CHRiSM: Chance Rules induce Statistical Models

Jon Sneyers, Wannes Meert, and Joost Vennekens

K.U.Leuven, Belgium
`Firstname.Lastname@cs.kuleuven.be`

Abstract. A new probabilistic-logic formalism, called CHRiSM, is introduced. CHRiSM is based on a combination of CHR and PRISM. It can be used for high-level rapid prototyping of complex statistical models by means of chance rules. The underlying PRISM system can then be used for several probabilistic inference tasks, including parameter learning. We describe a source-to-source transformation from CHRiSM rules to PRISM, via CHR(PRISM). Finally we discuss the relation between CHRiSM and probabilistic logic programming, in particular, CP-logic.

1 Introduction

Constraint Handling Rules [1, 2] is a high-level language extension based on multi-headed rules. We assume the reader to be familiar with CHR.

The idea of adding probabilities to CHR is not new. In [3], a probabilistic variant of CHR, called PCHR, was introduced. It was implemented by means of a source-to-source transformation [4]. In PCHR, every rule gets a weight which represents its relative probability. A rule is chosen randomly from all applicable rules, according to a probability distribution given by the normalized weights. For example, the following PCHR program implements a fair coin toss:

```
toss <=>0.5: head.
toss <=>0.5: tail.
```

One of the conceptual advantages of PCHR, at least from a theoretical point of view, is that its semantics instantiates the abstract operational semantics ω_t of CHR [2]: every PCHR derivation corresponds to some ω_t derivation.

However, the semantics of PCHR may also lead to some confusion, since it is not so clear what the meaning of the rule weight really is. For example, consider again the above coin tossing example. For the query `toss` we get the answer `head` with 50% chance and otherwise `tail`, so one may be tempted to interpret weights as rule probabilities. However, if the second rule is removed from the program, we will not get the answer `head` with 50% chance, but with a probability of 100%. The reason is that the weights are normalized w.r.t. the sum of the weights of all applicable rules. As a result of this normalization, the actual probability of a rule can only be computed at runtime and by considering the full program. In other words, the probabilistic meaning of a single rule heavily depends on the rest of the PCHR program; there is no localized meaning.

Also, adding weights to propagation rules usually does not make a lot of sense. Consider for example this program:

```
toss ==>0.5: head.
toss ==>0.5: tail.
```

For the above program, the result of the query `toss` will always be `toss, head, tail` (not necessarily in that order): the first propagation rule to fire is chosen randomly, and then the other one has to fire because it is the only applicable rule that is left (so its weight effectively does not matter).

The abstract semantics ω_t of CHR can be instantiated to allow more execution control and more efficient implementations. The best-known example of such an instantiation is the refined semantics ω_r [5]. However, the semantics of PCHR, even though it conforms to ω_t (all PCHR derivations are also ω_t derivations), cannot be instantiated in an analogous way. The reason is that the semantics of PCHR refers to all applicable rules in order to randomly pick one. This conflicts fundamentally with the purpose of instantiations like the refined semantics, which consider only a small portion of the set of applicable rules (i.e., only the rules corresponding to the current active constraint occurrence).

The above issues indicate that perhaps some further investigation of the combination of probabilities and CHR could be useful. In this paper we introduce a different probabilistic variant of CHR, which we call CHRiSM. Its semantics rather differs from that of PCHR and its derivations do not correspond exactly to ω_t derivations (in particular, CHRiSM derivations are partial ω_t derivations). However, the semantics of CHRiSM can be instantiated since it does not refer to the set of all applicable rules. As a result, it can be implemented more efficiently and more execution control can be obtained. Additionally, in the CHRiSM semantics, the rules have a localized meaning: the probabilities do not depend on whether or not other rules are applicable.

CHRiSM can be implemented easily given a CHR(PRISM) system, that is, a CHR system for the host language PRISM. PRISM (PRogramming In Statistical Modeling) is an extension of Prolog with probabilistic built-ins [6]. It has built-in support for several probabilistic inference tasks, including sampling, probability computation, and an expectation-maximization (EM) learning algorithm. These features directly transfer to CHRiSM. These features, in particular EM-learning, are an additional significant advantage of CHRiSM over PCHR: the latter only supports probabilistic execution, i.e. sampling. A state-of-the-art CHR(PRISM) system is currently not available and beyond the scope of this paper. We will however present a prototype implementation of CHRiSM that uses a naive CHR(PRISM) system based on `toychr`.

2 Syntax and Semantics of CHRiSM

CHRiSM extends CHR with two probabilistic statements: it it possible to specify the probability of an entire rule and of the disjuncts in a disjunction.

