Maintaining Correct Variable Domains in Distributed and Dynamic Constraint Processing

Georg Ringwelski

Cork Constraint Computation Center, University College Cork, Ireland
g.ringwelski@4c.ucc.ie

Abstract. The possibility to integrate new knowledge dynamically in continious applications is a major requirement for many applications today. Recently such applications are more and more required to cooperate online with similar entities in an open, distributed system of software agents. However, the known methods for dynamic constraint processing require too much communication to be used in the distributed setting. We define basic distributed algorithms for the maintainance of consistency in such continious and open computing systems. These dynamically adapt variable domains to changing knowledge in a distributed system of Reactive Agents without any central control or synchronisation.

1 Introduction

Constraint Satisfaction Problems (CSP) are widely used in commercial and academic applications to model arbitrary complex combinatorial problems. The mapping of NP-complete problems to a CSP allows for the use of reusable powerful constraint processing components to solve the thus modeled problems. Such generic constraint processing systems, called constraint solvers (e.g. CHIP, ILOG Solver, ECLiPSe or SICStus Prolog), make many results of 20 years of research in Constraint Programming (CP) available to the constraint programmer who can declaratively describe a problem instead of implementing algorithms. The general approach is to use search *and* inference to solve the problems. The search, which is necessary due to the NP-completeness of the problem, is speeded up significantly by an inference technique called constraint propagation. Before and during the execution of the search algorithms very large parts of the search tree can often be pruned due to the knowledge given in the posted constraints.

In most state-of-the-art applications, models in the finite-domain constraint system are used [10]. In this domain specialized (n-ary) global constraints were developed, that we think make up a lot of the success of Constraint Programming. Their powerful filtering algorithms coupled with (usually complete) search

This work was funded by the Embark Initiative of the Irish Research Council for Science, Engineering and Technology with fellowship PD/2002/21. The Cork Constraint Computation Center is supported from Science Foundation Ireland under Grant 00/PI.1/C075.

methods, such as Forward-Checking or Conflict-Directed Backjumping can be considered to be currently the most appropriate way to solve many CSPs.

However, the use of most finite domain constraint solvers is due to their execution models currently very restricted:

- not all of them support the incremental addition of new constraints,
- very few of them support incremental constraint retraction,
- only one of them (Mozart/Oz) allows for the distribution of problems among different processes.

Recently, however, the classical application domains of Constraint Programming occur more and more in software settings that require some of these unsupported properties. For example, modern user interfaces are expected to allow the user to undo any of her decisions, which is best implemented by constraint retraction; resource allocation, planning or scheduling applications are often to be distributed among different computers; etc.

Thus we intend to design the behavior of systems that use our new approach similar to established finite domain constraint solvers, but additionally support distribution and adapting to changing problems. In this paper we define a Multi-Agent-System to compute correct variable domains dynamically. This is, the domains are adapted online to all new knowledge in a distributed manner.

2 Background

2.1 CP Preliminaries

In CP, every variable is associated to a finite subset (its domain) of an initial set D by $dom : V \to \mathcal{P}(D)$. Every constraint c is posted over a tuple of variables and defines a set of allowed tuples of values for these variables. These constraint semantics are usually given by relations in the addressed constraint system: $sem(c) \subseteq dom^{+}(var(c))$, where $dom^{+} : V \times V ... \times V \to \mathcal{P}(D) \times \mathcal{P}(D) ... \times \mathcal{P}(D)$ is the point-wise extension of dom to tuples of variables. In efficient finite domain constraint processing systems, the allowed tuples $sem(c)$ are not stored explicitly (i.e. as tuples), but are calculated from the variable domains by so-called propagators [15, 14, 5] that are given with every constraint. Propagators imply monotonic and inflationary functions red in $\underset{v \in V}{\times} \mathcal{P}(D)$ that reduce the domains of variables $V_{out} \subseteq V$ if applicable, such that $\forall v \in V_{out} : red \downarrow_v (dom(v)) \subseteq dom(v)$ and $\forall v \in V \setminus V_{out} : red \downarrow_v (dom(v)) = dom(v)$ hold. The domain reduction, made by a propagator excludes values from the domains of all of its constraint's variables, that are (wrt. $sem(c)$) proven not to be part of any solution of the global problem.

