Towards Probabilistic Constraint Handling Rules

Thom Frühwirth,Alessandra Di Pierro, and Herbert Wiklicky

Institut für Informatik, Ludwig-Maximilians-Universität, München, Germany
Dipartimento di Informatica, Universitá di Pisa, Italy
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 extention of CHRs, 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 (constraint solving) algorithms such as for example Simulated Annealing.

1 Introduction

CHR (Constraint Handling Rules) [Frü98] are a concurrent 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 [SR90, SRP91] 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 is modelled after the Probabilistic Concurrent Constraint Programming (PCCP) framework [dW98]. The motivation behind PCCP was the formalisation of *randomised algorithms* within the CCP framework [SR90, SRP91]. 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 [Hoc97], e.g. simulated annealing in combinatorial optimisation [AK89], genetic algorithms [Gol89], probabilistic primality tests in particular for use in crypto-systems [MR95], and randomised proof procedures (e.g. for linear logic [LMS95]).

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 syntax in the program examples.*

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

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`.

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. A number of examples using PCHRs are then presented in Section 4. 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 [JM94, FA97, MS98].

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. In line with the usual CCP terminology we will also refer to a CHR constraint as to an *agent*.

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*

$$H \Leftrightarrow G \mid E, B$$

and a propagation CHR *is of the form*

$$H \Rightarrow G \mid E, B$$

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 CHR constraints B and a conjunction of built-in constraints E.*

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

Propagate
$H' \wedge D \longmapsto (H = H') \wedge G \wedge E \wedge B \wedge H' \wedge D$
if $(H \Rightarrow G \mid E, B)$ *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 $(H \Leftrightarrow G \mid E, B)$ is applied to the CHR constraints H', the **Simplify** transition removes H' from the state, adds the rhs $E \wedge B$ to the state and also adds the equation $H = H'$ and the guard G. If a propagation rule $(H \Rightarrow G \mid E, B)$ is applied to H', the **Propagate** transition adds $E \wedge B$, $H = H'$ and G, but does not remove H'. Trivial non-termination is avoided by applying a propagation rule at most once to the same constraints [Abd97].

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'. More precisely, by $(H_1 \wedge \ldots \wedge H_n) = (H'_1 \wedge \ldots \wedge H'_n)$ we mean $(H_1 = H'_1) \wedge \ldots \wedge (H_n = H'_n)$, where conjuncts can be permuted. By equating two constraints, $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$, we mean $(t_1 = s_1) \wedge \ldots \wedge (t_n = s_n)$. The symbol $=$ is to be understood as built-in constraint for syntactic equality (defined in CT).

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.

In most CHR implementations, the execution of any CHR program will result in an on-line (incremental) algorithm and any-time (approximation) algorithm. The first property means that constraints can be added during run-time, the second property means that execution can be stopped and resumed after any rule application, and the intermediate results obtained will approximate the final results.

To reflect these properties, the operational semantics above 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 probability measure:

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

$$H \Leftrightarrow_p G \mid E, B$$

and a probabilistic propagation CHR *is of the form*

$$H \Rightarrow_p G \mid E, B$$

where p is a nonnegative real number.

The probability associated with each rule alternative 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
$H' \wedge D \longmapsto_{\tilde{p}} (H = H') \wedge G \wedge E \wedge B \wedge D$
if $(H \Leftrightarrow_p G \mid E, B)$ *in* P *and* $CT \models \forall(D \rightarrow \exists \bar{x}(H = H' \wedge G))$

Propagate
$H' \wedge D \longmapsto_{\tilde{p}} (H = H') \wedge G \wedge E \wedge B \wedge H' \wedge D$
if $(H \Rightarrow_p G \mid E, B)$ *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 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. 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.

For example, in the 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 second rule.

4 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 also illustrate a number of interesting features of PCHRs like *probabilistic termination* (cf Example 4.2) which are fundamentally different from notions and concepts in the classical case. We use concrete Prolog-style syntax in the examples.

The first example shows that PCHR can be used to generate an n bit random numbers.

Example 4.1 (n Bit Random Number) *The random number is represented as a list of bits that are generated recursively and randomly one by one.*

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

The next two examples are taken from PCCP [DW00] and have been adopted to PCHR.

Example 4.2 (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 an 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*. 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 4.3 (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":*

```
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 [GS92, GS97]. 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 measure zero; in this way we can express absolute rule preference and negation of a guard (if-then-else).

Example 4.4 *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 allowing for parametrised probability measures.

Example 4.5 (Primes) *We implement the sieve of Eratosthenes to compute primes in a way reminiscent of the chemical abstract machine [BJPD88]: The constraint* `candidates(N)` *generates candidates for prime numbers,* `prime(M)`*, where* `M` *is between* `1` *and* `N`*. The candidates react with each other such that each number absorbs multiples of itself. In the end, only prime numbers remain.*

```
noprime  @ candidates(1) <=> true.
generate @ candidates(N) <=> N>1 | prime(N), candidates(N-1).

