Electronic Notes in Theoretical Computer Science 76 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume76.html 16 pages

Probabilistic Constraint Handling Rules

Thom Frühwirth

Institut für Informatik, University of Ulm, Germany

Alessandra Di Pierro

Dipartimento di Informatica, Universitá di Pisa, Italy

Herbert Wiklicky

Department of Computing, Imperial College London, UK

Abstract

Classical Constraint Handling Rules (CHR) provide a powerful tool for specifying and implementing constraint solvers and programs. The rules of CHR rewrite constraints (non-deterministically) into simpler ones until they are solved.

In this paper we introduce an extension of Constraint Handling Rules (CHR), namely Probabilistic CHRs (PCHR). These allow the probabilistic "weighting" of rules, specifying the probability of their application. In this way we are able to formalise various randomised algorithms such as for example Simulated Annealing.

The implementation is based on source-to-source transformation (STS). Using a recently developed prototype for STS for CHR, we could implement probabilistic CHR in a concise way with a few lines of code in less than one hour.

1 Introduction

Constraint Handling Rules (CHR) [7] are a committed-choice concurrent constraint logic programming language with ask and tell consisting of guarded rules that rewrite conjunctions of atomic formulas. CHR go beyond the CCP framework [24,25] in the sense that they allow for multiple atoms on the left hand side (lhs) of a rule and for propagation rules.

CHR are traditionally used to specify and implement constraint solvers and programs. The rules of CHR rewrite constraints (conjunctions of atomic formulas) into simpler ones until they are solved. Simplification rules replace constraints by simpler constraints. Propagation rules add new constraints which may cause further simplification. Over time, CHR have been found useful for implementing other classes of algorithms, especially in computational logic:

- theorem proving with constraints
- combining deduction, abduction and constraints
- combining forward and backward chaining
- bottom-up evaluation with integrity constraints
- top-down evaluation with tabulation
- parsing with executable grammars
- manipulating attributed variables
- in general, production rule systems

Our probabilistic extension of CHR [9] is modelled after the Probabilistic Concurrent Constraint Programming (PCCP) framework [4]. The motivation behind PCCP was the formalisation of *randomised algorithms* within the CCP framework [24,25]. These algorithms are characterised by a "coin flipping" device (random choice) which determines the flow of information. In the last decade randomised algorithms have found widespread application in many different areas of computer science, for example as a tool in computational geometry and number theory. The benefits of randomised algorithms are simplicity and speed. For this reason the best known algorithms for many problems are nowaday randomised ones [15], e.g. simulated annealing in combinatorial optimisation [1], genetic algorithms [11], probabilistic primality tests in particular for use in crypto-systems [21], and randomised proof procedures (e.g. for linear logic [18]).

In PCCP randomness is expressed in the form of a *probabilistic choice*, which replaces the non-deterministic committed choice of CCP and CHR and allows a program to make stochastic moves during its execution. For probabilistic CHR (PCHR), this translates to *probabilistic rule choice*. Among the rules that are applicable, the committed choice of the rule is performed randomly by taking into account the relative probability associated with each rule.

Example 1.1 The following PCHR program implements tossing a coin. We use concrete Prolog-style CHR syntax in the program examples. Syntactically, the probabilities (weights) are the argument of the `pragma` annotation that is used in normal CHR to give hints to the compiler. Here it will initiate source to source transformation.

```
toss(Coin) <=> Coin=head pragma 0.5.
toss(Coin) <=> Coin=tail pragma 0.5.
```

Each side of the coin has the same probability. This behaviour is modelled by two rules that have the same probability to apply to a query `toss(Coin)`, either resulting in `Coin=head` or `Coin=tail`.

Besides our constraint-based approach towards the integration of probabilities into a declarative setting there is a further, rich literature on probabilistic logic programs, stochastic logic programs and Bayesian logic programs that has to be mentioned in this context, for example: [23], [22], [14], and [17].

The paper is organised as follows. In Section 2 we briefly discuss the syntax and semantics of classical, non-deterministic CHRs. In Section 3 probabilistic CHRs are introduced formally and discussed via some examples. In Section 4 we describe an implementation of PCHR which is based on source-to-source transformation of CHR following [10]. Finally we conclude by discussing several further possible developments and ongoing work.

