e-constraints: explanation-based Constraint Programming

Narendra Jussien

École des Mines de Nantes
4 rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3, France
njussien@e-constraints.net
WWW: http://njussien.e-constraints.net

Abstract. In this paper, we present our experience in using explanations within constraint programming: how to implement an explanation system, what to use explanations for, how to use explanations to develop new algorithms. More precisely, beside classical uses (for debugging and/or solving dynamic problems), we introduce a more in-depth use of explanations that leads to a new kind of constraint programming that we call explanation-based constraint programming (e-constraints). This paper summarizes and extends previous works from the same author.

1 Introduction

Classical constraint programming systems (such as Solver from Ilog, Chip from Cosytec or gnuProlog from INRIA) are helpless when there is no solution to the constraint system to be solved. In fact, only a `no solution` message is provided. Users are left alone to find out why: is it because of the problem itself (no solution exists), an incorrect modelling, a bug in the solver, etc.

In order to promote constraint programming, the constraints community needs to address this issue. For example, a set of constraints that left alone lead to the unexpected situation can be provided to the user. A promising technique for providing such information seems to be using **explanations** [11]. An explanation is a set of constraints justifying propagation events generated by the solver (value removal, bound update, contradiction).

This paper is an introduction to how to produce and use explanations within constraint programming. We emphasize that using explanations within constraint programming can be very useful: not only for debugging purposes but also for automatic constraint relaxation or dynamic problems handling. Moreover, it can be used in building new past-oriented search algorithms.

Firstly, in section 2 we define an alternative notion for explanations: contradiction explanation and eliminating explanation. Secondly, we focus on integrating the computation of explanations within a constraint solver (section 3): `choco` [17] (a constraint layer above the programming language `claire` [7]). Finally, different possible uses are presented in section 4 along with the impact on using explanation on usual practice of constraint programming (which we call *e-constraints: explanation-based constraint programming*) in section 5.

2 Explanations for Constraint Programming

We consider here CSP represented by a couple (V, C). V is a set of variables and C a set of constraints on those variables. Notice that variable domains are considered as unary constraints. Moreover, the enumeration mechanism is handled as a series of constraints additions and retractions. Those constraints are called *decision constraints*. Indeed, we chose not to limit our tools to value assignments but to allow any kind of decision constraint (*eg.* ordering constraints between tasks in scheduling, splitting constraints in numeric CSP).

Let us consider a constraints system whose current state (*i.e.* the original constraint and the set of decisions made so far) is contradictory. A **contradiction explanation** (*a.k.a.* **nogood** [20]) is a subset of the current constraints system of the problem that, left alone, leads to a contradiction (no feasible solution contains a nogood). A contradiction explanation divides into two parts: a subset of the original set of constraints ($C' \subset C$ in equation 1) and a subset of decision constraints introduced so far in the search (here $dc_1, \ldots, dc_k$).

$$\neg\,(C' \wedge dc_1 \wedge \ldots \wedge dc_k) \tag{1}$$

An operational viewpoint of contradiction explanations can be made explicit by rewriting equation 1 the following way:

$$C' \wedge \left(\bigwedge_{i \in [1..k] \setminus j} dc_i \right) \rightarrow \neg dc_j \tag{2}$$

Let us consider $dc_j : v = a$ in the previous formula. The left hand side of the implication is called an **eliminating explanation** (explanation for short) because it justifies the removal of value a from the domain $d(v)$ of variable v. It will be noted: $\texttt{expl}(v \neq a)$.

Filtering operations in CSP can be considered as a sequence of value removals which can all be explained as in equation 2. The simplest of all explanations is to merely consider the complete set of currently active constraints (*i.e.* the initial constraints of the problem and the set of all the decisions – and their associated enumeration constraint – made so far). Notice that much more useful explanations can be provided as we will see in section 3.

Explanations can be combined with each other to provide new ones. Let us suppose that $dc_1 \vee \ldots \vee dc_j$ is the set of all possible choices for a given decision (set of possible values, set of possible sequences). If a set of explanations $C'_1 \rightarrow \neg dc_1$, ..., $C'_j \rightarrow \neg dc_j$ exists, a new explanation can be derived: $\neg(C'_1 \wedge \ldots \wedge C'_j)$. Such new explanation gives more information than each of the old ones.