Definition 1 (CSP,DistrCSP). *A* **C***onstraint* **S***atisfaction* **P***roblem* (C, V, dom) *in a constraint domain* D *is given by a finite set* C *of constraints, a finite set* V *of variables and a total mapping* $dom : V \to \mathcal{P}(D)$*. Such that* $\forall c \in C : var(c) \subseteq V \wedge sem(c) \subseteq dom^{+}(var(c))$*. A* **Distr***ibuted* **CSP** (C, V, dom, A, loc) *is a CSP with an additional set of agents* A *and a total mapping* $loc : V \cup C \to A$*.*

These definitions address closed world settings, where all constraints, variables and their initial domains are known in advance and one, all or optimal (wrt. some objective function) solutions are wanted as the result of constraint processing. Real-world problems usually require a more dynamic setting, where solutions are to be adapted to new knowledge. This new knowledge can arise in our framework in the following form:

1. The addition of new constraints and thus their integration into an already solved or partially processed CSP.
2. The exclusion of former beliefs and thus the retraction of constraints.

The addition or removal of variables from a CSP is not tackled in this paper. As in Logic Programming we assume a finite, but sufficiently large set of variables V to be given in advance and that for all constraints $var(c) \subseteq V$ holds.

Definition 2 (DynCSP,DDCSP). *A* **Dyn***amic* **CSP** *is a sequence* $a_o, a_1, \ldots.$ *of CSPs, where* $a_0 = (V, \emptyset, dom_0)$ *with* $\forall v \in V : dom_0(v) = D$ *and* $a_n = (V, C_n, dom_n)$*, with* $\forall v \in V : dom_n(v) \subseteq D$ *and* $C_n = C_{n-1} \cup \{c\}$ *or* $C_n = C_{n-1} \setminus \{c\}$*. A* **D***ynamic* **D***istributed* **CSP** *is a similar sequence of Distributed CSPs* $a_i = (V, \emptyset, dom_i, A_i, loc_i)$ *where* $\forall v \in V : loc_n(v) = loc_{n-1}(v)$ *and* $\forall c \in C_{n-1} \cap C_n : loc_n(c) = loc_{n-1}(c)$ *hold. The set of Agents* A_n *is always updated to the co-domain of* loc_n

2.2 Download!

We have implemented a rudimentary demo tool for our new approach. The system is provided and described on `http://www.4c.ucc.ie/~gringwelski/racp`.

2.3 Related Work

Distributed Constraint Satisfaction is intensively discussed in the CSP community in recent years. Most of these approaches [20, 12, 16] deal with search algorithms that do not make use of constraint propagation. Consequently they do not address the maintainance of correct domains in a dynamic setting. Boi Faltings [3] recently proposed the Open CSP approach. The term "Open" concerns the variable domains, which get new values during the process of constraint processing. Thus, OCSP has nothing in common with our approach to CSP, where variable domains are to be reduced and not extended in order to find solutions.

In monolithic settings Dynamic Constraint Satisfaction is investigated since the early 90's. Algorithms for the enforcement of arc-consistency [2] in dynamic but only binary and monolithic CSPs are known and applied in some academic work.

The incremental addition of Propagators of arbitrary constraints is supported by many systems today. They usually use the concept of storing adjacent constraints in every variable and triggering them, if they may infer new domain reductions [4]. This method of constraint propagation can be directly applied in distributed systems [13, 14], because of the confluence of this process even in Chaotic Iteration [1].

There is some research on the incremental constraint retraction in monolithic settings. The known algorithms compute a global state that would be (equivalent to) the current, if the retracted constraint had never been posted. There are three main approaches to such algorithms:

1. Keep track of the domain reductions on the value level and add all values after retraction, that can now be part of a solution again [5].
2. Keep track of all constraints that are entailed by the retracted one and recompute the sub-problem given by them after retraction [6, 19, 9, 14]
3. Store some selected current knowledge, restore a former state add the stored knowledge and recompute from there [17, 18].

The first two approaches make intensively use of structural information on the constraint network. To do this half way efficiently, most of them use central constraint stores or history trails. Such global structural information is hard to be gathered in distributed settings, where communication is expensive. Thus we chose approach number three for our current work and use a retraction algorithm, which stores all information that has to be computed by non-polynomial algorithms and re-does everything, that can be done polynomially. This is: store the result of search and re-compute all the propagation after constraint retraction. The application of this method in distributed settings yields some difficulties that we will discuss later.

