TLP **10** (4–6): 433–447, 2010. © Cambridge University Press 2010
doi:10.1017/S1471068410000207

CHR(PRISM)-based probabilistic logic learning

JON SNEYERS, WANNES MEERT, JOOST VENNEKENS
Department of Computer Science, K.U. Leuven, Belgium
(*e-mail:* {jon.sneyers,wannes.meert,joost.vennekens}@cs.kuleuven.be)

YOSHITAKA KAMEYA and TAISUKE SATO
Tokyo Institute of Technology, Japan
(*e-mail:* {kameya,sato}@mi.cs.titech.ac.jp)

submitted 5 February 2010; revised 21 April 2010; accepted 12 May 2010

Abstract

PRISM is an extension of Prolog with probabilistic predicates and built-in support for expectation-maximization learning. Constraint Handling Rules (CHR) is a high-level programming language based on multi-headed multiset rewrite rules.

In this paper, we introduce a new probabilistic logic formalism, called CHRiSM, 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 probability computation and parameter learning. We define the CHRiSM language in terms of syntax and operational semantics, and illustrate it with examples. We define the notion of ambiguous programs and define a distribution semantics for unambiguous programs. Next, we describe an implementation of CHRiSM, based on CHR(PRISM). We discuss the relation between CHRiSM and other probabilistic logic programming languages, in particular PCHR. Finally, we identify potential application domains.

KEYWORDS: probabilistic logic learning, constraint handling rules

1 Introduction

Constraint Handling Rules (Frühwirth 2009; Sneyers *et al.* 2010) is a high-level language extension based on multi-headed rules. Originally, CHR was designed as a special-purpose language to implement constraint solvers, but in recent years it has matured into a general purpose programming language. Being a language *extension*, CHR is implemented on top of an existing programming language, which is called the *host language*. An implementation of CHR in host language X is called CHR(X). For instance, several CHR(Prolog) systems are available.

PRISM (PRogramming In Statistical Modeling) is a probabilistic extension of Prolog (Sato 2008). It supports several probabilistic inference tasks, including sampling, probability computation, and expectation-maximization (EM) learning.

In this paper, we construct a new formalism, called CHRiSM—short for **CH**ance **R**ules **i**nduce **S**tatistical **M**odels. It is based on CHR(PRISM) and it combines the

advantages of CHR and those of PRISM. Like CHR, CHRiSM is a very concise and expressive programming language. Like PRISM, CHRiSM has built-in support for several probabilistic inference tasks. Furthermore, since CHRiSM is implemented as a translation to CHR(PRISM)—which itself is translated to PRISM and ultimately Prolog—CHRiSM rules can be freely mixed with CHR rules and Prolog clauses.

This paper is based on an earlier workshop paper (Sneyers *et al.* 2009). Although it is mostly self-contained, some familiarity with CHR and PRISM is recommended.

We use $\uplus$ for multiset union, $\sqsubseteq$ for multiset subset, and $\bar{\exists}_A B$ to denote $\exists x_1, \ldots, x_n : B$, with $\{x_1, \ldots, x_n\} = vars(B) \setminus vars(A)$, where $vars(A)$ are the (free) variables in A; if A is omitted it is empty (so $\bar{\exists} B$ denotes the existential closure of B).

2 Syntax and semantics of CHRiSM

In this section we define CHRiSM. The syntax is defined in Section 2.1 and the (abstract) operational semantics is defined in Section 2.2. Finally, in Section 2.3 the notion of *observations* is introduced.

2.1 Syntax and informal semantics

A CHRiSM program $\mathcal{P}$ consists of a sequence of *chance rules*. Chance rules rewrite a multiset $\mathbb{S}$ of data elements, which are called (CHRiSM) *constraints* (mostly for historical reasons). Syntactically, a constraint `c(X1,..,Xn)` looks like a Prolog predicate: it has a functor `c` of some arity n and arguments `X1,..,Xn` which are Prolog terms. The multiset $\mathbb{S}$ of constraints is called the *constraint store* or just *store*. The initial store is called the *query* or *goal*, the final store (obtained by exhaustive rule application) is called the *answer* or *result*.

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 called a *simplification* rule and the backslash is omitted; if `Hr` is empty, the rule is called a *propagation* rule and it is written as "`P ?? Hk ==> G | B`". If both `Hk` and `Hr` are non-empty, the rule is called a *simpagation* rule. The guard `G` is optional; if it is removed, the "`|`" is also removed. The body `B` is recursively defined as a conjunction of CHRiSM constraints, Prolog goals, and probabilistic disjunctions (as defined below) of bodies.