2.1 Syntax and Informal Semantics

Formally, a CHRiSM program $\mathcal{P}$ consists of a sequence of chance rules. A chance rule is of the following form:

```
P ?? Hk \ Hr <=> G | B.
```

where `P` is a probability expression (as defined below), `Hk` is a conjunction of (kept head) constraints, `Hr` is a conjunction of (removed head) constraints, `G` is a guard condition (a Prolog goal to be satisfied), and `B` is the body of the rule. If `Hk` is empty, the rule is a simplification rule and the backslash is omitted; if `Hr` is empty, the rule is a propagation rule and it is written as "`P ?? Hk ==> ...`".

The meaning of such a chance rule is that, whenever this rule *could* in principle be applied to rewrite its head, this will only happen with a probability given by `P`. Formally, the probability expression `P` is one of the following:

– A number between 0 and 1, indicating the probability that the rule fires. A rule of the form `1 ??` is a normal CHR rule; the "`1 ??`" may be dropped.
– An expression of the form `eval(E)`, where `E` is an arithmetic expression (in Prolog syntax) which may involve variables from the head and guard, which should be fully instantiated when the rule is applicable. The evaluated expression indicates the probability that the rule fires.
– Omitted (so the rule starts with "`??`"): this indicates that the rule probability is unknown.
– A set of variables `V1,...,Vn`: the probability is unknown and it is parametrized in (i.e., the distribution may depend on) the values of `V1,...,Vn`.

Initially, unknown probabilities are set to a uniform distribution (`0.5` in the case of rule probabilities). They can be changed by PRISM's EM-learning algorithm.

Next to these probability assignments to entire rules, CHRiSM also allows to assign probabilities to the disjuncts of a disjunction; that is, the rule body `B` is a conjunction of any of the following:

– Prolog goals and/or CHRiSM constraints (just like in regular CHR);
– $\mathrm{CHR}^{\vee}$-style disjunction [7] (i.e., Prolog disjunction, with backtracking);
– LPAD-style probabilistic disjunction [8] (LPAD: Logic Programs with Annotated Disjunctions), of the form

  ```
  D1:P1 ; ... ; Dn:Pn,
  ```

  where the disjuncts `D`i are rule bodies, and a disjunct `D`i is chosen (committed-choice) with probability `P`i. The probabilities should sum to 1.

In case the probabilities `P`i of the disjuncts are not known, one can also write:

```
P ?? D1 ; ... ; Dn,
```

where `P` can either be omitted (to denote that the probability distribution over the disjuncts is unknown), or a conjunction of variables `V1,..., Vn` (to denote that this distribution is unknown and dependent on the values of the `V`i).

2.2 Operational Semantics

The abstract operational semantics $\omega_t^{??}$ of a CHRiSM program $\mathcal{P}$ is given by a state-transition system that resembles the abstract operational semantics of CHR$^\vee$. In particular, the execution states are defined analogously, except that we additionally define a unique failed execution state which is denoted by "*fail*" (to prevent further rule applications after failure). We refer the reader to [2] for a definition of ω_t execution states. The transitions of $\omega_t^{??}$ are listed in Fig. 1. Note that the first four transitions, together with the last one where `P=1`, correspond to the usual semantics of CHR$^\vee$ (with the minor difference that $\omega_t^{??}$ has an explicit **Fail** transition). We use $\uplus$ for multiset union, and $\bar{\exists}_A B$ to denote $\exists x_1, \ldots, x_n : B$, with $\{x_1, \ldots, x_n\} = \mathit{vars}(B) \setminus \mathit{vars}(A)$, where $\mathit{vars}(A)$ are the (free) variables in A (if A is empty it may be omitted).

1. **Fail.** $\langle \{b\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \mathit{fail}$
   where b is a built-in (Prolog) constraint and $\mathcal{D}_{\mathcal{H}} \models \neg\bar{\exists}(\mathbb{B} \wedge b)$.
2. **Solve.** $\langle \{b\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{G}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$
   where b is a built-in (Prolog) constraint and $\mathcal{D}_{\mathcal{H}} \models \bar{\exists}(\mathbb{B} \wedge b)$.
3. **Introduce.** $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$
   where c is a CHRiSM constraint.
4. **Split.** $\langle \{d\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \{d_1\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mid \ldots \mid \langle \{d_k\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
   where d is a CHR$^\vee$ disjunction of the form d_1 `;` ... `;` d_k.
5. **Probabilistic Choice.** $\langle \{d\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \{d_i\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
   where d is a probabilistic disjunction of the form `P ??` d_1 `;` ... `;` d_k or of the form $d_1 {:} p_1$ `;` ... `;` $d_k {:} p_k$. One disjunct d_i is chosen probabilistically according to a distribution parametrized in `P` or given by $p_1, \ldots, p_k$.
6. **Maybe-Apply.** $\langle \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} X$
   where the r-th rule of $\mathcal{P}$ is of the form `P ??` H_1' `\` H_2' `<=> G | B`,
   θ is a matching substitution such that $\mathit{chr}(H_1) = \theta(H_1')$ and $\mathit{chr}(H_2) = \theta(H_2')$, $h = (r, \mathit{id}(H_1), \mathit{id}(H_2)) \notin \mathbb{T}$, and $\mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}(\theta \wedge G)$. With a probability determined by `P`, the resulting state $X = \langle B \uplus \mathbb{G}, H_1 \uplus \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$. Otherwise, the resulting state $X = \langle \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$.