3 Two Layers of Agents

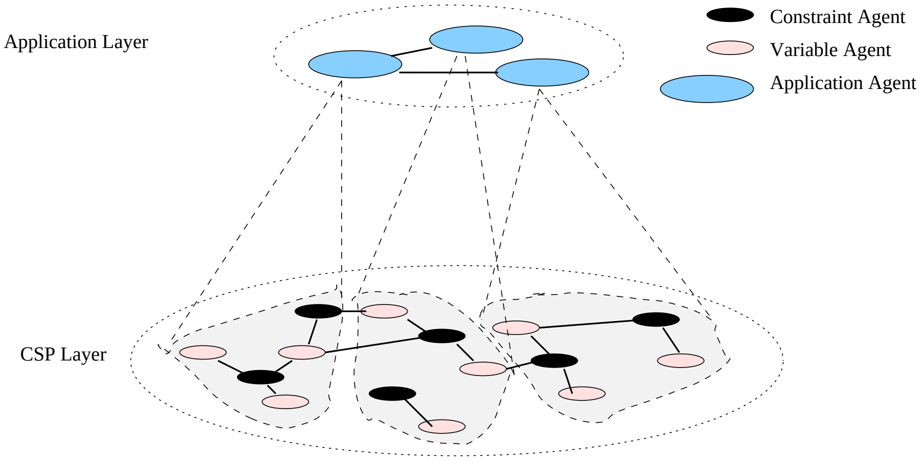
The proposed approach uses two layers of agents in distributed constraint-based applications (see Figure).

1. The application layer, which contains cognitive[1] agents and interfaces
2. The CSP layer, which contains two sorts of reactive agents: Variables and constraints

Application agents (AA) make up the application layer and define the DD-CSP that is to be solved on the CSP layer. They will each typically be connected to a user interface, where the CSP can be modified and results are presented. Every AA can hold proxies[2] to variable agents (VA) and constraint agents (CA). Furthermore they can hold proxies to other AA, such that they can communicate VA and CA proxies. The VA and CA are located on the CSP layer and do not know any of the application layer. The VA and CA solve the CSP using their "collective intelligence".

[1] We distinguish cognitive from reactive agents [7]. The first are rather complex entities, that behave "intelligently" themselves. while the latter's behavior is charcterized by simple $situation - action$ rules.

[2] A proxy provides a surrogate for an object to control access to it [8].



The approach is not restricted to binary constraints and it is adequate to process DDCSPs. The overall system is an open computing system, where new AA, CA and VA can be added anytime. The addition of a CA requires the existence of all its constraint's variables (Def. 2), such that they must always be created first. Creating CA can be seen as constraint addition and will yield propagation if applicable. The removal of a CA is interpreted as constraint retraction and variables can be removed, if there are no constraints over them. Thus the architecture allows for dynamic changes of the underlying CSP. The agents use no common memory and communicate exclusively by asynchronous messages. Thus the approach is naturally distributed and does not require any further consideration of the locations of agents. For the rest of the paper we assume that every sent message ocasionally arrives at its receiver.

4 Concurrent Constraint Propagation

Constraint Propagation can be formalized as the Chaotic Iteration of Propagators, as described in [6, 1]. One main result of the latter work is that the process of propagation will reach a stable and uniquely determined state independently of the execution order of the Propagators. This makes the approach directly applicable for concurrent propagation [13]. The only requirement for the constitution of a Chaotic Iteration process is that the propagators imply inflationary and monotonic functions.

The fixed point of the Iteration is formally evaluated by an infinite iteration process, but it is easy to prove complete propagation in terminating systems, that use the idea of triggering depending constraints [6, 14]. The soundness of such terminating propagation methods is generally regarded wrt. to the result of chaotic iteration. Thus the Propagators themselves are assumed as the semantics of constraints. The Propagators are expected to be implemented in a way, that matches the intended declarative semantics, but there is no formal link between operational and declarative semantics as it is regarded in CLP for example.

