Randomised Algorithms and Probabilistic Constraint Programming

Alessandra Di Pierro[1] and Herbert Wiklicky[2]

[1] Dipartimento di Informatica, Universitá di Pisa, Italy
[2] Department of Computing, Imperial College, London, UK

Abstract. We propose a declarative implementation of randomised algorithms, which exploits the Constraint Logic Programming (CLP) paradigm. For the high-level formalisation of probabilistic programs expressing such algorithms we actually refer to a generalisation of CLP, namely the Probabilistic Concurrent Constraint Programming (PCCP) language, previously introduced in [3, 5]. This language provides a construct for probabilistic choice which allows us to express randomness in a program. The design of PCCP does not require any additional structure on the underlying constraint system (e.g. fuzzy or belief systems) and therefore also allows a straight forward implementation.
We demonstrate the use of this language for implementing randomised algorithms. In particular, we give an extensive treatment of two popular (generic) randomised algorithms, namely Simulated Annealing and Randomised Rounding, and we discuss some instantiations of these algorithms for solving two well-known optimisation problems, namely the travelling salesman (TSP) and the maximum satisfaction (MAX SAT) problem.

1 Introduction

Algorithms where the flow of information is determined by an element of random choice (a "coin flipping" device), aka randomised algorithms [14], have known in the last decade a tremendous growth of interest, especially in the field of complexity theory. This is because they are particularly appropriate for the approximation of NP-complete problems [11].

Up to now such algorithms have been implemented mostly by means of probabilistic programs written in imperative languages. These programs have been analysed largely only by ad-hoc methods. However, real problems, and especially problems related to probabilistic concepts, are usually quite complicated and intuition is very likely to fail. Therefore, a formal deductive system would be much more handy.

In this paper, we propose the use of a declarative language, which generalises Constraint Logic Programming (CLP), for the formalisation and implementation of randomised algorithms. The main advantage in doing so is that we can rely on a sound and mathematically rigorous semantical framework, instead of on intuition, for tasks related to reasoning about probabilistic programs (analysis, synthesis, transformation and verification).

The foundations of such a semantical framework have been devised in previous papers [3,4], where we introduced a probabilistic language based on concurrent constraint programming, namely Probabilistic Concurrent Constraint Programming (PCCP), and provide it with an operational and a denotational semantics.

In PCCP randomness is expressed by means of a construct for probabilistic choice, which replaces (or refines) nondeterminism, and allows the program to make stochastic moves during its execution. Apart from this PCCP is identical to concurrent constraint programming. In particular, the underlying constraint system is the standard (cylindric) constraint system used in [17].

In this work we concentrate on the implementation aspects of PCCP. We demonstrate the capabilities of our system in implementing randomised algorithms, by presenting two examples of generic randomised algorithms commonly used for combinatorial optimisation, namely Simulated Annealing [1], and Randomised Rounding. Additionally, we discuss applications of such algorithms for solving specific NP-complete problems, like the Travelling Salesman Problem (TSP), and the Maximum Satisfiability (MAX SAT) problem.

These results show that PCCP is not only of theoretical interest, but can also be used for "real" applications. We do not claim the examples we present are efficient implementations from the complexity-theoretic point of view – this is not our objective. However, our system offers all the advantages stemming from declarativity, like readability, flexibility, clarity of semantics, etc.

The paper is organised as follows: In Section 2 we recall the syntax and the semantics of the PCCP language. We then present two PCCP examples of randomised algorithms in Sections 3 and 4, namely Simulated Annealing and Randomised Rounding. In Section 3 we discuss the application of Simulated Annealing for solving TSP and MAX SAT, and in Section 4 we present a solution of MAX SAT by using Randomised Rounding. Finally, in Section 5 we sum up the main results and discuss some interesting direction for future work.

2 The Language PCCP

The implementation of randomised algorithms we present in this work is based on a language previously introduced in [3] for expressing randomness in concurrent constraint programming (CCP), and called Probabilistic Concurrent Constraint Programming (PCCP). In this section we briefly recall the syntax and the semantics of this language.