Intuitively, the meaning of a chance rule is as follows: If the constraint store $\mathbb{S}$ contains elements that match with the head of the rule (i.e. if there is a matching substitution θ such that $(\theta(\texttt{Hk}) \uplus \theta(\texttt{Hr})) \sqsubseteq \mathbb{S}$), and furthermore, the guard `G` is satisfied, then we can consider rule application. The subset of $\mathbb{S}$ that corresponds to the head of the rule is called a rule *instance*. Depending on the probability expression `P`, the rule instance is either ignored or it actually leads to a rule application. Every rule instance may only be considered once.

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

Rule application has the following effects: the constraints matching `Hr` are removed from the constraint store, and then the body `B` is executed, that is, Prolog goals are called and CHRiSM constraints are added into the store.

Probability expressions. A probability expression `P` is one of the following:

- A number from 0 to 1, indicating the probability that the rule fires. A rule of the form `1 ?? ...` corresponds to a regular CHR rule; the "`1 ??`" may be dropped. A rule of the form `0 ?? ...` is never applied.
- An expression of the form `eval(E)`, where `E` is an arithmetic expression (in Prolog syntax). It should be ground when the rule is considered (otherwise a runtime instantiation error occurs). The evaluated expression indicates the probability that the rule fires.
- An experiment name. This is a Prolog term which should be ground when the rule is considered. The probability distribution is unknown. Initially, unknown probabilities are set to a uniform distribution (`0.5` in the case of rule probabilities). They can be changed manually using PRISM's `set_sw/2` builtin, or automatically using PRISM's EM-learning algorithm.
  The arguments of the experiment name can include *conditions*, which are of the form "`cond C`". Such arguments are evaluated at runtime and replaced by either "`yes`" or "`no`", depending on whether `call(C)` succeeded or failed.
  These conditions are just syntactic sugar, so we may ignore them w.l.o.g. For example, the rule "`foo(cond A>B) ?? c(A,B) <=> d`" is syntactic sugar for "`foo(X) ?? c(A,B) <=> (A>B -> X=yes ; X=no) | d`".
- Omitted (so the rule starts with "`??`"): this is a shorthand for a fresh zero-arity experiment name.

Probabilistic disjunction. The body `B` of a CHRiSM rule may contain probabilistic disjunctions. There are two styles:

- LPAD-style probabilistic disjunctions (Vennekens *et al.* 2004) of the form "`D1:P1 ; ... ; Dn:Pn`", where a disjunct `D`*i* is chosen with probability `P`*i*. The probabilities should sum to 1 (otherwise a compile-time error occurs).
- CHRiSM-style probabilistic disjunctions of the form "`P ?? D1 ; ... ; Dn`", where `P` is an experiment name determining the probability distribution.

The LPAD-style probabilistic disjunctions can be seen as a special case of CHRiSM-style disjunctions for which the experiment name is implicit and the distribution is given and fixed. Unlike $\text{CHR}^{\vee}$ disjunctions, which create a choice point, both kinds of probabilistic disjunctions are *committed-choice*: once a disjunct is chosen, the choice is not undone later. However, when later on in a derivation the same experiment is sampled again, the result can of course be different.

2.2 Operational semantics

The abstract operational semantics of a CHRiSM program $\mathscr{P}$ is given by a state-transition system that resembles the abstract operational semantics ω_t of CHR (Sneyers *et al.* 2010). The execution states are defined analogously, except that we additionally define a unique failed execution state which is denoted by "*fail*" (because

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

1. **Fail.** $\langle \{b\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{1}{\underset{\mathscr{P}}{\rightarrowtail}} \mathit{fail}$
   where b is a built-in (Prolog) constraint and $\mathscr{D}_{\mathscr{H}} \models \neg\bar{\exists}(\mathbb{B} \wedge b)$.
2. **Solve.** $\langle \{b\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{1}{\underset{\mathscr{P}}{\rightarrowtail}} \langle \mathbb{G}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$
   where b is a built-in (Prolog) constraint and $\mathscr{D}_{\mathscr{H}} \models \bar{\exists}(\mathbb{B} \wedge b)$.
3. **Introduce.** $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{1}{\underset{\mathscr{P}}{\rightarrowtail}} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$
   where c is a CHRiSM constraint.
4. **Probabilistic-Choice.** $\langle \{d\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{p_i}{\underset{\mathscr{P}}{\rightarrowtail}} \langle \{d_i\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
   where d is a probabilistic disjunction of the form $d_1 : p_1 \ ; \ \ldots \ ; \ d_k : p_k$ or of the form `P ??` $d_1 \ ; \ \ldots \ ; \ d_k$, where the probability distribution given by `P` assigns the probability p_i to the disjunct d_i.
5. **Maybe-Apply.** $\langle \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n 5;3\ 5;3 \overset{1-p}{\underset{\mathscr{P}}{\rightarrowtail}} \langle \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
   foo $\langle \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{p}{\underset{\mathscr{P}}{\rightarrowtail}} \langle B \uplus \mathbb{G}, H_1 \uplus \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
   where the r-th rule of $\mathscr{P}$ is of the form `P ??` $H_1' \setminus H_2'$ `<=> G | B`,
   θ is a matching substitution such that $chr(H_1) = \theta(H_1')$ and $chr(H_2) = \theta(H_2')$, $h = (r, id(H_1), id(H_2)) \notin \mathbb{T}$, and $\mathscr{D}_{\mathscr{H}} \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}(\theta \wedge G)$. If `P` is a number, then $p =$ `P`. Otherwise p is the probability assigned to the success branch of `P`.