Fig. 1. Transition relation $\rightarrowtail_{\mathcal{P}}$ of the abstract operational semantics $\omega_t^{??}$ of CHRiSM.

In addition to this very nondeterministic abstract operational semantics $\omega_t^{??}$, we can also define more deterministic instantiations of $\omega_t^{??}$, just like ω_r and ω_p are instantiations of ω_t (see also [9]). In the following, we will use a "refined semantics of CHRiSM", defined analogously to [5]. Of course CHRiSM can also be given a "priority semantics", analogous to [10], to get a more intuitive mechanism for execution control.

Any CHRiSM system uses a (computable) execution strategy in the sense of [9]. In the examples in the next section we will assume that this execution strategy is within the strategy class corresponding to the refined semantics of CHRiSM. Note that in [9], an execution strategy completely fixes the derivation for a given input goal. In the context of CHRiSM this is no longer the case because of the probabilistic choices. However, we may assume that the derivation is fixed if the same choices are made. In other words, the only choice is in the probabilistic choices inside the transitions "**Probabilistic Choice**" and "**Maybe-Apply**"; there is no nondeterminism in choosing which $\omega_t^{??}$ transition to apply next.

The CHRiSM built-in `Q <==> A` denotes that there exist a series of probabilistic choices such that (under the particular execution strategy which is used) a derivation starting with query `Q` results in the answer `A`.

The built-in `Q ===> A` denotes that the answer for query `Q` contains at least `A`: `Q ===> A` holds if `Q <==> B` with `A` $\subseteq$ `B`. In both cases, `Q` and `A` are conjunctions of ground CHRiSM constraints. We also allow "negated" CHRiSM constraints in the right hand side: `Q ===> A,~N` is a shorthand for `Q <==> B` with `A` $\subseteq$ `B` and `N` $\not\subseteq$ `B`.
The following PRISM built-ins can be used when querying a CHRiSM program:

- `sample Q` : probabilistically execute the query `Q`;
- `prob Q <==> A` : compute the probability that `Q <==> A` holds, i.e. the chance that the choices are such that query `Q` results in answer `A`;
- `prob Q ===> A` : compute the probability that the answer for `Q` contains `A`;
- `learn(L)` : perform EM-learning based on a list `L` of observations about derivations (`count/2` can be used to denote multiple identical observations);
- `learn` : perform EM-learning using the facts in a data file whose location can be set using the directive `set_prism_flag(data_source,file(Filename))`.

3 CHRiSM by Example

In this section we illustrate some of the features of CHRiSM by example. As a first toy example, consider the following CHRiSM program for tossing a coin:

```
toss <=> head:0.5 ; tail:0.5.
```

The query "`sample toss`" results in `head` or `tail`, with 50% chance each.

3.1 Rock-paper-scissors

Now consider the following slightly less trivial CHRiSM program to simulate naive "rock-paper-scissors" players:

```
player(P) <=> P ?? rock(P) ; scissors(P) ; paper(P).
rock(P1), scissors(P2) ==> winner(P1).
scissors(P1), paper(P2) ==> winner(P1).
paper(P1), rock(P2) ==> winner(P1).
```

We assume here that each player has his own fixed probability distribution for choosing between rock, scissors, and paper. This is denoted by using `P` as the probability expression for the choice in the first rule: the probability distribution depends on the value of `P` and thus every player has his own distribution.

However, these distributions are not known to us. By default, the unknown probability distributions for, say, `tom` and `jon` are therefore both set to the uniform distribution, which implies, among other things, that each player should win one third of the time. Here is a possible interaction: (user input is in **`bold`**)

```
| ?- sample player(tom),player(jon)
player(tom),player(jon) <==> rock(jon),rock(tom).
| ?- sample player(tom),player(jon)
player(tom),player(jon) <==> rock(jon),paper(tom),winner(tom).
| ?- prob player(tom),player(jon) ===> winner(tom)
Probability of player(tom),player(jon)===>winner(tom) is: 0.333333
```