In PCCP randomness is expressed in the form of a *probabilistic choice* construct directly within the semantics of the language. This is different from other approaches where randomness is introduced either in an ad-hoc way by referring to some "external" function or procedure call (like `random()` in imperative languages such as C or FORTRAN), or by means of random variables [12,10]. Moreover, the fact that randomisation is expressed at the algorithmic level rather than at the level of data allows us to avoid considering any “fuzzification” of the underlying constraint system: we still deal with *facts* and not with *believes* (cf. [2] and references therein). This is important for keeping separated the logic (constraint solving) and the control parts of the language, and allows us to use existing standard constraint solvers.

2.1 Syntax and Intuitive Semantics of PCCP

In PCCP randomness is expressed in the form of a *probabilistic choice*, which replaces the CCP nondeterministic choice and allows a program to make stochastic moves during its execution. We also replace the implicit non-deterministic scheduling in the interleaving semantics of the parallel construct by a probabilistic scheduler. This allows us to implement a kind of *prioritised parallelism*.

$$A ::= \mathbf{stop} \;\Big|\; \mathbf{tell}(c) \;\Big|\; \square_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow p_i : A_i \;\Big|\; \|_{i=i}^{n}\, q_i : A_i \;\Big|\; \exists_x A \;\Big|\; p(x)$$

Table 1. The syntax for PCCP.

A PCCP program P is then an object of the form $D.A$, where D is a set of procedure declarations of the form $p(x) : -A$ and A is an agent. The syntax of PCCP agents is given in Table 1. This is exactly the standard CCP syntax [16, 17], but for the probabilistic choice operator and the prioritised paralell construct. Successful termination is expressed by the agent **stop**; the agent **tell**, the hiding operator $\exists_x$ and the procedure call $p(x)$ are the usual ones. The operator $\|$ expresses the parallel composition of two agents. We discuss only the probabilistic choice construct, and refer to [6] for a full description of the operational semantics of PCCP.

The intended meaning of this construct is as follows: First, check whether constraints c_i are entailed by the store. All those agents A_i such that c_i are entailed are called *enabled*. At this point we have to ensure that all probabilities of the enabled agents are known. This means that if one of the p_i's is a variable, then a definite value for it must have been calculated before. After that we have to *normalise* the probability distribution by considering only the enabled agents. This can be done by replacing p_i for enabled agents by a normalised transition probability:

$$\tilde{p_i} = \frac{p_i}{\sum_{\vdash c_j} p_j},$$

where the sum is over all enabled agents. This guarantees that $\sum \tilde{p_i} = 1$. Finally, one of the enabled agents is chosen according to this new probability distribution $\tilde{p_i}$.

3 An Application: Simulated Annealing

Simulated Annealing (SA) is one of the most general and popular randomised optimisation algorithms. It was inspired by the physical process of annealing [13] and introduced for solving problems in the area of combinatorial optimisation [15]. We present one version of the SA algorithm in PCCP (Table 2,) which follows the general schema of [1].

$$
\begin{array}{l}
\mathit{simulated\_anneal}(\mathit{sol})\ :-\\
\quad \mathit{initial}(\mathit{control})\\
\ \| \ \mathit{first}(\mathit{instance})\\
\ \| \ \mathit{simulate}(\mathit{control}, \mathit{instance}, \mathit{sol})\\
\\
\mathit{simulate}(\mathit{control}, \mathit{instance}, \mathit{sol})\ :-\\
\quad \exists_{\mathit{control}'}\ (\mathit{cool}(\mathit{control}, \mathit{control}')\\
\ \| \ \exists_{\mathit{instance}'}\ (\mathit{generate}(\mathit{instance}, \mathit{instance}')\\
\ \| \ \neg\ \mathit{stopcriterion} \mid 1 \rightarrow \mathit{anneal}(\mathit{control}', \mathit{instance}, \mathit{instance}', \mathit{sol})))\\
\\
\mathit{anneal}(\mathit{control}, \mathit{instance}, \mathit{instance}', \mathit{sol})\ :-\\
\quad \mathit{cost}(\mathit{instance}') \leq \mathit{cost}(\mathit{instance}) \mid 1\\
\qquad\quad \rightarrow \mathit{simulate}(\mathit{control}, \mathit{instance}')\\
\ [] \ \mathit{cost}(\mathit{instance}') > \mathit{cost}(\mathit{instance}) \mid e^{\frac{\mathit{cost}(\mathit{instance}) - \mathit{cost}(\mathit{instance}')}{\mathit{temprature}}}\\
\qquad\quad \rightarrow \mathit{simulate}(\mathit{control}, \mathit{instance}', \mathit{sol})\\
\ [] \ \mathit{cost}(\mathit{instance}') > \mathit{cost}(\mathit{instance}) \mid 1 - e^{\frac{\mathit{cost}(\mathit{instance}) - \mathit{cost}(\mathit{instance}')}{\mathit{temprature}}}\\
\qquad\quad \rightarrow \mathit{simulate}(\mathit{control}, \mathit{instance}, \mathit{sol})
\end{array}
$$