Fig. 1. Transition relation $\overset{}{\underset{\mathscr{P}}{\rightarrowtail}}$ of the abstract operational semantics $\omega_t^{??}$ of CHRiSM.

we don't want to distinguish between different failed states). We use the symbol $\omega_t^{??}$ to refer to the abstract operational semantics of CHRiSM.

Definition 2.1 (*identified constraint*)
An *identified* constraint $c\#i$ is a CHRiSM constraint c associated with some unique integer i. This number serves to differentiate between copies of the same constraint. We introduce the functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences and sets in the obvious manner, e.g., $id(S) = \{i | c\#i \in S\}$.

Definition 2.2 (*execution state*)
An *execution state* σ is a tuple $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. The *goal* $\mathbb{G}$ is a multiset of constraints to be rewritten to solved form. The *store* $\mathbb{S}$ is a set of *identified* constraints that can be matched with rules in the program $\mathscr{P}$. Note that $chr(\mathbb{S})$ is a multiset although $\mathbb{S}$ is a set. The *built-in store* $\mathbb{B}$ is the conjunction of all Prolog goals that have been called so far. The *history* $\mathbb{T}$ is a set of tuples, each recording the identifiers of the CHRiSM constraints that fired a rule and the rule number. The history is used to prevent trivial non-termination: a rule instance is allowed to be considered only once. Finally, the counter $n \in n$ represents the next free identifier.

We use $\sigma, \sigma_0, \sigma_1, \ldots$ to denote execution states and Σ^{CHR} to denote the set of all execution states. We use $\mathscr{D}_{\mathscr{H}}$ to denote the theory defining the host language (Prolog) built-ins and predicates used in the CHRiSM program. For a given CHR program $\mathscr{P}$, the transitions are defined by the binary relation $\overset{}{\underset{\mathscr{P}}{\rightarrowtail}} \subset \Sigma^{\mathrm{CHR}} \times \Sigma^{\mathrm{CHR}}$ shown in Figure 1. Every transition is annotated with a probability.

Execution proceeds by exhaustively applying the transition rules, starting from an initial state (root) of the form $\langle Q, \emptyset, \mathit{true}, \emptyset \rangle_0$ and performing a random walk in the directed acyclic graph defined by the transition relation $\overset{}{\underset{\mathscr{P}}{\rightarrowtail}}$, until a leaf node is reached, which is called a final state. We consider only terminating programs (finite transition graphs). Given a path from an initial state to the state σ, we define

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

the probability of σ to be the product of the probabilities along the path. We use $\sigma_0 \overset{p}{\underset{\mathscr{P}}{\rightarrowtail}}{}^{*} \sigma_k$ to denote a series of $k \geqslant 0$ transitions

$$\sigma_0 \overset{p_1}{\underset{\mathscr{P}}{\rightarrowtail}} \sigma_1 \overset{p_2}{\underset{\mathscr{P}}{\rightarrowtail}} \sigma_2 \overset{p_3}{\underset{\mathscr{P}}{\rightarrowtail}} \ldots \overset{p_k}{\underset{\mathscr{P}}{\rightarrowtail}} \sigma_k$$

where $p = \prod_{i=1}^{k} p_i$ if $k > 0$ and $p = 1$ otherwise. If σ_0 is an initial state and σ_k is a final state, then we call such a series of transitions a *derivation* of probability p. We define a function *prob* to give the probability of a derivation: $prob(\sigma_0 \overset{p}{\underset{\mathscr{P}}{\rightarrowtail}}{}^{*} \sigma_k) = p$.

Note that if all rule probabilities are 1 and the program contains no probabilistic disjunctions—i.e. if the CHRiSM program is actually just a regular CHR program—then the $\omega_t^{??}$ semantics boils down to the ω_t semantics of CHR.

2.3 Full and partial observations

A full observation `Q <==> A` denotes that there exist a series of probabilistic choices such that a derivation starting with query `Q` results in the answer `A`. A partial observation `Q ===> A` denotes that an answer for query `Q` contains at least `A`: in other words, `Q ===> A` holds iff `Q <==> B` with `A` $\trianglelefteq$ `B`.

