a,a); cousin(a,o); cousin(o,a)).
child_of_cousin(b):-
  (cousin(b,b); cousin(b,o); cousin(o,b)).
child_of_cousin(o):- cousin(o,o).
child_of_cousin(ab):-
  (cousin(a,b); cousin(b,a)).

cousin(X,Y):- % randomly chosen two cousins have
              % gene X and gene Y
  inherit(from(XP),pair(sibling_parent,other),1),
  inherit(from(YP),pair(sibling_parent,other),2),
  ( XP==sibling_parent, YP==sibling_parent,
      siblings(X,Y) % randomly chosen two siblings
  ; ...             % have gene X and gene Y
```

**Figure 4. Program for children of cousin parents**

After writing down the whole program, we provide it with parameters in **Table 2**, i.e. parameters for the blood type distribution of general people, and calculate the distribution of blood type for children of cousin parents. This is a predication step. We compute an unobserved distribution by using our cousin model. The result is shown in **Table 4**. The next step would be comparing these probabilities with observations of blood types of children whose parents are cousins.

| atom | prob. |
|---|---|
| child_of_cousin(a) | 0.3899 |
| child_of_cousin(b) | 0.1988 |
| child_of_cousin(o) | 0.3273 |
| child_of_cousin(ab) | 0.0840 |
| (total) | 1.0000 |

**Table 4. Predicted probabilities**

## 5 Yet another blood type theory

Here we describe yet another blood type theory that lost to ABO blood type theory [6]. This theory, AaBb blood type theory, assumes two loci. Blood type is determined by alleles at each locus according to **Table 5**.

| type | locus-1 | locus-2 |
|---|---|---|
| O | aa | bb |
| A | A- | bb |
| B | aa | B- |
| AB | A- | B- |

**Table 5. AaBb blood type theory**

Here A,B are dominant genes whereas a,b are recessive genes. - is a don't care symbol. This theory was once one of the competing theories for explaining the ABO blood type [6] but failed to pass a statistical test and eventually was rejected. We trace this whole scientific process of proposing a hypothesis, checking it with data, determining acceptance or rejection, in the framework of **PRISM** programming.

We first write a program shown in **Figure 5**. Note that in the program, "-" is represented as an anonymous logical variable "_". Next by using the same data, i.e. **Table 1** used for ABO blood type theory, we let it learn statistical parameters in the program. Then we calculate the distribution of blood types determined from the learned parameters. Finally we check them against the observed data **Table 1** by $\chi$-square fitting test.

```
blood_type(o):-
  gtype(1,[a,a]),gtype(2,[b,b]).
blood_type(a):-
  (gtype(1,['A',_]); gtype(1,[_,'A'])),
  gtype(2,[b,b]).
blood_type(b):-
  gtype(1,[a,a]),
  (gtype(2,['B',_]); gtype(2,[_,'B'])).
blood_type(ab):-
  (gtype(1,['A',_]); gtype(1,[_,'A'])),
  (gtype(2,['B',_]); gtype(2,[_,'B'])).

gtype(N,[X,Y]):- gene(N,father,X),gene(N,mother,Y).

gene(1,P,G):-(bsw(0,P,1), G=a; bsw(0,P,0), G='A').
gene(2,P,G):-(bsw(1,P,1), G=b; bsw(1,P,0), G='B').
gene('A'):- gene(1,father,'A').
gene(a):- gene(1,father,a).
gene('B'):- gene(2,father,'B').
gene(b):- gene(2,father,b).
```

**Figure 5. AaBb blood type program**

The probabilities of gene/1 and blood_type/1 computed

from the learned parameters are summarized in **Table 6**.

| atom | prob. | atom | prob. |
|---|---|---|---|
| gene(A) | 0.2845 | blood_type(a) | 0.3455 |
| gene(a) | 0.7155 | blood_type(b) | 0.1495 |
| gene(B) | 0.1586 | blood_type(o) | 0.3625 |
| gene(b) | 0.8414 | blood_type(ab) | 0.1425 |
| | | (total) | 1.0000 |

**Table 6. Computed probabilities for AaBb theory**

Comparing the observed data in **Table 1** with the computed data in **Table 6**, we find $\chi^2 = 19.16486$. With degrees of freedom 1 (again, we have two parameters), this happens with probability $< 0.001$. So AaBb theory should be rejected.