Table 2. The Generic Simulated Annealing

3.1 Description of the Algorithm

The simulated annealing algorithm tries to find an optimum by iteratively progressing towards better solutions, while avoiding to get trapped in local optima. For the sake of presentation we consider minimisation problems.

The SA search proceeds from one possible solution to another by first creating a new (neighbouring) instance in the search space and then deciding randomly whether to take the newly created instance as starting point of further improvements. The basic idea of SA is to avoid falling into local minima by sometimes choosing the worse of two contesting solutions (thus being locally sub-optimal). The likelihood to take a locally sub-optimal solution depends on a control parameter which is often referred to as temperature. At an (initial) high temperature it is more likely to choose a new instance even if the situation becomes worse, but as the temperature falls, the probability of choosing the worse of two solutions tends to zero.

The algorithm is a generic one in the sense that it may be used on a variety of problems provided that one can supply instantiations of basically three parameters: a *cooling scheme*, a domain specific *data structure* (with a method to generate new instances), and a *cost function*. A cooling scheme may contain an initial and a stopping temperature, a method for decreasing the temperature and also a method for calculating the number of steps to be made at each temperature. For the data structure one has to provide a generate function to create new possible solution from a given instance (normally by changing the morphology of a data structure). Finally, a cost function – representing the objective function to be optimised – calculates the cost of any given instance.

A PCCP program for the SA algorithm is given in Table 2. The process ($simulated\_anneal/1$) is implemented via two mutually recursive predicates.

In $simulate/3$ the next candidate solution and next temperature are computed. It takes three arguments: The first one is the control parameter (temperature), the second is the current solution, the third one is the channel through which the (final) solution will be transmitted outwards.

On the other hand, $anneal/4$, has four arguments and is essentially the place where the choice is made between the old and the new instance, according to the cost function and the current temperature. Its arguments are (again): the control parameter (temperature), the old instance, the new instance, and finally the variable which will eventually be instantiated to the best solution found.

The particular cooling scheme used is represented by the predicates $initial/1$ and $cool/2$ which are concerned with the initial value and the changing of the temperature (control parameter). Additionally, the constraint *stop_criterion* defines condition for termination (based on a problem specific notion of optimality).

The rest of the predicates are concerned with the concrete structure of a particular problem to be solved by simulated annealing: The predicates $first/1$ and $generate/2$ deal with the initial instance and the consecutive generation of new possible solutions.

Cooling Schemes The SA algorithm depends critically on the way the control parameter, the temperature, is changed. Some analysis of this problem can be found for example in [1]. We were investigating two simple cooling schemes which are shown in Table 3.