2 Syntax and Semantics of CHR

We first introduce syntax and semantics for CHR before extending it with a probabilistic construct. We assume some familiarity with (concurrent) constraint (logic) programming [16,8,19]. A *constraint* is an atomic formula in first-order logic. We distinguish between *built-in (predefined) constraints* and *CHR (user-defined) constraints*. Built-in constraints are those handled by a predefined, given constraint solver. CHR constraints are those defined by a CHR program.

2.1 Abstract Syntax

In the following, upper case letters stand for conjunctions of constraints.

Definition 2.1 A CHR program is a finite set of CHR. There are two kinds of CHR. A *simplification CHR* is of the form

$H \Leftrightarrow G \mid B$

and a *propagation CHR* is of the form

$H \Rightarrow G \mid B$

where the left hand side (lhs) H is a conjunction of CHR constraints. The guard G followed by the symbol $\mid$ is a conjunction of built-in constraints. A trivial guard of the form *true* $\mid$ may be dropped. `true` is a built-in constraint that is always satisfied. The right hand side (rhs) of the rule consists of a conjunction of built-in and CHR constraints B.

2.2 Operational Semantics

The operational semantics of CHR programs is given by a state transition system. The semantics uses interleaving for the parallel construct of conjunction. With *derivation steps (transitions, reductions)* one can proceed from one state to the next. A *derivation* is a sequence of derivation steps.

Definition 2.2 A *state (or goal)* is a conjunction of built-in and CHR constraints. An *initial state (or query)* is an arbitrary state. In a *final state (or answer)* either the built-in constraints are inconsistent or no new derivation step is possible anymore.

Definition 2.3 Let P be a CHR program for the CHR constraints and CT be a constraint theory for the built-in constraints. The transition relation $\longmapsto$ for CHR is as follows:

Simplify
$H' \wedge D \longmapsto (H = H') \wedge G \wedge B \wedge D$
if $(H \Leftrightarrow G \mid B)$ in P and $CT \models \forall (D \rightarrow \exists \bar{x}(H = H' \wedge G))$

Propagate
$H' \wedge D \longmapsto (H = H') \wedge G \wedge B \wedge H' \wedge D$
if $(H \Rightarrow G \mid B)$ in P and $CT \models \forall (D \rightarrow \exists \bar{x}(H = H' \wedge G))$

When we use a rule from the program, we will rename its variables using new symbols, and these variables form the sequence $\bar{x}$. A rule with lhs H and guard G is *applicable* to CHR constraints H' in the context of constraints D, when the condition holds that $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$. Any of the applicable rules can be applied, but it is a committed choice, it cannot be undone.

If a simplification rule $(H \Leftrightarrow G \mid B)$ is applied to the CHR constraints H', the **Simplify** transition removes H' from the state, adds the rhs B to the state and also adds the equation $H = H'$ and the guard G. If a propagation rule $(H \Rightarrow G \mid B)$ is applied to H', the **Propagate** transition adds B, $H = H'$ and G, but does not remove H'. Trivial non-termination is avoided by applying a propagation rule at most once to the same constraints [2].

We now discuss in more detail the *rule applicability condition* $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$. The equation $(H = H')$ is a notational shorthand for equating the arguments of the CHR constraints that occur in H and H'. Operationally, the rule applicability condition can be checked as follows: Given the built-in constraints of D, try to solve the built-in constraints $(H = H' \wedge G)$ without further constraining (touching) any variable in H' and D. This means that we first check that H' matches H and then check the guard G under this matching.

The operational semantics of CHR is concretised in the following way: States are split into two parts - one for the built-in constraints and one for the CHR constraints. Built-in constraints are handled immediately by the built-in constraint solver. The conjunction of CHR constraints is implemented as a FIFO queue. The left-most (first) constraint must be involved (match one lhs atom) when a rule is applied. We call this constraint the *currently active constraint*. The other constraints that match the remaining rule lhs atoms may be taken from anywhere in the queue. If the rule is applied, the active constraint may be removed depending on the rule type, the built-in constraints of the rhs of the rule are added to the built-in constraints in the state and the new CHR constraints from the rhs of the rule are added to the queue. If no rule was applicable to the currently active constraint, it is moved to the end of the queue, and the next constraint becomes active. If all constraints of the queue have been passed without new rule application or if the built-in constraints became inconsistent, the computation stops. The final result (answer) is the contents of the queue together with the built-in constraints.

3 Probabilistic CHR

Probabilistic CHR (PCHR) is characterised by a *probabilistic rule choice*: Among the rules that are applicable, the committed choice of the rule is performed randomly by taking into account the relative probability associated with each rule.