Now suppose that we watch 100 games, and want to use our observations to obtain a better model of the playing style of both players. If we can fully observe these games, then this is easy: we can just use the frequency with which each player played rock, paper or scissors as an estimate for the probability of him making that particular move. The situation becomes more difficult, however, if the games are only partly observable. For instance, suppose that we do not know which moves the players made, but are only told the final scores: `tom` won 50 games, `jon` won 20, and 30 games were a tie. Deriving estimates for the probabilities of individual moves from this information is less straightforward. For this reason, PRISM comes with a built-in implementation of the EM-algorithm for performing parameter estimation in the presence of missing information [11]. We can use this algorithm to find plausible corresponding distributions:

```
| ?- learn([ count((player(tom),player(jon) ===> winner(tom)),50),
             count((player(tom),player(jon) ===> winner(jon)),20),
count((player(tom),player(jon) ===> ~winner(tom),~winner(jon)),30)])
...
```

The PRISM built-in `show_sw` shows the learned probability distributions, which do indeed correspond (approximately) to the observation frequencies, e.g.:

```
| ?- show_sw
Switch exp1(jon): 1 (p: 0.600570) 2 (p: 0.065368) 3 (p: 0.334061)
Switch exp1(tom): 1 (p: 0.084208) 2 (p: 0.209736) 3 (p: 0.706054)
| ?- prob player(tom),player(jon) ===> winner(tom)
Probability of player(tom),player(jon)===>winner(tom) is: 0.499604
```

3.2 Monopoly dice rolling

The following CHRiSM program implements dice rolling in the Monopoly game. Two dice rolls determine the number of steps to go forward. As long as doubles are rolled, the player may roll again. However, if doubles are rolled three times, the player must go to jail. Note that this program makes use of the refined semantics of CHRiSM: the last rule (roll again) may only be fired if the penultimate rule (go to jail) is not applicable.

```
   go <=> roll, roll.
   roll <=> ?? die(1);die(2);die(3);die(4);die(5);die(6).
   die(X) ==> steps(X).
   steps(X), steps(Y) <=> Z is X+Y, steps(Z).
   die(X), die(X) <=> again.
   again, again, again <=> jail.
   again ==> go.
```

We assume that the dice are fair, so the default uniform distribution is correct. If given enough observations, this assumption can be tested by learning the dice probability distribution. If needed we can also distinguish between the two dice:

```
   go <=> roll(die_A), roll(die_B).
   roll(D) <=> D ?? die(1);die(2);die(3);die(4);die(5);die(6).
```

Here is an example interaction:

```
| ?- sample go
go <==> die(3),die(4),steps(7).
| ?- prob go ===> steps(10)
Probability of go===>steps(10) is: 0.063571
| ?- prob go ===> jail
Probability of go===>jail is: 0.004629
| ?- prob go ===> ~again, ~jail
Probability of go===>~again,~jail is: 0.833333
```

3.3 Random graphs

Suppose we want to generate a random directed graph, given its nodes. The following rule generates every possible directed edge with probability 50%:

```
   0.5 ?? node(A), node(B) ==> edge(A,B).
```

The above rule of course generates dense graphs; if we want to get a sparse graph, say with an average (out-)degree of 3, we can use the following rule. The auxiliary constraint nb_nodes(n) contains the total number of nodes n; the probability of the rule is such that each of the $n(n-1)$ possible edges is generated with probability $3/(n-1)$, so we can expect that on average it will generate about $3n$ edges:

```
   eval(3/(N-1)) ?? nb_nodes(N), node(A), node(B) ==> edge(A,B).
```

3.4 Bayesian networks

Bayesian networks are one of the most widely used kinds of probabilistic models. A classical example [12] of a Bayesian network is that describing the following alarm system (see Figure 2). Suppose there is some probability that there is a burglary, and also that there is some probability that an earthquake happens.

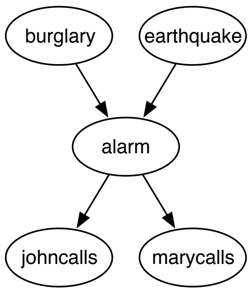

Fig. 2. Bayesian network for alarm system

The probability that the alarm goes off depends on whether those events happen. Also, the probability that John calls the police depends on whether the alarm went off, and similarly for the probability that Mary calls.

This Bayesian network can be described in CHRiSM in a straightforward way:

```
go ==> ?? burglary(yes) ; burglary(no).
go ==> ?? earthquake(yes) ; earthquake(no).
burglary(B), earthquake(E) ==> B,E ?? alarm(yes) ; alarm(no).
A ?? alarm(A) ==> johncalls.
A ?? alarm(A) ==> marycalls.
```

and the probability distributions can be estimated given a number of full observations like

```
go <==> go, burglary(no), earthquake(yes), alarm(yes), marycalls.
go <==> go, burglary(no), earthquake(no), alarm(no).
```

or given partial observations like

```
go ===> johncalls, ∼marycalls.
alarm(yes) ===> johncalls.
```

In this way, each Bayesian network can be represented in CHRiSM. We can derive the same information from it as can be derived from the network itself.