The simplest solution is to start at a relatively hight temperature and decrement it for each new instance by a small value $c_{i+1} = \alpha c_i$. The initial temperature should make it possible to consider a great number of possible solutions (for a sufficiently long time), i.e. initially the probability of choosing a locally suboptimal solutions should be high. On the other hand α, the factor by which the temperature is dropped, should allow a small enough cooling down in order not to get trapped too early in a local optimum. Typically α takes values between 0.8 and 0.95.

$$initial(control) \quad :- \mathbf{tell}(control = 100)$$

$$cool(control, control') :- \mathbf{tell}(control' = 0.9 * control)$$

$$initial(control) \quad :- \mathbf{tell}(control = [100, 100])$$

$$\begin{array}{ll} cool([len, temp], control') :- & len \neq 0 \mid 1 \rightarrow \\ & \quad \mathbf{tell}(control' = [len - 1, temp]) \\ \Box & len = 0 \mid 1 \rightarrow \\ & \quad \mathbf{tell}(control' = [100, 0.9 * temp]) \end{array}$$

Table 3. Two Cooling Schemes

A variation of this scheme slows down the cooling by keeping the same temperature for a number of steps, e.g. 100. A cooling scheme implementing this strategy (with a fixed number of steps at each level) is shown in Table 3. There exist more complicated schemes where one can vary the amount of steps according to some measure of success on the current and previous temperatures so far.

3.2 An Instance: The Travelling Salesman Problem (TSP)

In this section we present an instantiation of the SA algorithm for solving the Travelling Salesman Problem (TSP) following [1].

The main objective is to evaluate our system by implementing SA, not in developing a new approach towards solving the TSP. The reason to chose the TSP is because it is a well known and researched problem in combinatorial optimisation. We first give a brief description of the TSP. Then we present the predicates implementing the scheme. Finally, we discuss the behaviour of the tested examples.

The TSP is to find the shortest cyclic path which connects all nodes of a weighted graph. The name is due to its relevance to the real life problem of travelling salesman/person when he/she wishes to visit a number of cities (nodes) minimising the distance travelled (weights on arcs).

More formally, we have N cities $\{c_1, c_2, \ldots, c_N\}$ and the distances between them d(c_i, c_j). Here we will only consider the symmetric case where $d(c_i, c_j) = d(c_j, c_i)$. We represent a (cyclic) tour by a permutation π of $\{c_1, c_2, \ldots, c_N\}$. The function to be optimised is then given by,

$$length(\pi) = \sum_{i=1}^{n} d_{\pi(i),\pi(i+1)}.$$

The generic SA algorithm from Table 2 is specialised for solving the particular TSP problem by providing the following parameters:

data structures to represent the topology of our problem, i.e. the *distance matrix* between two cities, by means of a predicate/function $dist/2$; and to represent a *route* or *permutation* π by a list of cities,

cost function represented by $cost/1$, which computes $length(\pi)$, i.e. the sum of all distances between consecutive cities in the particular route,

generation of new routes from another one, by reversal of the path between two arbitrary cities (nodes), in order to define the predicate $generate/2$ (and $first/1$),

cooling scheme given by the standard one in Table 3, to define $cool/2$ (and $initial/1$), and finally a

stop criterion to define when the SA algorithm should return an approximative solution (*stop_criterion*).

Although very limited work was spent on developing and testing the system, it was a very simple task to implement all these predicates and to undertake some test runs on "real-world" data. These included a TSP problem containing twenty five cities in Great Britain, which is solved to within 2% of optimality in seconds, and a hundred and one cities map (EIL101 from Christofides/Eilon) with results suboptimal to about 20% from best solution, within minutes (< 10).

3.3 Another Instance: Maximum Satisfiability (MAX SAT)

We now use SA for solving another problem, namely the maximum satisfiability problem (MAX SAT). This serves as an illustration to the flexibility and expressive power of the system. We do not claim this to be the best way to solve the particular problem or even a good way. Our primary interest is to recast already existing algorithms to the presented system.

We are looking at simple (propositional) formulas in conjunctive normal form (CNF), i.e. formulas of the form $\bigwedge \bigvee x_i$ where x_i, called *literal*, can be either positive or negative. The subformulas $\bigvee x_i$ are referred to as *clauses*. An *assignment* is a function which assigns to each literal a truth value in $\{true, false\}$ or $\{T, F\}$. A small problem of this type is given by $C = x_1 \vee x_2 \vee x_3 \wedge \bar{x_1} \wedge \bar{x_2} \wedge \bar{x_3} \wedge x_2 \vee \bar{x_3}$

Given a CNF formula the basic objective is to find an assignment (or to decide if such an assignment exists) such that the complete formula is satisfied, i.e. evaluates to *true*. This leads to some of the best known class of problems in complexity theory, namely SAT, 2SAT, 3SAT, etc. Our objective is a little bit more general: It is not always possible to satisfy all clauses of a CNF formula simultaneously, therefore we are looking for a truth assignment which maximise the number of satisfied clauses in a given set of clauses. This is referred to as the maximum satisfiability problem (MAX SAT).