Definition 2.3 (*observation*)
A *full observation* is of the form `Q <==> A`, where `Q` and `A` are conjunctions of constraints. Given a program $\mathscr{P}$, a full observation refers to derivations of the form

$$\langle Q, \emptyset, true, \emptyset \rangle_0 \overset{p}{\underset{\mathscr{P}}{\rightarrowtail}}{}^{*} \langle \emptyset, A', \mathbb{B}, \mathbb{T} \rangle_n \not\underset{\mathscr{P}}{\rightarrowtail}$$

such that `A` $= chr(A')$. A *partial observation* is of the form `Q ===> A`. It refers to derivations of the above form, such that `A` $\trianglelefteq chr(A')$.

We also allow "negated" CHRiSM constraints in the right hand side:
`Q ===> A,~N` is a shorthand for `Q <==> B` with `A` $\trianglelefteq$ `B` and `N` $\not\trianglelefteq$ `B` $\setminus$ `A`.

The following PRISM built-ins can be used to query 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 an answer for `Q` contains `A`;
- `learn(L)` : perform EM-learning based on a list `L` of observations

In observation lists, the syntax "*n* `times` *X*" or "`count(`*X*`,`*n*`)`" can be used to denote that observation *X* occurred *n* times. This is simply a shorthand for repeating the same observation (*X*) a number of times (*n*).

3 Example programs

As a first toy example, consider the following CHRiSM program for tossing a coin:

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

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

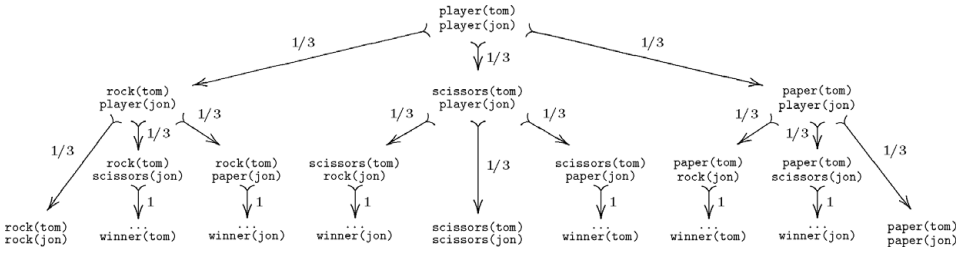

Fig. 2. A derivation tree for the rock-paper-scissors example.

The query "`sample toss`" results in "`head`" or "`tail`", with 50% chance each. The query "`sample toss,toss`" has four possible outcomes, each with 25% chance: "`head,head`", "`head,tail`", "`tail,head`", and "`tail,tail`".

Note that observations are not sensitive to the order in which the result is given. As a result, the query "`prob toss,toss <==> head,tail`" returns a probability of 50%, because the outcome "`tail,head`" also matches the observation.

3.1 Rock-paper-scissors

Consider the following CHRiSM program simulating "rock-paper-scissors" players:

```
player(P) <=> choice(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 that each player has his own fixed probability distribution for choosing between rock, scissors, and paper. This is denoted by using `choice(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 (cf. Figure 2). Here is a possible interaction:

```
?- 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

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

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 (Kameya and Sato 2000). We can use this algorithm to find plausible corresponding distributions:

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

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

```
| ?- show_sw
Switch choice(jon): 1 (p: 0.60057) 2 (p: 0.06536) 3 (p: 0.33406)
Switch choice(tom): 1 (p: 0.08420) 2 (p: 0.20973) 3 (p: 0.70605)
| ?- prob player(tom),player(jon) ===> winner(tom)
Probability of player(tom),player(jon)===>winner(tom) is: 0.499604
```

3.2 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 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 on average it generates $3n$ edges:

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

3.3 Bayesian networks

Bayesian networks are one of the most widely used kinds of probabilistic models. A classical example (Pearl 1988) of a Bayesian network is that describing the following alarm system. Suppose there is some probability that there is a burglary, and also that there is some probability that an earthquake happens. 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.
```

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

The probability distributions can be estimated given full observations (e.g., `go <==> go, burglary(no), earthquake(yes), alarm(yes), marycalls.`), or given partial observations (e.g., `go ===> johncalls, ~marycalls.`).

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 Ambiguity and a distribution semantics for CHRiSM

In addition to the 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 (Sneyers and Frühwirth 2008)). In the current implementation of CHRiSM we use the "refined semantics of CHRiSM", defined analogously to (Duck *et al.* 2004). Of course CHRiSM can also be given a "priority semantics" (De Koninck *et al.* 2007) in order to get a more intuitive mechanism for execution control.

4.1 Instantiations of $\omega_t^{??}$

Any CHRiSM system uses a (computable) execution strategy in the sense of (Sneyers and Frühwirth 2008). Note that in (Sneyers and Frühwirth 2008), 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.

Definition 4.1 (*execution strategy*)
An *execution strategy* fixes the non-probabilistic choices during an $\omega_t^{??}$ derivation. Formally, $\xrightarrow[\xi,\mathscr{P}]{}$ is an execution strategy for a program $\mathscr{P}$ if $\xrightarrow[\xi,\mathscr{P}]{} \subseteq \rightarrowtail_{\mathscr{P}}$ and for every execution state $\sigma \in \Sigma^{\mathrm{CHR}}$, the set S of all transitions of the form $\sigma \xrightarrow[\xi,\mathscr{P}]{} \sigma'$ corresponds to at most one of the five types of transitions of $\omega_t^{??}$, that is, either

- $S = \emptyset$ and no $\omega_t^{??}$ transition is applicable;
- or S is a singleton corresponding to a **Fail**, **Solve** or **Introduce** transition;
- or S is a set of transitions corresponding to the **Probabilistic-Choice** transition for one specific disjunction;
- or S is a set of transitions corresponding to the **Maybe-Apply** transition for one specific rule instantiation.

It follows from this definition that for non-final states σ, the sum of the probabilities of all transitions from σ is one under any execution strategy. We use $\sigma_0 \xrightarrow[\xi,\mathscr{P}]{p}{}^{*} \sigma_k$ to denote a series of $k \geqslant 0$ transitions

$$\sigma_0 \xrightarrow[\xi,\mathscr{P}]{p_1} \sigma_1 \xrightarrow[\xi,\mathscr{P}]{p_2} \sigma_2 \xrightarrow[\xi,\mathscr{P}]{p_3} \dots \xrightarrow[\xi,\mathscr{P}]{p_k} \sigma_k$$

where $p = \prod_{i=1}^{k} p_i$ if $k > 0$ and $p = 1$ otherwise, as before.

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

Definition 4.2 (*strategy class*)
A *strategy class* $\Omega(\mathscr{P})$ is a set of execution strategies for $\mathscr{P}$. The strategy class $\Omega_t^{??}(\mathscr{P})$ is the set of *all* execution strategies for $\mathscr{P}$.

4.2 Distribution semantics

Firstly, we define equivalence of execution states. We use a definition based on (Raiser *et al.* 2009) but adapted to our needs. Intuitively, we say two states are equivalent if the constraint stores are equal and the built-in stores are equivalent; we do not care about identifiers and propagation histories.

Definition 4.3 (*equivalent states*)
Equivalence between execution states is the smallest equivalence relation $\equiv$ s.t.:

1. $\langle \mathbb{G}, \mathbb{S}, x = t \wedge \mathbb{B}, \mathbb{T} \rangle_n \equiv \langle \mathbb{G}, \mathbb{S}[x/t], x = t \wedge \mathbb{B}, \mathbb{T}' \rangle_{n'}$
2. $\langle \mathbb{G}, \mathbb{S}, x = t \wedge \mathbb{B}, \mathbb{T} \rangle_n \equiv \langle \mathbb{G}[x/t], \mathbb{S}, x = t \wedge \mathbb{B}, \mathbb{T}' \rangle_{n'}$
3. $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \equiv \langle \mathbb{G}, \mathbb{S}', \mathbb{B}, \mathbb{T}' \rangle_{n'}$ if $chr(\mathbb{S}) = chr(\mathbb{S}')$
4. $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \equiv \langle \mathbb{G}, \mathbb{S}, \mathbb{B}', \mathbb{T} \rangle_n$ if $\mathscr{D}_{\mathscr{H}} \models \bar{\exists}_{\mathbb{G},\mathbb{S}} \mathbb{B} \leftrightarrow \bar{\exists}_{\mathbb{G},\mathbb{S}} \mathbb{B}'$

We now define the probability of getting some result (given an execution strategy) as the sum of the probabilities of ending up in a final state equivalent with it:

Definition 4.4 (*observation probability*)
Given a program $\mathscr{P}$ and an execution strategy $\xrightarrow[\xi,\mathscr{P}]{} \in \Omega_t^{??}(\mathscr{P})$, we write

$$\sigma_i \overset{p_{tot}}{\underset{\xi,\mathscr{P}}{\Longrightarrow}} \sigma_f$$

if σ_f is a final state and $p_{tot} = \sum_{d \in D} prob(d)$ where $D = \{\sigma_i \overset{p}{\underset{\xi,\mathscr{P}}{\longrightarrow}}{}^{*} \sigma'_f \mid \sigma'_f \equiv \sigma_f\}$. We say that p_{tot} is the probability of observing the result σ_f for the query σ_i.

4.3 Ambiguity

Some programs are *ambiguous* in the sense that they do not define a unique probability distribution over the possible end states. Consider the following example:

```
0.5 ?? a <=> b.
0.5 ?? a <=> c.
```

Suppose the query is "`a`". If we use an execution strategy that starts with the first rule, then with 50% chance this rule is applied and we get the final result "`b`", with 50% chance the second rule is considered resulting in "`c`" with a probability of 25%, and when no rule is applied the result is "`a`" with a probability of 25%. However, if we use an execution strategy that considers the second rule first, then we get a different distribution: "`c`" has a probability of 50%, and "`b`" a probability of 25%.

A program is unambiguous if the probability of an observation does not depend on the execution strategy. The program in the above example is ambiguous in general, but it is unambiguous w.r.t. the refined strategy class. Under the refined semantics, the first rule is always considered first, thus the above program defines only the first probability distribution on final states. In general, we define ambiguity

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

w.r.t. a strategy class—if the strategy class is omitted, we assume it is the most general strategy class corresponding to all execution strategies that instantiate $\omega_t^{??}$.

Definition 4.5 (*unambiguous program*)
A CHRiSM program $\mathscr{P}$ is unambiguous (w.r.t. a strategy class Ω) if, for all states $\sigma_i, \sigma_f \in \Sigma^{\mathrm{CHR}}$ and all execution strategies $\xrightarrow[\xi_1,\mathscr{P}]{}, \xrightarrow[\xi_2,\mathscr{P}]{} \in \Omega$, we have:
if $\sigma_i \overset{p_1}{\underset{\xi_1,\mathscr{P}}{\Longrightarrow}} \sigma_f$ and $\sigma_i \overset{p_2}{\underset{\xi_2,\mathscr{P}}{\Longrightarrow}} \sigma_f$ then $p_1 = p_2$.

The distribution semantics (w.r.t. strategy class Ω) of an unambiguous (w.r.t. Ω) CHRiSM program is defined for every query Q as the probability distribution over the equivalence classes of final states of derivations (of Ω).

Without specification of an execution strategy, ambiguous CHRiSM programs do not have a well-defined meaning—they don't define a unique probability distribution over the final states, but *several* distributions, depending on which execution strategy is used. Ambiguity can be reduced by using a more instantiated strategy class. The current CHRiSM system uses the refined semantics. Many programs that are ambiguous in general are unambiguous w.r.t. the refined strategy class, but not all of them. As a counterexample, consider the program consisting of the rule "`0.5 ?? a, b(X) <=> c(X)`" with the query "`b(1), b(2), a`". There are two ways to execute this program in the refined semantics: one in which the rule instantiation "`a, b(1)`" is considered first, and one in which the rule instantiation "`a, b(2)`" is considered first. According to the first execution strategy, the result is "`c(1), b(2)`" with a probability of 50%, "`c(2), b(1)`" with a probability of 25%, and "`a, b(1), b(2)`" with a probability of 25%. According to the second execution strategy the probabilities of the first two outcomes are switched.

Ambiguity vs. confluence. Ambiguity of CHRiSM programs is related to confluence (Abdennadher *et al.* 1999) of CHR programs. A CHR program is confluent if for every query, all derivations (under the ω_t semantics) lead to equivalent final states. Confluent CHR programs tend to correspond to unambiguous CHRiSM programs. For example, programs with only propagation rules are always confluent and unambiguous. However, confluence and unambiguity do not coincide. For example, a program consisting of the rule "`a <=> b:0.5 ; c:0.5`" is not confluent (because for the query "`a`" it has two non-equivalent final states) but it is unambiguous. Vice versa, some programs are confluent CHR programs while they are ambiguous CHRiSM programs. For example, consider the following program:

```
0.5 ?? a <=> b.
0.5 ?? a <=> c.
0.5 ?? c <=> b.
```

If we ignore the probabilities and consider this as a regular CHR program, then we get a confluent program (all derivations for the query "`a`" end in the result "`b`"). However, as a CHRiSM program, it is ambiguous. If the execution strategy is such that the first rule is considered first for the query "`a`", then the probability of ending up with the result "`b`" is 67.5%. Using an execution strategy that considers the second rule first, the probability of getting "`b`" is only 50%. Therefore, the probability depends on the execution strategy and the program is ambiguous.

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

5 Implementation of CHRiSM

The implementation of CHRiSM is based on a source-to-source transformation from CHRiSM rules to CHR(PRISM) rules. PRISM is implemented on top of B-Prolog, and several CHR systems are currently available for B-Prolog. In (Sneyers *et al.* 2009) we presented a prototype implementation of CHRiSM that used a naive CHR(PRISM) system based on `toychr`[1], which is a rather naive implementation of (ground) CHR in pure Prolog. The current implementation of CHRiSM[2] is based on the more advanced Leuven CHR system (Schrijvers and Demoen 2004).

5.1 PRISM

PRISM (Sato 2008) is a probabilistic logic programming language. It is an extension of Prolog with a probabilistic built-in *multi-valued random switch (msw)*. A multi-valued switch atom `msw(exp, Result)` represents a probabilistic experiment named `exp` (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])` and `term` unifies with `exp`. 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`/2 and `set_sw`/2 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*).

5.2 Transformation to CHR(PRISM)

The transformation from CHRiSM to CHR(PRISM) is rather straightforward and can be done efficiently (linear time). We illustrate it by example. Consider again the rule "`player(P) <=> choice(P) ?? rock(P) ; scissors(P) ; paper(P)`" from Section 3.1. It is translated to the following CHR(PRISM) code:

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

Another example is the random graph rule from Section 3.2:

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

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

[1] by Gregory J. Duck, 2004. Download: `http://www.cs.mu.oz.au/~gjd/toychr/`
[2] Download the CHRiSM system at `http://people.cs.kuleuven.be/jon.sneyers/chrism/`

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