4 Prototype Implementation and Future Work

A prototype CHRiSM system was implemented.[1] It is based on a source-to-source transformation to CHR(PRISM) rules. PRISM is implemented on top of B-Prolog, and several advanced CHR systems are currently available for B-Prolog (e.g. [13] and [14]). However, these advanced CHR systems typically make use of non-pure Prolog constructs (e.g. global variables and destructive updates in [13], Action Rules in [14]) which cause some problems in PRISM. Therefore,

[1] Download the CHRiSM system at `http://www.cs.kuleuven.be/∼jon/chrism/`

as a simple workaround, the prototype CHRiSM system is based on `toychr`[2], a rather naive implementation of (ground) CHR in pure Prolog, which works fine in PRISM.

4.1 PRISM

PRISM is a probabilistic logic programming language. More precisely: PRISM is Prolog extended with a probabilistic built-in *multi-valued random switch (msw)*. A multi-valued switch atom `msw(term, Result)` represents a probabilistic experiment named `term` (a ground Prolog term), which produces an outcome `Result`. The set of possible outcomes for such an experiment is defined by means of a predicate `values(term,[v1,..., vn])`. By default, a uniform distribution is assumed (all values are equally likely). Different probabilities can be assigned by means of `set_sw(term, [p1, ..., pn])`.

A PRISM program consists out of two parts, rules R and facts F. The facts F define a *base probability distribution* P_F on `msw`-atoms, by means of the `values` and `set_sw` predicates. The rules R are a set of definite clauses, which are allowed to contain the `msw` predicate in the body (but not in the head). This set of clauses R serves to extend the base distribution P to a distribution $P_{DB}(\cdot)$ over the set of Herbrand interpretations: for each interpretation M of the `msw` terms, the probability $P_F(M)$ is assigned to the interpretation I that is the least Herbrand model of $R \cup M$ (*distribution semantics*).

HMM Example. A 2-state HMM is modeled with PRISM as an example. Consider a very simple left-to-right HMM with two states $\{s_0, s_1\}$. s_0 is the initial state and the next state is again s_0 or s_1 which is the end state. In each state, the HMM outputs a symbol either 'a' or 'b'.

```
values(tr(s0), [s0, s1]).
values(out(_), [a,b]).

hmm(Cs) :- hmm(0,s0,Cs).
hmm(T,s1,[C]) :- msw(out(s1),T,C). % Final state
                                   % output symbol and terminate
hmm(T,S,[C|Cs]) :-  S\==s1,        % Not the final state
  msw(out(S),T,C),                 % output symbol
  msw(tr(S),T,Next),               % go to next state
  T1 is T+1,
  hmm(T1,Next,Cs).
```

The first two clauses are declarations to indicate the possible values (the facts). `values`(i, V_i) says that V_i is a list of possible values the switch i can take. The remaining clauses define the probability distribution on the strings generated by the HMM (the rules). `hmm`(Cs) denotes a probabilistic event that the HMM generates a string Cs. `hmm`(T, S, Cs') denotes that the HMM, whose state is S at time T, generates a substring Cs' from that time on.

[2] by Gregory J. Duck, 2004. Download: `http://www.cs.mu.oz.au/∼gjd/toychr/`

Alarm System Example. In PRISM the Bayesian network describing the alarm system given before is modeled as follows:

```
values(_,[yes,no]). % all variables are boolean
world(Fire,Earthquake,Alarm,JohnCalls,MaryCalls) :-
   msw(fire,Fire),
   msw(earthquake,Earthquake),
   msw(alarm(Fire,Earthquake),Alarm),
   msw(johncalls(Alarm),JohnCalls),
   msw(marycalls(Alarm),MaryCalls).
```

If the goal is to learn the probabilities from data, this suffices; otherwise, the probabilities can be set explicitly, for example:

```
:- set_sw(johncalls(yes), [0.9, 0.1]).
:- set_sw(johncalls(no),  [0.1, 0.9]).
...
```

4.2 Transformation to CHR(PRISM)

The transformation from CHRiSM to CHR(PRISM) is rather straightforward. We illustrate it by example. Consider again the rule

```
player(P) <=> P ?? rock(P) ; scissors(P) ; paper(P).
```

from Section 3.1. It is translated to the following CHR(PRISM) code:

```
values(experiment1(_),[1,2,3]).
player(P) <=> msw(experiment1(P),X),
              (X=1->rock(P); X=2->scissors(P); X=3->paper(P)).
```