## 6 A case of anthropological study – the Kariera tribe

To explore the descriptive power of a first-order computational language combined with a statistical learning method, we take up the Kariera tribe from anthropology where social rules interact with a genetic inheritance mechanism.

In the beginning of the 20th century, the Kariera tribe lived along the coast of West Australia which had the territory of about 3500 square miles. A special type of cross-cousin marriage (bilateral cross-cousin marriage) was a norm in their society. It states that a husband H and his wife W must be cousins such that H's father and W's mother are siblings (H and W are patrilateral cross-cousins), and at the same time H's mother and W's father are siblings as well (H and W are matrilateral cross-cousins) [21].

The tribe was divided into four clans 'Palyeri', 'Karimera', 'Banaka' and 'Burung'. 'Palyeri' and 'Karimera' form a marriage pair, and so do 'Banaka' and 'Burung' (**Figure 6**). A husband and his wife must come respectively from each of the paired clans. Their children however belong to the third clan uniquely determined by the parents' clans.

The blood type inheritance mechanism in this society which was complicated by the social rules mentioned above can be described as a **PRISM** program. We first include in the following facts about the Kariera clans.

```
child('Banaka',  parents('Palyeri','Karimera')).
child('Burung',  parents('Karimera','Palyeri')).
child('Palyeri', parents('Banaka','Burung')).
child('Karimera',parents('Burung','Banaka')).
```

Here child(C, parents(FC,MC)) for instance means if the father's clan is FC and the mother's clan is MC then the child belongs to clan C. We then add the usual ABO blood program (**Figure 2**) appropriately modified to the Kariera case. For instance, to say a randomly chosen two individual in

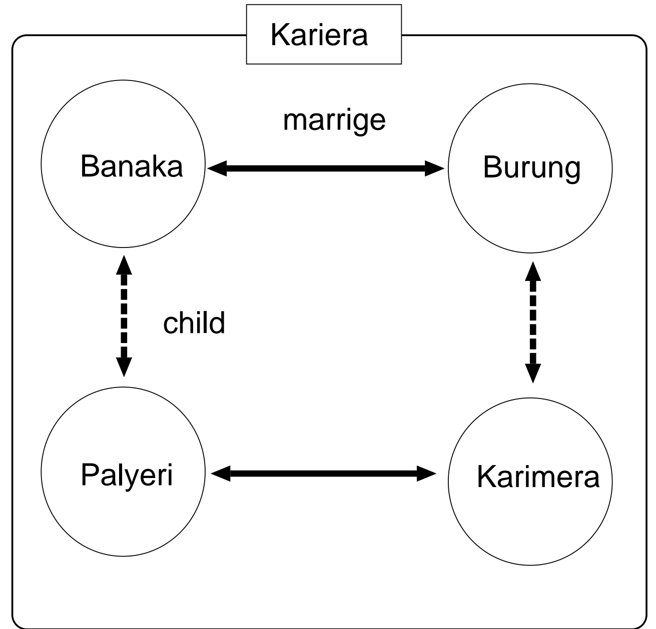


**Figure 6. Clans in the Kariera tribe**

```
gtype(C,[X,Y],N):-
  N>0,
  children(C,parents(FC,MC)),
  inherit(genes,from(X1,Y1),pair(X,Y),N),
  cousins(clan(FC,X1),clan(MC,Y1),N).
```

clan C has genotype <X,Y> at the N-th generation, we define gtype(C,[X,Y],N) as follows.

inherit(genes,from(X1,Y1), pair(X,Y), N) says the parents have genes {X1,Y1} as a set and the child's genotype <X,Y> coincides with either <X1,Y1> or <Y1,X1> with probability 1/2. cousins(clan(FC,X1),clan(MC,Y1),N) says a randomly chosen two cousins, one (male) in clan FC and the other (female) in clan MC, have gene X1 and Y1 respectively at the N-th generation.