3.1 Syntax and Operational Semantics of PCHR

Syntactically, PCHR rules are the same as CHR rules but for the addition of a weighting representing the relative probability of each rule:

Definition 3.1 A *probabilistic simplification CHR* is of the form

$H \Leftrightarrow_p G \mid B$

and a *probabilistic propagation CHR* is of the form

$H \Rightarrow_p G \mid B$

where p is a nonnegative number.

The probability associated with each alternative rule expresses how likely it is that, by repeating the same computation sufficiently often, the computation will continue by actually performing that rule choice. This can be seen as restricting the original non-determinism in the choice of the rule by specifying the frequency of choices.

The operational meaning of the probabilistic rule choice construct is as follows: Given the current constraint, find all the rules that are applicable. Each rule is associated with a probability. We have to *normalise* the probability distribution by considering only the applicable rules. This means that we have to re-define the probability distribution so the sum of these probabilities is one. Finally, one of the applicable rules is chosen according to the normalised probability distribution.

As a consequence, in the definition of the transition system, each transition (resulting from a rule application) will have a probability associated to it.

Definition 3.2 The transition relation $\longmapsto_{\tilde{p}}$ for PCHR is indexed by the normalised probability $\tilde{p}$ and is defined as follows:

Simplify
$H' \wedge D \longmapsto_{\tilde{p}_i} (H = H') \wedge G \wedge B \wedge D$
if $(H \Leftrightarrow_{p_i} G \mid B)$ in P and $CT \models \forall(D \rightarrow \exists\bar{x}(H = H' \wedge G))$

Propagate
$H' \wedge D \longmapsto_{\tilde{p_i}} (H = H') \wedge G \wedge B \wedge H' \wedge D$
if $(H \Rightarrow_{p_i} G \mid B)$ in P and $CT \models \forall(D \rightarrow \exists\bar{x}(H = H' \wedge G))$

where

$$\tilde{p_i} = \begin{cases} \frac{p_i}{\sum_{r_j} p_j} & \text{if } \sum_{r_j} p_j > 0 \\ \frac{1}{n} & \text{otherwise} \end{cases}$$

where the sum $\sum_{r_j} p_j$ is over the probabilities of all rules r_j which are applicable to the current constraint in the current state and the number of applicable rules is n.

This definition specifies the probabilities associated to a single rewrite step. If we look at a whole sequence of rewrites we have to combine these probabilities: The *probability of a derivation* is the product of the probabilities associated with each of its derivation steps. We will use the symbol $\longmapsto_p^*$ to indicate a derivation with probability p. Finally, we may end up with the same result along different derivations, i.e. different sequences of rewrites may end up with the same final state: In this case we have to sum the probabilities associated to each of these derivations leading to the same result.

Consider for example the following PCHR program:

```
c(X) <=>1: X>=0 | a(X).
c(X) <=>2: X=<0 | b(X).
```

The query constraint `c(X)` will be replaced by `a(X)` if `X` is greater than zero, by `b(X)` if `X` is less than zero. In those two cases, only one rule is applicable and its normalised application probability is therefore always one. If `X` is zero, both rules are applicable, and their normalised probabilities are $\frac{1}{3}$ and $\frac{2}{3}$, respectively. That means that in the long run, the second rule will be applied two times as often as the first rule.

3.2 Examples

In order to give an overview of the type of programs and algorithms we can easily specify using PCHR we present in the following a number of examples. These examples will also be used to illustrate a number of interesting features of PCHRs such as *probabilistic termination* which was introduced in [6] (cf. Example 3.4 and Example 3.5), and *probabilistic confluence* which will be introduced in Section 3.3. We recall here the definition of probabilistic termination.

Definition 3.3 A program is probabilistically terminating if the probability of an infinite path is zero.

We use concrete Prolog-style syntax in the examples. The following two examples are taken from PCCP [6] and have been adapted to PCHR.

Example 3.4 [Randomised Counting] Consider the following PCHR program to compute natural numbers:

```
nat(X) <=>0.5: X=0.
nat(X) <=>0.5: X=s(Y), nat(Y).
```

In a non-probabilistic implementation, a fixed rule order among the applicable rules is likely to be used, and then the result to the query `nat(X)` is either always `X=0` or the infinite computation resulting from the infinite application of the second rule.