Before showing its soundness and completeness, we will describe how propagation is performed in our approach. One Propagator is implemented for every constraint. For the $X < Y$-constraint for example, this could be the following procedure:

```
1 procedure awake()
2   max(dom[X]) <- min(max(dom[X]),max(dom[Y]) - 1);
3   min(dom[Y]) <- max(min(dom[Y]),min(dom[X]) + 1);
```

Such Propagators are invoked (line 7) by a CA-behavior (lines 4-9) that reacts to **propagate(d)** messages, which are sent by VA and contain their variable's domain **d** at the time of sending the message:

```
4 receive propagate(d) from v            // v represents X or Y
5   dom[v] <- d;                         // update local knowledge
6   old_X = dom[X]; old_Y = dom[Y];
7   awake();
8   if(dom[X] reduces old_X) send(X,propagate(dom[X]));
9   if(dom[Y] reduces old_Y) send(Y,propagate(dom[Y]));
```

The knowledge on its variables' domains is stored and updated in the CA in the `dom` array (line 5). If domain reductions could be evaluated, they will be sent to their corresponding VA by **propagate** messages (lines 6,8,9).

Variables react to the receipt of a **propagate(d)** message by reading the new domain **d** from the message and intersecting it with its current domain. If this does not lead to a domain reduction (i.e. $dom(v) \subseteq$**d**), the behavior resumes waiting for new **propagate** messages. Otherwise, the VA will assign its domain to the intersection of its former and the received domain and communicate it to all its *adjacent constraints* by sending a **propagate(**$dom(v) \cap$**d)** message.

Besides the concurrently executable cyclic behavior described above, every CA integrates itself upon its construction into the current DDCSP with the following "OneShot"-behavior: Send every variable a **confirm** message containing proxies to all its variables. The proxies of the variables must be given as a parameter to the constructor of every CA. The VA react to **confirm** messages by adding its sender (i.e. a CA proxy) to the their locally stored set of *adjacent constraints* and storing the VA proxies locally as *neighbors*.

Whenever a stable state of (V, C_n, dom_n) where no messages are underway is reached in a DynCSP with this propagation, the variable domains dom_n will be the same as one gets with Chaotic Iteration over the propagators of C_n.

Lemma 1. *The Propagation constituted by this protocol is sound and complete wrt. the result of Chaotic Iteration of all constraints.*

Proof. 1. All propagators are inflationary ($\forall v \in V : red\downarrow_v (dom(v)) \subseteq dom(v)$) and monotonic ($\forall v, v' \in V : dom(v) \subseteq dom(v') \Rightarrow red\downarrow_v (dom(v) \subseteq red\downarrow_{v'} (dom(v'))$), since the variable domains are sets and the VA only save intersected domains. Thus the Chaotic Iteration of these procedures is well defined and will lead to a uniquely determined fixed point F.

2. The propagation is sound wrt. F, since no domain reductions are made except of those defined in the propagators.
3. The propagation is complete wrt. F, since
   (a) Every CA is a *adjacent constraint* of all of its variables, because it sent a **confirm** message upon its construction.
   (b) Every VA sends all its *adjacent constraints* a **propagate** message, every time its domain was reduced.
   (c) Every message arrives.
4. The propagation will terminate, since all domains contain a finite number of values and can thus only be reduced finitely often.

5 Distributed Constraint Retraction

As motivated in Section 2.3, we chose a constraint retraction algorithm for this new approach that can be described roughly by the following steps:

(1) Store result $ass : V \to D$ of search, if applicable.
(2) Remove retracted constraint c from all lists of *adjacent constraints*
(3) Enlarge $dom(v)$ to initial set $dom_0(v)$ for every v that is reachable in the constraint graph from c
  (4a) If ass was defined, create a $v = ass(v)$ constraint for every $v \in V$ (this will start propagation due to the domain reduction).
  (4b) Otherwise, send every constraint that is reachable in the constraint graph from c a **propagate** message.

Naturally, this procedure cannot be implemented directly in our framework, since it requires more than the local knowledge provided in any agent. We thus describe in the following, how the tasks (1)-(4b) of this global behavior can be implemented by local behaviors of the VA and CA.