Another example is the random graph rule from Section 3.3:

```
eval(3/(N-1)) ?? nb_nodes(N), node(A), node(B) ==> edge(A,B).
```

which gets translated to the following CHR(PRISM) code:

```
values(experiment1,[1,2]).
nb_nodes(N), node(A), node(B) ==>
     P1 is 3/(N-1), P2 is 1-P1, set_sw(experiment1,[P1,P2]),
     msw(experiment1,X),
     (X=1->edge(A,B); X=2->true).
```

Probabilistic simplification rules and simpagation rules are a bit more tricky since it does not suffice to add a “nop”-disjunct like above. Putting the `msw`-test in the guard of the rule also does not work as expected. In sampling mode, this works fine, but when doing probability computations or learning, an unwanted behavior emerges because of the way PRISM implements explanation search. During explanation search, PRISM essentially redefines `msw/2` such that it creates a choice point and tries all values. This causes the guard to always succeed and thus explanations that involve *not* firing a chance rule are erroneously missed. Hence some care has to be taken to translate such rules to PRISM code that behaves correctly. Our solution involves a simple but somewhat ad hoc change to the `toychr` compiler.

In order to further clarify the above issue, consider the PRISM code generated for a CHR rule of the form `Head <=> Guard | Body`. The generated code consists of a series of PRISM clauses, where one clause ends like this:

```
...
( Guard ->
      RemoveConstraints,
      Body
;     Continue ).
```

where `RemoveConstraints` is the code responsible for removing the constraints of `Head` from the constraint store, and `Continue` is the code to try the next applicable rule (in case the guard failed). Clearly, if the `msw`-test is put in the body, we get the problem that the head constraints are removed even if the chance rule was "not applied". If the `msw`-test is put in the guard, the `Continue` branch is never entered when PRISM does explanation search. Instead, we generate code of the following form, which does result in the right behavior:

```
...
( Guard ->
      msw(experimentk,X),
      ( X = 1 ->
            RemoveConstraints,
            Body
      ;     Continue )
; Continue ).
```

4.3 Future work

Our prototype implementation can obviously be improved and extended in many ways. We mention a few general directions for future work:

- The computational performance of `toychr` does not suffice for programs that are not toy examples. It will thus be necessary to get a more advanced CHR system to cooperate with PRISM. One of the obstacles is that PRISM heavily relies on tabling for efficiency [15]; some work has already been done on CHR and tabling in the context of XSB (e.g. [16] and [17]), which may be transferable to PRISM/B-Prolog.
- So far we have only considered ground CHRiSM programs, that is, all constraint arguments are ground at runtime. It is not straightforward to add support for non-ground programs and queries, but it would certainly be useful to have (at least) support for queries like

```
| ?- prob player(tom),player(jon) ===> winner(_)
| ?- prob go ===> steps(S), S > 20
```

– In [3], the notion of probabilistic termination was explored for PCHR programs. Consider for example the Monopoly example. This program always terminates for the query `go`, since the number of re-rolls is at most two. Suppose we remove the "go to jail" rule. Now the program is nonterminating: if we keep rolling doubles, it never ends. However, it is probabilistically terminating since the probability of terminating is 1 (the probability of non-termination is $(1/6)^\infty = 0$). The probability of some derivation, say `prob go ===> steps(20)` can still be computed and learning from a set of observations still makes sense for programs that only terminate probabilistically; however PRISM cannot handle such programs because it will go in an infinite loop during explanation search (or more precisely, run out of stack space). This is not just an issue in CHRiSM but also in PRISM.

5 Relation to Probabilistic Logic Programming

There exist numerous probabilistic extensions of logic programs. One particular family of such extensions is formed by CP-logic or LPADs [18], ProbLog [19], ICL [20], and PRISM itself [6]. All of these can be easily transformed to CHRiSM. Let us consider the case of CP-logic. A theory in this language consists of a set of normal logic programming rules with probabilistically quantified disjunctions in the head; i.e., a CP-logic rule r is of the form:

$$(h_1 : \alpha_1) \vee \cdots \vee (h_n : \alpha_n) \leftarrow \phi$$

with the h_i propositions, the α_i probabilities (with $\sum \alpha_i = 1$) and ϕ a formula. Such a rule expresses that if the formula ϕ is satisfied, a random event will take place, which causes one of the h_i; each α_i represent the probabilities that the associated h_i is the atom that is in fact caused.

We can translate such a rule to CHRiSM by introducing some new symbols. We introduce a symbol $\texttt{b}_r$ to denote that the body ϕ of this rule r is satisfied, and for each h_i in its head, we introduce a symbol $\texttt{c}_r^{h_i}$ to represent that h_i is the atom caused by r. For all the symbols s that we now have, we also introduce a symbol $\texttt{not\_}s$ to represent its negation. We can then translate the above CP-logic rule as:

$$\texttt{b}_r \;\texttt{<=>}\; \alpha_1\texttt{:}\texttt{c}_r^{h_1} \;\texttt{;}\; \texttt{...} \;\texttt{;}\; \alpha_n\texttt{:}\texttt{c}_r^{h_n}$$

and

$$\texttt{not\_b}_r \;\texttt{<=>}\; \texttt{not\_c}_r^{h_1}\texttt{,} \texttt{...} \texttt{,}\; \texttt{not\_c}_r^{h_n}.$$

In CP-logic, each rule can "fire" at most once. To avoid that the multiset semantics of CHR would give a different result, we add $\texttt{b}_r\backslash\ \texttt{b}_r$ `<=> true` to the top of the CHRiSM program.