```
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. The reason is that any removed heads are removed from the constraint store as soon as the body is entered, and just reinserting the removed heads potentially causes nontermination. 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. The solution we have adopted is to add a built-in to CHR to explicitly remove a constraint from the head of a rule. All CHRiSM rules are translated to propagation rules. The removed heads are explicitly removed in the body of the rule, but only in the branch in which the rule instance is actually applied.

6 Related work

The idea of a probabilistic version of CHR is not new. In (Frühwirth *et al.* 2002), a probabilistic variant of CHR, called PCHR, was introduced. In PCHR, every rule gets a weight representing a 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 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 (Sneyers *et al.* 2010): 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 do 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 is not very useful in practice.

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

The abstract semantics ω_t of CHR can be instantiated to allow more execution control and more efficient implementations. However, the PCHR semantics, even though it conforms to ω_t, cannot be instantiated in a similar 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 fragment of the set of applicable rules, e.g. only rules for the current active constraint occurrence.

The $\omega_t^{??}$ semantics of CHRiSM differs from that of PCHR. In particular, $\omega_t^{??}$ derivations do not always correspond to ω_t derivations (although they do, in a sense, correspond to *unfinished* ω_t derivations). However, the semantics of CHRiSM can be instantiated since chance rules have a localized meaning: the application probability does not depend on the set of all applicable rules like in PCHR. As a result, it can be implemented efficiently and more execution control can be obtained.

Another advantage of CHRiSM over PCHR are the features inherited from PRISM, in particular probability computation and EM-learning. The existing PCHR implementation only supports probabilistic execution, i.e. sampling.

Probabilistic Logic Programming. There are numerous probabilistic extensions of logic programming. One particular family of such extensions is formed by CP-logic or LPADs, ProbLog, ICL, and PRISM itself (Sato 2008). All of these can be encoded in CHRiSM: in (Sneyers *et al.* 2009) we have shown that CP-logic (of which ProbLog, ICL, etc. are sublogics) can be encoded in CHRiSM in a compact and modular way.

Next to these "logic programming flavored" languages, there are also a number of formalisms that are inspired by Bayesian networks, such as BLP, RBN, CLP(BN), and Blog. Based on the encoding of Bayesian networks that we gave in Section 3.3, 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. (A more detailed description of these formalisms can be found in (Getoor and Taskar 2007).)

7 Potential applications

Both PRISM and CHR have been successfully applied in a wide range of research fields. Since the features of PRISM and CHR are largely orthogonal, we can expect CHRiSM to be extremely suitable for applications at the intersection of the application areas of PRISM and CHR. One example of an application area at the intersection is abduction, which has been studied in the context of PRISM (Sato and Kameya 2002) and also in the context of CHR (Sneyers *et al.* (2010), Section 7.3.2). Computational linguistics and bio-informatics are two other domains in which both PRISM and CHR have proven to be very valuable tools (Sato 2008; Christiansen 2005; Christiansen and Lassen 2009).

Furthermore, in many application domains of CHR, there is clearly a potential for probabilistic extensions of the existing approaches, for instance to deal with uncertain information. Examples are (section numbers refer to Sneyers *et al.* (2010)): scheduling (Section 7.1.1), spatio-temporal reasoning and robotics (Section 7.1.2), multi-agent

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

systems (Section 7.1.3), the semantic web (Section 7.1.4), type systems (Section 7.3.1), testing and verification (Section 7.3.5).

Another interesting application area is the automatic analysis and generation of music. In the past, we have used PRISM to analyse and generate music in a probabilistic setting (Sneyers *et al.* 2006). There are also several deterministic approaches based on constraints and strict rules (e.g. Boenn *et al.* (2008)). Preliminary results indicate that a combined approach, using CHRiSM, is very promising. In this application, sampling of a probabilistic model corresponds to music generation, while parameter learning from a training set corresponds to tuning the model to a specific genre or composer, and probability computation (or Viterbi computation) can be used for music classification.

8 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 defined an operational semantics for arbitrary CHRiSM programs and a distribution semantics for unambiguous CHRiSM programs. We have illustrated the CHRiSM system by example and we have outlined some potential application areas in which CHRiSM can be used. Finally, we have sketched the implementation of the CHRiSM system and discussed related languages, in particular PCHR.