On the other hand, the probabilistic PCHR program will compute all natural numbers, each with a certain likelihood that decreases as the numbers get larger. For example, `X=0` has probability 0.5, `X=s(0)` has probability 0.25, etc. More precisely, the probability of generating the number $s^n(0)$ is $1/2^{n+1}$.

Note that although this program does not terminate in CHR, it is probabilistically terminating in PCHR as the probability of a derivation with infinite length is zero.

Example 3.5 [Gambler's Ruin] Consider the following PCHR program which implements a so called "Random Walk in one Dimension" illustrating what is also known as "Gambler's Ruin" [12]:

```
walk(X,Y) <=>1: X\=Y | walk(X+1,Y).
walk(X,Y) <=>1: X\=Y | walk(X,Y+1).
walk(X,Y) <=>1: X=Y  | true.
```

Let `X` be the number of won games (or number of pounds won) and let `Y` be the number of lost games (or number of pounds lost). Then we can interpret `walk(1,0)` as meaning that the game starts with a one pound stake and is over when all money is lost.

Elementary results from probability theory show that the game will terminate with a ruined gambler with probability 1, despite the fact that there exists the possibility of (infinitely many) infinite derivations, i.e. enormously rich gamblers.

Although there are these infinite computations (corresponding to infinite random walks), the sum of the probabilities associated to all finite derivations (i.e. random walks which terminate in `X=Y`) is one [12,13]. Thus, the probability of (all) infinite derivations must be zero. As a consequence, this program, which classically does not terminate, does terminate in a probabilistic sense: If one continues playing, almost certainly he will ultimately loose everything.

In the following example we make use of the probability zero in order to express absolute rule preference and negation of a guard (if-then-else).

Example 3.6 The following PCHR program is an implementation of `merge/3`, i.e. merging two lists into one list while the elements of the input lists arrive. Thus the order of elements in the final list can differ from computation to computation.

```
merge([],L2,L3) <=>1: L2 = L3.
merge(L1,[],L3) <=>1: L1 = L3.
merge([X|L1],L2,L3) <=>0: L3 = [X|L], merge(L1,L2,L).
merge(L1,[Y|L2],L3) <=>0: L3 = [Y|L], merge(L1,L2,L).
```

The effect of the probabilities associated with the rules is as follows: If an empty input list is involved in the query, one of the first two rules will always be chosen, even though one of the recursive two rules may apply as well. A query `merge([a],[b],L3)` may either result in `L3=[a,b]` or `L3=[b,a]`. Since in that case, the first two rules do not apply and both recursive rules have the same probability as a consequence, both outcomes are equally likely. In that sense the PCHR implementation of `merge` is efficient and fair.

The next example shows the use of parametrised probabilities.

Example 3.7 [Simulated Annealing] Simulated Annealing (SA) is one of the most general and most popular randomised optimisation algorithms. It was inspired by the physical process of annealing in thermodynamics [20]: If a slow cooling is applied to a liquid, it freezes naturally to a state of minimum energy. The SA algorithm applies annealing to the minimisation of a cost function for solving problems in the area of combinatorial optimisation.

The SA algorithm tries to find a global optimum by iteratively progressing towards better solutions while avoiding to get trapped in local optima.

The algorithm proceeds by random walks from one solution to another one, i.e. from the current solution a new solution is computed randomly. Each solution is associated with a cost, and we are looking for the best solution, one with the least cost. To avoid being trapped in a local optimum, sometimes the worse of two subsequent solutions is chosen. The likelihood to do so depends on a control parameters called the temperature. With each iteration, the temperature decreases and thus makes the choice of the worse solution more and more unlikely. The actual probability to choose a worse solution was taken from thermodynamics. It is exponential in the cost difference of the two solution divided by the temperature multiplied with a constant.

The following PCHR program scheme implements the generic SA algorithm:

```
% solution(Temperature, Solution)

solution(T,S) <=>1:
                stop_criterion(T,S) |
                good_solution(S).

solution(T,S) <=>0:
                cool_down(T,T1),
                gen_next_sol(S,S1),
                anneal((T,S),(T1,S1)).
anneal((T,S),(T1,S1)) <=>1:
                solution(T1,S1).

anneal((T,S),(T1,S1)) <=>
              C=cost(S), C1=Cost(S1),
              e^((C1-C)/(k*T))-1: C1>C |
                solution(T1,S).
```

3.3 Confluence of PCHR programs

Confluence is an important property of (non-probabilistic) CHR programs [2]. In a confluent program, the result of a computation is always the same no matter which of the applicable rules is actually applied.