To relate the $\texttt{c}_r^{h_i}$ and $\texttt{not\_c}_r^{h_j}$, we use: $\texttt{c}_r^{h_i}$ `==>` $\texttt{not\_c}_r^{h_j}$, for all $i \neq j$.

The semantics of CP-logic has a non-monotonic aspect, which stems from its use of "negation-as-failure" as in the well-founded semantics for logic programs. This can be captured in CHRiSM by expressing that a atom h is true if and only if it is caused by at least one rule r_i:

$$
\begin{array}{c}
\mathtt{c}^{h}_{r_1}\ \mathtt{==>}\ \mathtt{h} \\
\vdots \\
\mathtt{c}^{h}_{r_n}\ \mathtt{==>}\ \mathtt{h} \\
\mathtt{not\_c}^{h}_{r_1}\mathtt{,\ \ldots\ ,\ not\_c}^{h}_{r_n}\ \mathtt{==>}\ \mathtt{not\_h}.
\end{array}
$$

Here, $r_1, \ldots, r_n$ are all the rules that have h in their head.

All that remains is to define the $\mathtt{b}_r$ and $\mathtt{not\_b}_r$ in such a way that they correctly correspond to the truth of the bodies of the rules r. This requires us to encode the logical connectives in CHR. If we push negation down to the atom level, we can simply replace each $\neg h$ with `not_h`. Encoding the meaning of $\vee$ and $\wedge$ is straightforward.

We have now shown that CP-logic can be encoded in CHRiSM in a compact and modular way. It follow that we can do the same for sublogics of CP-logic, such as ProbLog, ICL, and PRISM.

Next to these "logic programming flavored" languages, there also exist a number of formalisms that are inspired by Bayesian networks, such as BLP [21], RBN [22], CLP(BN) [23], and Blog [24]. Based on the encoding of Bayesian networks that we gave in Section 3.4, we can also translate BLPs to CHRiSM. RBNs, CLP(BN) and Blog would be more difficult, because they allow more complex probability distributions, for which CHRiSM currently does not offer support.

As the above paragraphs show, we can encode CP-logic or BLPs using only the `D1:P1; ...; Dn:Pn` construction of CHRiSM. In particular, the ability to assign a probability to an entire rule is not used. This construct offers some expressivity that these other language do not have; namely, it allows to explicitly make things false. For instance, suppose we want to model the evolution of a population. The fact that an individual might die could be represented in CHRiSM as: `0.1 ?? alive <=> true`. In a language such as CP-logic, this would have to be encoding in a roundabout way, using explicit time points and a probabilistic law of persistence:

$$(alive(T+1) : 0.9) \leftarrow alive(T).$$

The CHRiSM representation is more compact and elegant.

6 Conclusion

In this exploratory paper, we have introduced a novel rule-based probabilistic-logic formalism called CHRiSM, which is based on a combination of CHR and PRISM. We have introduced its syntax and semantics and illustrated them with a few example programs. These examples, being only toy examples, show only a glimpse of the power, elegance, and expressiveness of CHRiSM's chance rules. Just like CHR has important advantages over Prolog (a.o., multi-headedness), CHRiSM has the same advantages over PRISM.

When comparing CHRiSM to PCHR, the authors consider the most important difference to be that CHRiSM has a cleaner and more natural semantics. In contrast to PCHR, the probabilistic meaning of CHRiSM's chance rules is local, that is, it does not depend on the full program and runtime information (the applicability of other rules). This difference between PCHR and CHRiSM can also be approached from the point of view of execution control: while PCHR can only be defined in the framework of a very nondeterministic abstract operational semantics, CHRiSM can also be given a refined operational semantics or a priority semantics.

Another practical advantage of CHRiSM over PCHR is that there are many features inherited from PRISM: the support for computing probabilities, learning from examples, etc. These features essentially come for free — although from the system implementation point of view, the combination of CHR and PRISM also introduces several challenges, most of which have not been tackled yet in this exploratory work.

We described a naive prototype implementation and identified some issues and directions for future work. We also showed how CP-logic can be embedded in CHRiSM. CP-logic (and, by virtue of the embedding of CP-logic in CHRiSM, also CHRiSM) can express many other probabilistic logic formalisms (including ProbLog, ICL, and PRISM).