The retraction procedure of constraints is implemented by sending **disconfirm** messages to all their variables (and nothing else). The VA behavior (lines 10-20) that reacts to this message removes the constraint from the set `AC` of *adjacent constraints* (2) and starts the relaxation of its own and the domains of all variables it stored in its *neighbors* `N`. Finally it sends all its constraints a **propagate** message in order to recompute the correct fixed point (4b):

```
10 receive disconfirm from c
11   AC <- AC \ {c};                  // remove constraint
12   if( NOT domain.relaxed) relax(); // relax if not done
13   forall ca in AC                  // trigger propagation
14      send(ca,propagate(dom))
15
16 procedure relax()
17   domain <- dom_0                  // relax domain to initial
18   domain.relaxed <- true           // set flag
19   forall( n in N)                  // notify neighbors
20      send(n,relax)
```

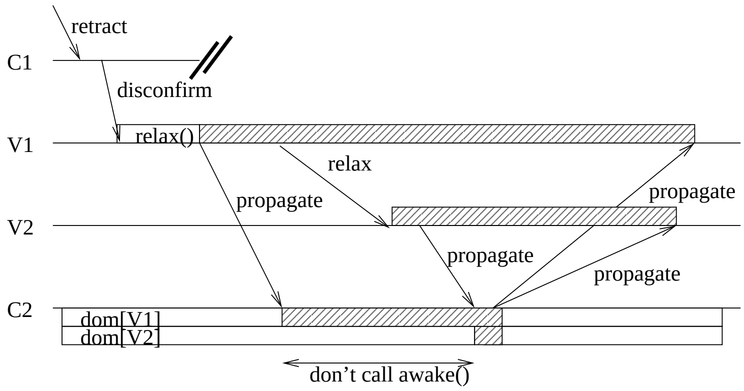


Fig. 1. C1 over variable V1 is retracted and C2 interrupts propagation, until it received a **propagate** message of *all* its variables (V1 and V2). During the hatched periods, the `relaxed`-flag is set in the respective domain.

Every VA will upon the receipt of a **relax** message call its procedure `relax()` if it did not already relax its domain. The relaxation is thus performed in every reachable variable exactly once by using a concurrent "wave of markers" as described in [11]. The used marker is the `relaxed`-flag, it will be unset with the first execution of a domain reduction after the relaxation phase.

The `relaxed`-flag is furthermore necessary, in order not to confuse old and new knowledge in the running process of propagation. Since we enlarge variable domains with the relaxation, it interferes with the required monotony of the propagation process described in the previous section. Thus the relaxation has to be separated properly from the iteration process of propagation. We perform this by interrupting the propagation locally, until relaxation is finished in the relevant variables with the use of the `relaxed`-flag. The flags are used to allow the constraints only to propagate, if either all its known domains are relaxed or none of them is. Thus propagation in every constraint does not start after a retraction before all its variables finished their domain relaxation. For example, a binary constraint over X and Y will only execute its `awake()` procedure upon a **propagate** message, if `dom[X].relaxed` = `dom[Y].relaxed`. With the first execution of a propagator, the flag is unset. With this technique, we allow for continuation of propagation in some parts of the CSP, while others are still or not yet busy with retraction. We illustrate the integration of a relaxation into the running propagation process by a time-line diagram shown in Fig. 5.

In a DynCSP, this retraction algorithm will have calculated correct variable domains dom_{n+1} when reaching a stable state of $(V, C_n \setminus \{c\}, dom_{n+1})$ from a previous state of (V, C_n, dom_n). If $c \notin C_n$, the problem did not change and $dom_{n+1} = dom_n$. Otherwise, the correct fixed point will be calculated as a consequence of the described retraction algorithm:

Lemma 2. *The retraction algorithm constituted by the described protocol will terminate and transform a correct state with the constraints C_n to a correct state with $C_n \setminus \{c\}$.*

Proof. 1. The domain of all relevant variables is relaxed to dom_0, because they will call their relax method (lines 12 nad 17). These variables V' are all that can be reached in the constraint graph from the retracted constraint c, due to the communication of *neighbors* with every **confirm** message.
2. The relaxation will terminate, because there are finitely many variables in V'.
3. All constraints C' that can reduce the domains of V' are triggered (lines 13,14). There can be no relevant constraints that are not triggered due to the storing of *adjacent constraints* upon **confirm** messages.
4. The propagators don't use outdated knowledge (which would include the semantics of c) due to the use of the `relaxed`-flag. Every propagator will not be triggered before the last of its variables has sent a **propagate(**dom_0**)** message. Thus, every propagator of c' will start calcualting with `dom[v]` $=$ dom_0 for all `v`$\in vars(c')$.
5. The propagators will evaluate the correct domains dom_{n+1} in finite time after retraction due to Lemma 1.