We recall the basic definitions as given in [2].

Definition 3.8 Two states S_1 and S_2 of a CHR program are *joinable* if there exist states T_1 and T_2 such that $S_1 \longmapsto^* T_1$ and $S_2 \longmapsto^* T_2$ and T_1 and T_2 are variants of each other, i.e. they can be obtained from each other by a variable renaming.

Definition 3.9 A CHR program is *confluent* if for all states S, S_1, S_2 the following holds: If $S \longmapsto^* S_1$ and $S \longmapsto^* S_2$ then S_1 and S_2 are joinable.

Given a PCHR program its CHR *support* (or CHR *version*) is given by the CHR program obtained by removing the probability information from the rules. For example the CHR support of the PCHR program

```
c(X) <=>1: X>=0 | a(X).
c(X) <=>2: X=<0 | b(X).
```

is given by:

```
c(X) <=> X>=0 | a(X).
c(X) <=> X=<0 | b(X).
```

The notion of confluence generalises in the obvious way to PCHR programs: In a confluent PCHR program we always reach the same result, possibly through different paths and with different probabilities.

Definition 3.10 Two states S_1 and S_2 of a PCHR program are *joinable* if there exist states T_1 and T_2 such that $S_1 \longmapsto^*_{p_1} T_1$ and $S_2 \longmapsto^*_{p_2} T_2$ and T_1 and T_2 are variants of each other, i.e. they can be obtained from each other by a variable renaming.

Definition 3.11 A PCHR program is *confluent* if for all states S, S_1, S_2 the following holds: If $S \longmapsto^*_{p_1} S_1$ and $S \longmapsto^*_{p_2} S_2$ then S_1 and S_2 are joinable.

For example, the above PCHR program is not confluent, since `X=0, c(X)` may lead to either `X=0, a(X)` (with probability 1/3) or `X=0, b(X)` (with probability 2/3).

It is easy to see that any PCHR program with a confluent CHR support is itself confluent. The converse does not hold in general, as the following simple PCHR program demonstrates:

```
c(X) <=>1: X>=0 | a(X).
c(X) <=>0: X>=0 | b(X).
```

This program (as a PCHR program) is confluent: both rules have the same guard, but since the second has a zero probability associated only the first rule will *always* be executed. Its CHR support however is not confluent: without probabilities both rules are possible rewrites and we might therefore end up with different results.

If we consider the results of all possible derivations of a CHR program — i.e. if we look at *fair executions* where all possible rewrites are eventually executed — then this corresponds to considering strictly positive probabilities for all rules in a corresponding PCHR program. In other words, if for the CHR support we have $S \longmapsto^* S_i$ then there exists a probabilistic derivation for the original PCHR $S \longmapsto^*_{p_i} S_i$ for some $p_i > 0$ and vice versa. For PCHR programs with non-zero probabilities we therefore have:

Proposition 3.12 *If all probabilities in a PCHR program P are strictly positive then P is confluent iff its CHR support is confluent.*

This means that the introduction of probabilities does not worsen the situation with respect to confluence: CHR programs which are confluent are also confluent in their probabilistic version.

For PCHR programs we can define a notion of *probabilistic confluence* which is more "realistic" than the notion of confluence in the sense that it allows us to ignore those computations which although possible are almost never performed (their probability is zero). Note that such computations must be infinite; in fact, as the (finite) product of non-zero numbers is always non-zero, we can only get probability zero as the limit of an infinite product. As a consequence, non-terminating programs which are classically non-confluent might result confluent according to the new notion.

Definition 3.13 A PCHR program is *probabilistically confluent* if for all states S, S_1, S_2 the following holds: If $S \longmapsto^*_{p_1} S_1$ and $S \longmapsto^*_{p_2} S_2$ then S_1 and S_2 are probabilistically joinable.

Two states S_1 and S_2 of a PCHR program are *probabilistically joinable* if there exist states T_1 and T_2 such that $S_1 \longmapsto^*_1 T_1$ and $S_2 \longmapsto^*_1 T_2$ and T_1 and T_2 are variants of each other.

That means we require that from an initial state S all derivations will meet again at the same (or equivalent under variance) state with probability one. Note that this does not exclude the existence of derivations which do not reach that unique (up to variance) state, provided that their probability is zero, that is they are infinite.