In our opinion, CHR has important advantages over Prolog, including complexity-wise completeness and the expressivity of multi-headed rules. We expect CHRiSM to have the same advantages over plain PRISM.

There are several directions for future work. The notion of ambiguity and its relation to confluence has to be explored; in particular, the existence of a decidable ambiguity test (for terminating CHRiSM programs). Although the current implementation is sufficiently efficient for sampling, it is too naive for probability computation and learning, since those tasks require an efficient mechanism to find explanations (sequences of probabilistic choices) for observations. Improving the efficiency of explanation search is the topic of ongoing work (Sneyers 2010). Another limitation of the current implementation is that it only supports ground queries and observations. Finally, it would be interesting to transfer automatic CHR program generation techniques (e.g. Abdennadher *et al.* (2006)) to CHRiSM in order to obtain a system that supports not only parameter learning but also structure learning (rule learning).

References

Abdennadher, S., Frühwirth, T., and Meuss, H. 1999. Confluence and semantics of constraint simplification rules. *Constraints 4,* 2, 133–165.

Abdennadher, S., Olama, A., Salem, N., and Thabet, A. 2006. ARM: Automatic Rule Miner. In *LOPSTR 2006*, G. Puebla, Ed. Lecture Notes in Computer Science, vol. 4407. Springer, 17–25.

Boenn, G., Brain, M., De Vos, M., and ffitch, J. 2008. Automatic composition of melodic and harmonic music by answer set programming. In *ICLP 2008*, M. Garcia de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, 160–174.

https://doi.org/10.1017/S1471068410000207 Published online by Cambridge University Press

Christiansen, H. 2005. CHR grammars. *TPLP 5(4–5)*, 467–501.

Christiansen, H. and Lassen, O. T. 2009. Preprocessing for optimization of probabilistic-logic models for sequence analysis. In *ICLP 2009*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, 70–83.

De Koninck, L., Schrijvers, T., and Demoen, B. 2007. User-definable rule priorities for CHR. In *PPDP 2007*, M. Leuschel and A. Podelski, Eds. ACM Press, 25–36.

Duck, G. J., Stuckey, P. J., García de la Banda, M., and Holzbaur, C. 2004. The refined operational semantics of Constraint Handling Rules. In *ICLP 2004*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer, 90–104.

Frühwirth, T. 2009. *Constraint Handling Rules*. Cambridge University Press.

Frühwirth, T., Di Pierro, A., and Wiklicky, H. 2002. Probabilistic Constraint Handling Rules. In *WFLP 2002*, M. Comini and M. Falaschi, Eds. Electronic Notes in Theoretical Computer Science 76. Elsevier.

Getoor, L. and Taskar, B., Eds. 2007. *Statistical Relational Learning*. MIT Press.

Kameya, Y. and Sato, T. 2000. Efficient EM learning with tabulation for parameterized logic programs. In *CL 2000*, J. Lloyd, V. Dahl, *et al*., Eds. Lecture Notes in Artificial Intelligence, vol. 1861. Springer, 269–294.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference*. Morgan Kaufmann.

Raiser, F., Betz, H., and Frühwirth, T. 2009. Equivalence of CHR states revisited. In *CHR 2009*, F. Raiser and J. Sneyers, Eds. 34–48.

Sato, T. 2008. A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems 31*, 161–176.

Sato, T. and Kameya, Y. 2002. Statistical abduction with tabulation. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, A. Kakas and F. Sadri, Eds. Lecture Notes in Computer Science, vol. 2408. Springer, 567–587.

Schrijvers, T. and Demoen, B. 2004. The K.U. Leuven CHR system: Implementation and application. In *CHR 2004*, T. Frühwirth and M. Meister, Eds. 8–12.

Sneyers, J. 2010. Result-directed CHR execution. In *CHR 2010*, P. Van Weert and L. De Koninck, Eds. (to appear).

Sneyers, J. and Frühwirth, T. 2008. Generalized CHR machines. In *CHR 2008*, T. Schrijvers, T. Frühwirth, and F. Raiser, Eds. 143–157.

Sneyers, J., Meert, W., and Vennekens, J. 2009. CHRiSM: Chance rules induce statistical models. In *CHR 2009*, F. Raiser and J. Sneyers, Eds. 62–76.

Sneyers, J., Van Weert, P., Schrijvers, T., and De Koninck, L. 2010. As time goes by: Constraint Handling Rules—a survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming 10,* 1 (January), 1–47.

Sneyers, J., Vennekens, J., and De Schreye, D. 2006. Probabilistic-logical modeling of music. In *PADL 2006*, P. Van Hentenryck, Ed. 60–72.

Vennekens, J., Verbaeten, S., and Bruynooghe, M. 2004. Logic programs with annotated disjunctions. In *ICLP 2004*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer, 431–445.