sieve @ prime(I), prime(J) <=>1/J: I mod J =:= 0 | prime(J).
```

the first two rules generate candidate primes from `N` *to* `2`*. The first rule says that the number* `1` *is not a good candidate for a prime,* `candidates(1)` *is thus rewritten into* `true`*. The* `generate` *rule generates a candidate* `prime(N)` *and proceeds with recursively with the next smaller number, provided the guard* `N>1` *is satisfied.*

The third rule named `sieve` *actually performs the work of determining which numbers are not prime: If there is* `prime(I)` *and some other* `prime(J)` *such that* `I mod J =:= 0` *holds, i.e.* `I` *is a multiple of* `J`*, then remove* `prime(I)` *but keep* `prime(J)`*.*

The probability of applying a rule to a certain pair of prime candidates depends on the value of `J`*. The larger* `J`*, the less likely it is. Thus the computation prefers rule applications where it is more likely that a multiple of* `J` *will be found, because* `J` *is small. In particular, if* `prime(I)` *is the active constraint, it is likely that a small* `J` *will be chosen. If* `prime(J)` *is the active constraint, all choices of an* `I` *are equally likely, i.e. any* `I` *is chosen with same probability.*

Example 4.6 (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 [MRR+53]: 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(T,S,C) <=>1: stop_criterion(T,S,C) |
                best_solution(S,C).
solution(T,S,C) ==>0:
                cool_down(T,T1),
                generate(S,C,S1,C1),
                anneal((T,S,C),(T1,S1,C1)).

anneal(TSC,TSC1) <=>1:
                 solution(TSC1).
anneal((T,S,C),(T1,S1,C1)) <=>e^((C1-C)/(k*T))-1: C1>C |
                 solution(T1,S,C).
```

5 Conclusions

In this paper we presented a first step towards Probabilistic Constraint Handling Rules. These allow for an explicite control of the likeliness that certain rewrite rules are applied. The resulting extention of traditional (non-deterministic) CHRs is straight forward, nevertheless does exhibit interesting new aspects, e.g. allows the analysis and discussion of average properties.

Our immediate plan is to implement probabilistic CHRs in order to gain practical experience in the specification of various algorithms. In particular, 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 to study in depth the relation between

"chaotic iteration" in the context of classical CHR [Apt97] and "ergodicity" in a probabilistic setting [dW99]: these two concepts seem to exhibit a striking similarity, and we think that a more detailed analysis of their relationship would be desirable.

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. Along the lines for computing derivation lengths for CHR [Frü01], we want to compute *average* derivation lengths for PCHR, similar in spirit to what has been done for PCCP [DW00].

References

[Abd97] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *3rd Intl Conf on Principles and Practice of Constraint Programming, LNCS 1330*, pages 252–266, Berlin, Heidelberg, 1997. Springer.

[AK89] Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines*. John Wiley & Sons, Chicester, 1989.

[Apt97] Krzysztof R. Apt. From chaotic iteration to constraint propagation. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 36–55, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[BJPD88] Coutant A. Banatre J.-P. and Le Metayer D. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.

[dW98] Alessandra di Pierro and Herbert Wiklicky. An Operational Semantics for Probabilistic Concurrent Constraint Programming. In Y. Choo P. Iyer and D. Schmidt, editors, *Proceedings of ICCL'98 – International Conference on Computer Languages*, pages 174–183, Chicago, 1998. IEEE Computer Society and ACM SIGPLAN.

[dW99] Alessandra di Pierro and Herbert Wiklicky. Ergodic average in constraint programming. In Marta Kwiatkowska, editor, *Proceedings of PROBMIV'99 – 2nd International Workshop on Probabilistic Methods in Verification*, number CSR-99-8 in Technical Report, pages 49–56, Eindhoven, 1999. University of Birmingham.

[DW00] Alessandra Di Pierro and Herbert Wiklicky. Quantitative observables and averages in Probabilistic Concurrent Constraint Programming. In K.R. Apt, T. Kakas, E. Monfroy, and F. Rossi, editors, *New*

Trends in Constraints, number 1865 in Lecture Notes in Computer Science, pages 212–236. Springer Verlag, 2000.

[FA97] T. Frühwirth and S. Abdennadher. *Constraint-Programming.* Springer, Berlin, 1997.

[Frü98] T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.

[Frü01] T. Frühwirth. On the number of rule applications in constraint programs. *Electronic Notes on Theoretical Computer Science (ENTCS)*, 48, June 2001.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison–Wesley, Reading, Massachusetts, 1989.

[GS92] Geoffrey R. Grimmett and D.R. Stirzaker. *Probability and Random Processes.* Clarendon Press, Oxford, second edition, 1992.

[GS97] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability.* American Mathematical Society, Providence, Rhode Island, second revised edition, 1997.

[Hoc97] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, Boston, Massachusetts, 1997.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–581, 1994.

[LMS95] Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Stochastic interaction and Linear Logic. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Note Series*, pages 147–166. Cambridge University Press, Cambridge, 1995.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, Cambridge, England, 1995.

[MRR+53] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations for fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[MS98] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[SR90] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of POPL'90 – Symposium on Principles of Programming Languages*, pages 232–245. ACM, 1990.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantics foundations of concurrent constraint programming. In *Proceedings of POPL'91 – Symposium on Principles of Programming Languages*, pages 333–353. ACM, 1991.