We implement a simple algorithm to solve this optimisation problem using the generic SA algorithm in Table 2 to solve the MAX SAT problem. To do so, we have only to define the problem and domain specific elements of the algorithm,

data structures to represent the target *CNF formula* as an array of positive and negative literals, and an *assignment* by a list of truth values,

cost function to be minimised as the number of unsatisfiable clauses to define *cost*/1,

generation of new assignments from another one, by negating the truth value for an element in the list representing the previous assignment, in order to define the predicate *generate*/2 (and *first*/1),

cooling scheme similar to the one used for the TSP problem and depicted in Table 3, to define *cool*/2 (and *initial*/1), and finally a

stop criterion to define when the SA algorithm should return an approximative solution (*stop_criterion*).

4 An Application: Randomised Rounding

An alternative way which has been proposed for (approximately) solving the MAX SAT and other maximisation problems, like MAX CUT, etc., in discrete structures uses linear programming relaxation [8, 9].

4.1 Description of the Algorithm

The basic idea of this approach is to recast the original (integer programming) problem to a linear programming problem. Then standard techniques form linear programming are used to solve this "relaxed problem". Finally, the solutions to the linear formulation are used to guide the assignment of integer or truth values to the variables of the original integer problem. The generic PCCP program implementing this approach is shown in Table 4.

$$
\begin{array}{rl}
semidef(intprog, intsol) :- & int2lp(intprog, linprog) \\
\| & lpsolve(linprog, linsol) \\
\| & lp2int(linsol, intsol)
\end{array}
$$

Table 4. Semidefinite Programming Scheme

The solution obtained by linear optimisation (which we think of as a list of reals) will be re-translated into an integer or (as in the case of MAX SAT) into a boolean solution by so-called *randomised rounding* (RR). The mechanism is depicted in Table 5.

4.2 An Instance: Maximum Satisfiability (MAX SAT)

We use randomised rounding to solve or – to be precise – to approximate the MAX SAT problem. Consider a set of clauses $C = \{C_1, \ldots, C_m\}$ in the variables/literals $x_1, \ldots, x_n$. For any clause $C_i \in C$ let I_i^+ be the set of positive literals and I_i^- that of the negative literals.

$$
\begin{array}{lll}
rround(p,b) & :- & true \mid p \rightarrow \mathbf{tell}(b = T) \\
 & \Box & true \mid 1-p \rightarrow \mathbf{tell}(b = F) \\
 & & \\
lp2int(sol, asgn) & :- & sol = [\,|\,] \mid 1 \rightarrow \mathbf{tell}(asgn = [\,|\,]) \\
 & \Box & sol = [s|ol] \mid 1 \rightarrow rround(s,a) \\
 & & \quad \| \ lp2int(ol, sgn) \ \| \ \mathbf{tell}(asgn = [a|sgn])
\end{array}
$$

Table 5. Randomised Rounding

Then we can reformulate the MAX SAT problem as an integer programming problem in the new variables $y_1, \ldots, y_n$ which correspond to the boolean variables $x_1, \ldots, x_n$:

$$\max \sum_{C_i \in C} z_i$$

subject to:

$$\sum_{i \in I_i^+} y_i + \sum_{i \in I_i^-} (1 - y_i) \geq z_j, \forall C_j \in C$$

$$y_i \in \{0,1\}, z_i \in \{0,1\}$$

The next step is to look at this discrete problem as a continuous problem. To do so we relax the integer conditions on y_i and z_i,

$$0 \leq y_i \leq 1, 0 \leq z_i \leq 1$$

allowing real values as solutions, which turns our integer program into a linear program. In our case, we can make use of the linear equations solver (through the CLP(R) library module of SICStus) which handles the linear programming part.