It is easy to see that any confluent PCHR program is also probabilistically confluent. If a PCHR program is confluent then all derivations from an initial state S will meet at some unique (up to variance) state T. In particular, confluence requires that there are no (infinite or finite) derivations which do not reach T. That implies that indipendently of the probability of each of the derivations which lead to T they must all sum up to one. However, the opposite is not true in general as the program in Example 3.5 implementing a one-dimensional *random walk* shows: It is probabilistically confluent (it always terminates in the state where `X=Y`) but not confluent (from the state `X=Y=0` we can reach `X=`∞`, Y=0` and `X=0, Y=`∞ which are not joinable).

4 Implementation

We implement PCHR by source-to-source program transformation (STS) in CHR [10]. In STS, users will write STS programs to manipulate other programs during their compilation. The key idea of STS for CHR is that CHR rules will be translated into relational normal form by introducing special CHR constraints for the components of a rule, which are head, guard, body and compiler pragmas. The STS program is a special purpose constraint solver that acts on this representation. When a fixpoint is reached, the relational form is translated back into CHR rules and normal compilation continues.

The result of this approach are strikingly simple STS programs. They are concise, compact and thus easy to inspect and analyse. Indeed, the complete STS program to implement probabilistic CHR consists of a few rules that easily fit one page. The STS system for CHR was implemented by Christian Holzbaur from the University of Vienna while visiting Thom Frühwirth at the Ludwig-Maximilians-University Munich.

Before we look at the STS, we show by means of an example, how the object program is represented and transformed. The example shows that PCHR can be used to generate an n bit random number. More examples of PCHR can be found in [9].

Example 4.1 [n Bit Random Number] The random number is represented as a list of `N` bits that are generated recursively and randomly one by one.

```
r1 @ rand(N,L) <=>    N=:=0 | L=[].
r2 @ rand(N,L) <=>0.5: N>0 | L=[0|L1], rand(N-1,L1).
r3 @ rand(N,L) <=>0.5: N>0 | L=[1|L1], rand(N-1,L1).
```

As long as there are bits to generate, the next bit will either get value 0 or 1, both with same probability. When the remaining list length `N` is zero, a non-probabilistic simplification rule closes the list.

The three rules above will be represented as the following conjunction of constraints to which the STS program will be applied:

```
constraint(rand/2),

head(r1,rand(N,L),id1,remove), guard(r1,N=:=0),
        body(r1,L=[]),
head(r2,rand(N,L),id2,remove), guard(r2,N>0),
        body(r2,(L=[0|L1],rand(N-1,L1))), pragma(r2,0.5),
head(r3,rand(N,L),id3,remove), guard(r3,N>0),
        body(r3,(L=[1|L1],rand(N-1,L1))), pragma(r2,0.5).
```

For each CHR constraint symbol in the object program, there is a corresponding STS constraint `constraint`. Each of the remaining STS constraints `head`, `guard`, `body` and `pragma` starts with an identifier for the rule they come from. The second argument is the respective component of the rule. For the constraint `head`, the third argument is an identifier for the constraint matching the rule head, and the last argument indicates if the constraint is to be kept or removed. This information is necessary, because any type of CHR rule is represented in the same normalised, relational way.

Now we consider the STS program for PCHR which will be applied to the above example code in relational form. It simply states how the components of the rules should be translated in case the rule is probabilistic. The two rules below basically define a standard transformation that makes the *conflict set* of the object rules explicit. The conflict set is the set of all rules that are applicable at a particular derivation step. While in normal CHR, any rule can be chosen and it is a committed choice, in probabilistic CHR we have to collect the unnormalised probabilities from all candidates in the conflict set and then randomly choose one rule according to their probabilities (weights).

```
pragma(R,N), head(R,H,I,remove),
             body(R,G) <=> number(N) |
  pragma(R,N), head(R,H,I,keep),
               body(R,(remove_constraint(I),G)).

pragma(R,N), body(R,G) <=> number(N) |
             body(R,cand(N,G)).
```

The first transformation rule maps all probabilistic rules into propagation rules that explicitly remove the head constraint(s) in the body of the rule using the standard CHR built-in `remove_constraint`. (The same effect could also be achieved using an auxiliary variable and without this standard CHR built-in, but it would be less efficient.) The second transformation rule wraps the body of a probabilistic rule with the run-time CHR constraint `cand`, whose first argument is the probability measure (weight) from the pragma. Note that the transformation rules are applied in textual order.