This paper obviously only scratches the surface of the research questions associated with CHRiSM.

Acknowledgments

Jon Sneyers and Wannes Meert are funded by Ph.D. grants of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Joost Vennekens is a postdoctoral researcher of FWO Vlaanderen.

References

1. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
2. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules — a survey of CHR research between 1998 and 2007. Theory and Practice of Logic Programming (2009)
3. Frühwirth, T., Di Pierro, A., Wiklicky, H.: Probabilistic Constraint Handling Rules. In: 11th Intl. Workshop Funct. and (Constraint) Logic Progr. ENTCS 76 (2002)
4. Frühwirth, T., Holzbaur, C.: Source-to-source transformation for a class of expressive rules. In Buccafurri, F., ed.: AGP '03: Joint Conf. Declarative Programming APPIA-GULP-PRODE, Reggio Calabria, Italy (September 2003) 386–397
5. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In Demoen, B., Lifschitz, V., eds.: ICLP '04. LNCS, vol. 3132, Saint-Malo, France, Springer (2004) 90–104

6. Sato, T.: A glimpse of symbolic-statistical modeling by PRISM. Journal of Intelligent Information Systems **31** (2008)
7. Abdennadher, S., Schütz, H.: CHR$^\vee$, a flexible query language. In Andreasen, T., Christiansen, H., Larsen, H., eds.: FQAS '98: Proc. 3rd Intl. Conf. on Flexible Query Answering Systems. Volume 1495 of LNAI., Roskilde, Denmark, Springer (May 1998) 1–14
8. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In Demoen, B., Lifschitz, V., eds.: ICLP '04. LNCS, vol. 3132, Saint-Malo, France, Springer (2004) 431–445
9. Sneyers, J., Frühwirth, T.: Generalized CHR machines. In Schrijvers, T., Frühwirth, T., Raiser, F., eds.: CHR '08, Hagenberg, Austria (July 2008)
10. De Koninck, L., Schrijvers, T., Demoen, B.: User-definable rule priorities for CHR. In Leuschel, M., Podelski, A., eds.: 9th International Conference on Principles and Practice of Declarative Programming, Wrocław, Poland, ACM Press (2007) 25–36
11. Kameya, Y., Sato, T.: Efficient EM learning with tabulation for parameterized logic programs. In: Proceedings of the 1st International Conference on Computational Logic (CL2000). Volume 1861 of Lecture Notes in Artificial Intelligence. (2000) 269–294
12. Pearl, J.: Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference. Morgan Kaufmann (1988)
13. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: Implementation and application. In: 1st Workshop on Constraint Handling Rules. (2004)
14. Schrijvers, T., Zhou, N.F., Demoen, B.: Translating Constraint Handling Rules into Action Rules. In: 3rd Workshop on Constraint Handling Rules. (2006)
15. Zhou, N.F., Sato, T., Shen, Y.D.: Linear tabling strategies and optimizations. Theory and Practice of Logic Programming **8**(1) (2008) 81–109
16. Schrijvers, T., Demoen, B., Warren, D.: TCHR: A framework for tabled CHR. Theory and Practice of Logic Programming **8**(4) (2008)
17. Sarna-Starosta, B., Ramakrishnan, C.: Compiling Constraint Handling Rules for efficient tabled evaluation. In Hanus, M., ed.: PADL '07: Proc. 9th Intl. Symp. Practical Aspects of Declarative Languages. Volume 4354 of LNCS., Nice, France, Springer (January 2007) 170–184 System's homepage at `http://www.cse.msu.edu/~bss/chr_d`.
18. Vennekens, J., Denecker, M., Bruynooghe, M.: Representing causal information about a probabilistic process. In: Logics in Artificial Intelligence, 10th European Conference, JELIA'06, Proceedings. Volume 4160 of Lecture Notes in Computer Science., Springer (2006) 452–464
19. Raedt, L.D., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: IJCAI. (2007) 2462–2467
20. Poole, D.: The Independent Choice Logic for modelling multiple agents under uncertainty. Artificial Intelligence **94**(1-2) (1997) 7–56
21. Kersting, K., De Raedt, L.: Bayesian logic programming: Theory and tool. In Getoor, L., Taskar, B., eds.: An Introduction to Statistical Relational Learning, MIT Press (2007)
22. Jaeger, M.: Relational Bayesian networks. In: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97). (1997)
23. Costa, V., Page, D., Cussens, J.: CLP (BN): Constraint logic programming for probabilistic knowledge. Lecture Notes in Computer Science **4911** (2008) 156
24. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D.L., Kolobov, A.: BLOG: Probabilistic models with unknown objects. In Getoor, L., Taskar, B., eds.: Statistical Relational Learning, MIT Press (2007)