The only task left is the translation of the solution to the linear problem into the final assignment of truth values. The translation phase uses the randomised rounding procedure of Table 5 ($rround/2$) and is straightforward. The assignment of a truth value to a boolean variable x_i depends on the value its corresponding linear variable y_i. The linear variables can only assume values in the real interval $[0, 1]$. We interpret the value of y_i simply as the probability that the corresponding boolean variable x_i assumes the truth value 'T/true' (or 'F/false').

5 Conclusions and Future Work

We have developed a system for implementing randomised algorithms, which is based on a declarative language, namely the PCCP language introduced in [3, 4].

The examples presented show that the implementation of randomised algorithms in our system is simple and straightforward.

The treatment of randomisation is different from the usual approaches based on some random number generators. This has been possible because of the characteristics of the language, which allows us to introduce randomness directly within the semantics as a probabilistic choice construct, instead of by referring to some external pseudo-function or -predicate for generating random numbers.

A prototypical implementation of PCCP has been implemented before in cooperation with Nicos Angelopoulos by means of a meta-interpreter for this language written in SICStus prolog. Since the design of PCCP does not require any particular assumption on the underlying constraint system, we can rely on existing constraint solvers (for CLP). This leads to a very elegant and economical formulation for the meta-interpreter.

Although these results are interesting in themselves, they represent the basis for the realisation of a more ambitious project, namely the design and implementation of randomised proof procedures and reasoning techniques. In particular, we plan to exploit simulated annealing and other randomised optimisation algorithms for proof-search purposes, by relying on the relationship between optimisation problems and verification problems [7, 11].

Furthermore, there is ongoing theoretical research into various aspects of PCCP: its (full) denotational semantics, towards average cost analysis, probabilistic abstract interpretation, etc.

References

1. Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines. A Stochastic Approach to Combinatorial Optimization and Neural Computing.* Discrete Mathematics and Optimization. John Wiley & Sons, 1989.
2. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.
3. Alessandra Di Pierro and Herbert Wiklicky. On probabilistic CCP. In *Proceedings of APPIA-GULP-PRODE'97 Joint Conference on Declarative Programming*, June 1997.
4. Alessandra Di Pierro and Herbert Wiklicky. A Banach space based semantics for Probabilistic Concurrent Constraint Programming. In *CATS'98 – Computing: the 4th Australian Theory Symposium*, Lecture Notes in Computer Science, page (to appear), Heidelberg – New York, 1998. Springer Verlag.
5. 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*, IEEE Computer Society and ACM SIGPLAN, pages 174–183, Chicago, 1998. IEEE Comp. Soc. Press.
6. Alessandra di Pierro and Herbert Wiklicky. Quantitative observables and averages in Probabilistic Concurrent Constraint Programming. In *ERCIM Workshop on Constraints*, Paphos, Cyprus, 2000. Springer Verlag.
7. Michel X. Goemans. Advanced algorithms. Technical Report MIT/LCS/RSS-27, Massachusetts Institute of Technology, `http://theory.lcs.mit.edu/~goemans/`, December 1994.

8. Michel X. Goemans and David P. Williamson. New 3/4-approximation algorithms for the maximum satisfiability problem. *SIAM Journal on Discrete Mathematics*, 7, 1994.
9. Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, November 1995.
10. Vineet Gupta, Radha Jagadeesan, and Vijay A. Saraswat. Probabilistic concurrent constraint programming. In *Proceedings of CONCUR 97*. Springer Verlag, 1997.
11. Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.
12. Dexter Kozen. Semantics for probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.
13. 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.
14. Rajeev Motwani and Prabhakar Raghavan. *Randomised Algorithms*. Cambridge University Press, 1995.
15. C. D. Gelatt S. Kirkpatrick and M. P. Vecchi. Optimization by simulated annealling. *Science*, 220:671–681, 1983.
16. Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *In Proceedings of POPL*, pages 232–245. ACM, 1990.
17. Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantics foundations of concurrent constraint programming. In *Proceedings of POPL*, pages 333–353. ACM, 1991.