Last but not least there is a final, third rule that adds a last object rule for each defined CHR constraint:

```
constraint(C) ==>
  head(R1,C,I,keep), guard(R1,true), body(R1,collect(0,_)).
```

The resulting propagation rule is added at the end of the object program and just calls the CHR constraint `collect(0,_)` which triggers the probability normalisation and evaluation of the candidate set of applicable probabilistic rule bodies.

For our example of random n-bit numbers, the application of the STS rules and the final translation back into rule syntax results in the following code:

```
r1 @ rand(A,B)#C <=> A=:=0 | B=[].
r2 @ rand(A,B)#C ==> A>0 |
  cand(1,(remove_constraint(C),B=[0|D],rand(A-1,D))).
r3 @ rand(A,B)#C ==> A>0 |
  cand(1,(remove_constraint(C),B=[1|D],rand(A-1,D))).
r4 @ rand(A,B)#C ==> collect(0,D).
```

The `#C` added to the rule heads is CHR syntax for accessing the identifier of the constraint that matched the head. Note that the first rule is left untranslated since it was not probabilistic.

The probability normalisation and evaluation of the candidate set is achieved by the following rules that are defined in the STS program for PCHR and that are added to the transformed object program:

```
collect(M,R), cand(N,G) <=> cand(R,M,M+N,G), collect(M+N,R).
collect(M,R) <=> random(0,M,R).

cand(R,M,M1,G) <=> R < M | true.
cand(R,M,M1,G) <=> R >= M1 | true.
cand(R,M,M1,G) <=> M =< R, R < M1 | call(G).
```

The constraint `collect(M,R)` takes a candidate rule body `cand(N,G)` and replaces it by `cand(R,M,M+N,G)` before continuing with `collect(M+N,R)`. The effect of this rule is that each candidate constraint is extended by the common variable `R` and by the interval `M` to `M+N`, where `N` is its unnormalised probability measure (weight).

Instead of explicitly normalising the probabilities (weights), `collect` adds them up and finally calls `random(0,M,R)` to produce a random number in the interval from `0` to `M`. Note that this random number will be bound to the variable `R`.

The conjunction of extended candidate rule bodies act like a concurrent collection of agents. As soon as they receive the random number through the variable (channel) `R`, they can proceed. If the value of `R` is outside of their range of probabilities `M` to `M1`, the candidate agent simply goes away. Otherwise, it is the randomly chosen candidate and it will call its original rule body `G` (that first removes its head constraint `rand`).

In this way, from the set of applicable rules, one of the rules is randomly applied. The probability distribution follows the weights of the rules.

5 Conclusions

In this paper we presented Probabilistic Constraint Handling Rules (PCHR) which allow for an explicite control of the likelihood that certain rewrite rules are applied. The resulting extension of traditional (non-deterministic) CHRs is straightforward. It nevertheless does exhibit interesting new aspects which improve the expressivenes and the capabilities of the original language. For example, we can express fairness directly at the syntactic level by means of an appropriate probability distribution on the rules, and we can analyse average properties.

We implemented PCHR in CHR using source-to-source transformation (STS). The complete STS program to implement probabilistic CHR consists of a few rules that easily fit one page.

In the future, we would like to apply PCHRs to the search procedures of constraint solver written in CHR. Simulated Annealing algorithms are promising candidates for essentially probabilistic constraint solving and/or optimisation algorithms.

Another research direction — closely related to the application of PCHR to constraint solving problems — is the study of the relation between “chaotic iteration” in the context of classical CHR [3] and “ergodicity” in a probabilistic setting [5]: these two concepts seem to exhibit a striking similarity, and we think that a more detailed analysis of their relationship would lead to interesting results in the semantics and reasoning about (P)CHR.

Finally, the introduction of probabilities into the CHR framework seems to be an essential step in allowing for an “average case” analysis of classical as well as probabilistic algorithms. A particular aspect in this context concerns the investigation of the average running time of algorithms and/or the notion of probabilistic termination for PCHR, similar in spirit to what has been done for PCCP [6].

References

[1] Aarts, E. and J. Korst, “Simulated Annealing and Boltzmann Machines,” John Wiley & Sons, Chicester, 1989.

[2] Abdennadher, S., *Operational semantics and confluence of constraint propagation rules*, in: *3rd Intl Conf on Principles and Practice of Constraint Programming, LNCS 1330* (1997), pp. 252–266.