Since all married couples are cousins in this society, cousins' ancestors are also cousins. This brings unbound recursion through generation into the program. And a sufficient number of recursion through generations are expected to lead the blood gene distribution computed by the program to a stable state. In actual computation however, computational cost of recursion was so high that recursion depth was limited to 2.

We here describe one of the several learning experiments with artificial data. We first randomly sampled 100 ground blood_type/1 atoms (the sampling ratio was A:B:O:AB =4:2:3:1) and obtained **Table 7**.

In the learning step, we let the program learn statistical parameters from this data set in which we assumed the ratio of clan population is 'Palyeri':'Karimera':'Banaka':'Burung' = 2:2:1:1. The table **Table 8** of probabilities computed from

| atom | counts | ratio |
|---|---|---|
| blood_type(a) | 34 | 0.340 |
| blood_type(b) | 27 | 0.270 |
| blood_type(o) | 33 | 0.330 |
| blood_type(ab) | 6 | 0.060 |
| (total) | 100 | 1.000 |

**Table 7. Sample observations**

the program using learned parameters (and this passed successfully $\chi$-square fitting test).

| atom | prob. |
|---|---|
| blood_type(a) | 0.3382 |
| blood_type(b) | 0.2690 |
| blood_type(o) | 0.3312 |
| blood_type(ab) | 0.0616 |
| (total) | 1.0000 |

**Table 8. Learned results**

Unfortunately, due to the lack of conrete data and for other reasons, this learning experiment can only be theoretically meaningful. It however exemplifies that **PRISM** programming can still work for such complicated phenomena as the interaction between social rules and a genetic mechanism.

## 7 Discussion and related work

Modeling is an indispensable step in the process of scientific discovery and **PRISM** was designed exactly for this purpose [18, 19].

Since **PRISM** integrates a general logic programming language and the statistical learning by the EM algorithm at semantic level and at procedural level, not only can it declaratively describe and simulate complex phenomena governed by rules and probabilities, but can learn from data (goals), and hence, can change the behavior of programs adaptively. In this paper, we have presented the basic idea of **PRISM** and its modeling examples taken from the area of blood gene inheritance.

It should be mentioned however that **PRISM** programming tends to be more difficult as we have to consider logical meaning and probabilistic relationship at the same time, and encode them, coherently, into a single program. This means if we want to verify our program, we need the logical and statistical verification of the denotation of the program, i.e. a joint distribution of all ground atoms. Little is known about such verification however. Despite these difficulties, we have shown by examples that **PRISM** can be a valuable tool with first-order descriptive power and learning ability for modeling complex phenomena from HMM to Bayesian nets [18, 19] to the ones presented in this paper.

Next step seems diverse. One is rule extraction from a basic distribution. First we augment **PRISM** with a more powerful class of basic distributions such as Boltzmann distributions than random switches. We then maximize the probability of given goals (random samples) by the Boltzmann machine learning method [1]. Since the learned distribution reflects probabilistic correlations between atoms, we may have a good chance of extracting logical rules from it.

Another is modeling dynamic temporal phenomena such as cooperation among agents with logical reasoning ability. This seems to require an on-line learning mechanism unlike the current off-line learning mechanism of **PRISM**. Research in the area of reinforcement learning will be of much help.

Last but not least, it is also important to explore new areas which **PRISM** modeling can contribute to. We are currently looking into the possibility of applying **PRISM** modeling to the area of marketing.

We state related work.

Ng and Subrahmanian combined (function free) logic programs with probability ranges (upper and lower bounds) to express uncertainty [12]. After assigning probability ranges to atoms in the logic program, they checked if probabilities satisfying those ranges actually exist or not by the technique of linear programming. The use of linear programming however confines their approach to a finite domain. Learning was not considered. Recent development is described in [7].

Probabilistic Horn abduction [15] also combined logic programs with probabilities, in which Poole showed, for example, how to express Bayesian networks. His semantics on the other hand excludes large part of usual logic programs because of the requirement of acyclicity of programs (programs must be stratifiable by the assignment of integers to ground atoms) [15]. His recent publication [16] focused on modeling dynamic behavior such as that of agents and dealt with decision/game theoretic means based on acyclic logic programs (with negation as failure). Neither of the frameworks mentioned how to learn probabilities.