It can be argued, that this algorithm will not be efficient since it re-performs a lot of computations that were done before. Other algorithms are "more incremental", such as [5, 6, 14] for example. We have not yet evaluated the performance of our algorithms, but we think our approach will perform well for the following three reasons: First, we have observed in monolithic approaches, that the known retraction algorithms can only perform significantly better in some CSP settings and not in general. Second, all incremental retraction algorithms we know would require a lot of communication in a distributed setting, which would slow them down significantly. And third, we only recompute the easy part of the problem, namely what can be done by the polynomial propagation algorithm. The result of the non-polynomial search is stored and re-used by triggering the found instantiations after relaxation.

6 Conclusion and Further Work

We have introduced our new approach to dynamic and distributed constraint processing that uses one reactive agent for every constraint and variable. The main contribution of the paper is to define locally processable reactions on events performed by variables and constraints that maintain correct variable domains wrt. the theoretical fixed point of propagation of all constraints.

Currently we are integrating different search algorithms into our prototype system. We focus on adaptive and decentralized methods for search that suit into our general approach of Reactive Agents. In future research we will investigate methods to react DDCSP-specificically to events, that typically occur in open distributed systems. This can for example be agents that don't react to messages (anymore), messages that don't arrive, new agents that show up, new knowledge that somebody enters etc. The underlying Reactive Agents approach is designed for such flexible usage in distributed and dynamic environments and can be used for DDCSP to allow foe new application areas of CP.

References

[1] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1998.

[2] Christian Bessiere. Arc-consistency for non-binary dynamic csps. In *Procs. ECAI 92*, 1992.

[3] S. Macho-Gonzalez Boi Faltings. Open constraint satisfaction. In *Principles and Practice of Constraint Programming*. Springer LNCS, 2002.

[4] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 1996.

[5] R. Debruyne, G. Ferrand, N. Jussien, W. Lesaint, S. Ouis, and A. Tessier. Correctness of constraint retraction algorithms. In *Special Track on Constraint Solving and Programming of FLAIRS 2003*, 2003.

[6] F. Fages, J. Fowler, and T. Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37:1-3:185–212, Oct-Dec 1998.

[7] Jaques Ferber. *Multi-Agent-Systems, An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[9] Yan Georget, Philippe Codognet, and Francesca Rossi. Constraint retraction in CLP(FD). *Constraints*, 4(1):5–42, February 1999.

[10] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[11] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[12] Amnon Meisels and Igor Razgon. Distributed forward checking with dynamic ordering. In *Proc. CP01 workshop on collective search algorithms*, 2001.

[13] Eric Monfroy and Jean-Hughes Réty. Chaotic iteration for distributed constraint propagation. In *Proc. 14th ACM Symposium on Applied Computing, SAC'99, Artificial Intelligence and Computational Logic Track*, 1999.

[14] Georg Ringwelski. *Asynchrones Constraintlösen*. PhD thesis, Technical University Berlin, 2003. `edocs.tu-berlin.de/diss/2003/ringwelski_georg.htm`.

[15] Christian Schulte and Mats Carlson. Finite domain constraint programming systems. Tutorial at CP02 conference, Ithaca, USA, September 2002. `www.it.kth.se/schulte/talks/FD`.

[16] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *Proc. AAAI/IAAI 2000*, pages 917–922, 2000.

[17] Pascal Van Hentenryck and Thiery Le Provost. Incremental search in constraint logic programming. *New Generation Computing*, 9:257–275, 1991.

[18] Gérard Verfaillie and Thomas Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the Twelfth Conference of the American Association of Artificial Intelligence*, pages 307–312, 1994.

[19] Armin Wolf. Adaptive constraint handling with chr in java. In Toby Walsh, editor, *Proc. CP2001*, LNCS 2239, pages 256–270. Springer, 2001.

[20] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.