[3] Apt, K. R., *From chaotic iteration to constraint propagation*, in: P. Degano, R. Gorrieri and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, Lecture Notes in Computer Science **1256** (1997), pp. 36–55.

[4] Di Pierro, A. and H. Wiklicky, *An Operational Semantics for Probabilistic Concurrent Constraint Programming*, in: Y. C. P. Iyer and D. Schmidt, editors, *Proceedings of ICCL'98 – International Conference on Computer Languages* (1998), pp. 174–183.

[5] Di Pierro, A. and H. Wiklicky, *Ergodic average in constraint programming*, in: M. Kwiatkowska, editor, *Proceedings of PROBMIV'99 – 2nd International Workshop on Probabilistic Methods in Verification*, number CSR-99-8 in Technical Report (1999), pp. 49–56.

[6] Di Pierro, A. and H. Wiklicky, *Quantitative observables and averages in Probabilistic Concurrent Constraint Programming*, in: K. Apt, T. Kakas, E. Monfroy and F. Rossi, editors, *New Trends in Constraints*, number 1865 in Lecture Notes in Computer Science (2000), pp. 212–236.

[7] Frühwirth, T., *Theory and practice of constraint handling rules*, Journal of Logic Programming **37** (1998), pp. 95–138, Special Issue on Constraint Logic Programming.

[8] Frühwirth, T. and S. Abdennadher, "Essentials of Constraint Programming," Springer, Berlin, 2002.

[9] Frühwirth, T., A. D. Pierro and H. Wiklicky, *Towards probabilistic constraint handling rules*, Third Workshop on Rule-Based Constraint Reasoning and Programming (RCoRP'01) at CP'01 and ICLP'01 (2001).

[10] Frühwirth, T., A. D. Pierro and H. Wiklicky, *An implementation of probabilistic constraint handling rules*, 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002) (2002).

[11] Goldberg, D. E., "Genetic Algorithms in Search, Optimization, and Machine Learning," Addison–Wesley, Reading, Massachusetts, 1989.

[12] Grimmett, G. R. and D. Stirzaker, "Probability and Random Processes," Clarendon Press, Oxford, 1992, second edition.

[13] Grinstead, C. M. and J. L. Snell, "Introduction to Probability," American Mathematical Society, Providence, Rhode Island, 1997, second revised edition.

[14] Gupta, V., R. Jagadeesan and P. Panangaden, *Stochastic programs as concurrent constraint programs*, in: *Proceedings of POPL'99 — 26th Symposium on Principles of Programming Languages* (1999), pp. 189–202.

[15] Hochbaum, D. S., editor, "Approximation Algorithms for NP-Hard Problems," PWS Publishing Company, Boston, Massachusetts, 1997.

[16] Jaffar, J. and M. J. Maher, *Constraint logic programming: A survey*, The Journal of Logic Programming **19 & 20** (1994), pp. 503–581.

[17] Kersting, K. and L. D. Raedt, *Bayesian logic programs*, in: *Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming*, 2000.

[18] Lincoln, P. D., J. C. Mitchell and A. Scedrov, *Stochastic interaction and Linear Logic*, in: J.-Y. Girard, Y. Lafont and L. Regnier, editors, *Advances in Linear Logic*, London Mathematical Society Lecture Note Series **222**, Cambridge University Press, Cambridge, 1995 pp. 147–166.

[19] Marriott, K. and P. J. Stuckey, "Programming with Constraints: An Introduction," MIT Press, 1998.

[20] Metropolis, N., A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller, *Equation of state calculations for fast computing machines*, Journal of Chemical Physics **21** (1953), pp. 1087–1092.

[21] Motwani, R. and P. Raghavan, "Randomized Algorithms," Cambridge University Press, Cambridge, England, 1995.

[22] Muggleton, S., *Stochastic logic programs*, in: L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming* (1995), p. 29.

[23] Ng, R. T. and V. S. Subrahmanian, *Probabilistic logic programming*, Information and Computation **101** (1992), pp. 150–201.

[24] Saraswat, V. A. and M. Rinard, *Concurrent constraint programming*, in: *Proceedings of POPL'90 – Symposium on Principles of Programming Languages* (1990), pp. 232–245.

[25] Saraswat, V. A., M. Rinard and P. Panangaden, *Semantics foundations of concurrent constraint programming*, in: *Proceedings of POPL'91 – Symposium on Principles of Programming Languages* (1991), pp. 333–353.