Charniak and Goldman proposed a special language FRAIL3 for construction of Bayesian networks [4]. Although their rules look much like definite clauses annotated with probability dependencies, the semantics is not very clear and no learning mechanism is provided for their programs.

Hashida [9] proposed a rather general framework for natural language processing as probabilistic constraint logic programming. He assigned probabilities to between literals and let them denote the degree of the probability of invocation. He has shown constraints are efficiently solvable by making use of these probabilities.

Koller [10] deals with the problem of gene inheritance. Knowledge about gene inheritance is encode by a first order language and then translated into a Bayesian network. The EM learning algorithm adapted for Bayesian networks is applied to statistically estimate parameters in the network. It is not clear

however how to extend this approach to a general programming language with Turing computation power. She also proposed a functional probabilistic programming language [11] for stochastic processes.


## ACKNOWLEDGEMENTS

We would like to thank the referees for their comments which helped improve this paper.


## REFERENCES


[1] Ackley,D.H., Hinton,G.E. and Sejnowski,T.J., A learning algorithm for Boltzmann machines, Cognitive Sci. 9, pp147-169, 1985.

[2] Asai,K., Hayamizu,S. and Handa,K., Prediction of protein secondary structure by the hidden Markov model, CABIOS 9 No.2 pp141-146, 1993.

[3] Charniak,E., Statistical Language Learning, The MIT Press, 1993.

[4] Charniak,E. and Goldman,R.P., A Language for Construction of Belief Networks, IEEE PAMI 15 No 3, pp196-208, 1993

[5] J.F.Crow, Genetics Notes (8th ed.)(translated into J), Burgess Publishing Company, Minneapolis, 1983.

[6] J.F.Crow, Basic Concepts in Population, Quantitative and Evolutionary Genetics (translated into J), W.H.Freeman and Company, New York, 1988.

[7] Dekhtyar,A. and Subrahmanian,V.S., Hybrid Probabilistic Programs, Proc. of Fourteenth ICLP, pp391-405, 1997.

[8] Doets,K., From Logic to Logic Programming, MIT Press, Cambridge, 1994.

[9] Hashida,K., Dynamics of Symbol Systems, NGC 12, pp285-310, 1994.

[10] Koller,D. and Pfeffer,A., Learning probabilities for noisy first-order rules, Proc. of IJCAI'97, Nagoya, pp1316-1321, 1997.

[11] Koller,D., McAllester,D. and Pfeffer,A., Effective Bayesian Inference for Stochastic Programs, Proc. of AAAI'97, Rhode Island, pp740-747, 1997.

[12] Ng,R. and Subrahmanian,V.S., Probabilistic Logic Programming, Information and Computation 101, pp150-201, 1992.

[13] Ng,R., Haddawy,P. and Helwig,J., A Theoretical Framework for Context-Sensitive Temporal Probability Model Construction with Application to Plan Projection, Proc. of the eleventh UAI,Montreal, pp419-429, 1995.

[14] Pearl,J., Probabilistic Reasoning in Intelligent Systems, Morgan Kaufmann, 1988.

[15] Poole,D., Probabilistic Horn abduction and Bayesian networks, Artificial Intelligence 64, pp81-129, 1993.

[16] Poole,D., The independent choice logic for modeling multiple agents under uncertainty, Artificial Intelligence 94, pp7-56, 1997.

[17] Rabiner,L.R., A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, Proc. of the IEEE 77, No. 2, pp257-286, 1989.

[18] Sato,T., A Statistical Learning Method for Logic Programs with Distribution Semantics, Proc. of ICLP'95, pp715-729, 1995.

[19] Sato,T., and Kameya,Y. PRISM:A Language for Symbolic-Statistical Modeling, Proc. of IJCAI'97, pp.1330-1335, 1997.

[20] Tanner,M., Tools for Statistical Inference (2nd ed.), Springer-Verlag, 1986.

[21] H.C.White, An Anatomy of Kinship, Prentice-Hall INC., 1963.