From the empty domain of a variable v, a *contradiction explanation* can be computed:

$$\neg \left(\bigwedge_{a \in d(v)} \texttt{expl}(v \neq a) \right) \tag{3}$$

Notice that when a contradiction explanation does not contain any decision constraint, the associated problem is proved to be over-constrained.

Several eliminating explanations generally exist for the removal of a given value. Recording all of them leads to an exponential space complexity. Another technique relies on *forgetting* (erasing) explanations that are no longer relevant[1] to the current variable assignment. By doing so, the space complexity remains polynomial. We here retain **one** explanation at a time for a value removal.

3 Computing explanations

The most interesting explanations are those which are minimal regarding inclusion. Those explanations allow highly focused information about dependency relations between constraints and variables. Unfortunately, computing such an explanation can be exponentially costly [10]. We claim that a good compromise between precision and ease of computation is to use the solver embedded knowledge in order to provide interesting explanations. Indeed, constraint solvers always know, although it is scarcely explicitly, *why* they remove values from the domain of the variables. By making that knowledge explicit, quite precise and interesting explanations can be computed.

3.1 Exploiting filtering algorithms to provide explanations

The solvers that we develop use an event-based model as in the `choco` solver [17]: during propagation, constraints are awaked (like agents or daemons) each time a variable domain is reduced (this is an event) possibly generating new events (value removals)[2]. In this model, a constraint is fully characterized by its behavior regarding the basic events such as value removal from the domain of a variable (method `awakeOnRem`) and domain bound updates (methods `awakeOnInf` and `awakeOnSup`).

[1] An explanation is said to be *relevant* if all the decision constraints in it are still valid in the current search state [2].

[2] As recalled in [17], this architecture is well suited for solvers that maintain a local consistency (arc-consistency and others) but is not so well suited for consistencies that are defined upon several constraints at the same time.

> **Example 1 (Constraint $x \geq y + c$) :**
> This is one of the basic constraints in `choco`. It is represented by the `GreaterOrEqualxyc` class. Reacting to an upper bound update for this constraint can be stated as: if the upper bound of x is modified then the upper bound of y should be lowered to the new value of the upper bound of x (taking into account the constant c). This is encoded as:
>
> ```
> [awakeOnSup(c:GreaterOrEqualxyc, idx:integer) : void
>  -> if (idx = 1) updateSup(c.v2, c.v1.sup - c.cste)]
> ```
>
> `idx` is the index of the variable of the constraint whose bound (the upper bound here) has been modified. This particular constraint only reacts to modification of the upper bound of variable x (`c.v1` in the `choco` representation of the constraint). The `updateSup` method only modifies the value of y (`c.v2` in the constraint) when the upper bound is really updated.

Explanations for events need to be computed when the events are generated *i.e.* within the propagation code of the constraints (namely the `awakeOnXXX` methods of `choco`). In order to make it as simple as possible, extra information only needs to be added to the `updateInf` or `updateSup` calls: the actual explanation. Example 2 shows how such an explanation can be computed and what the resulting code is for a basic constraint. Example 3 gives another point of view on explanations for classical binary CSP.

> **Example 2 (Modifying the solver) :**
> It is quite simple to make modifications considering example 1. Indeed, all the information is at hand in the `awakeOnSup` method. The modification of the upper bound of variable `c.v2` (y) is due to:
>
> (a) the use of the constraint itself (it will be added to the computed explanation);
> (b) the previous modification of the upper bound of variable `c.v1` (x) that we captured through the *calling* variable (`idx`).
>
> The source code is therefore modified in the following way (the additional third parameter for `updateSup` contains the explanation attached to the intended modification):
>
> ```
> [awakeOnSup(c:PalmGreaterOrEqualxyc, idx:integer) : void
>  -> if (idx = 1) updateSup(c.v2, c.v1.sup - c.cste,
>                            becauseOf(c, theSup(c.v1)))]
> ```
>
> The `becauseOf` method builds up an explanation from the events-parameters.

Our implementation of explanations provides a set of tools in order to ease the modification process[3]:

[3] Notice that those tools are not tightly related to `choco`. They would be the same for any other constraint solver.

- the `Explanation` class that captures *contradiction explanations* and *eliminating explanations*;
- the modification of the domain updating method in order to efficiently store the explanations associated to a given variable;
- tools for consulting explanations for a variable (the `becauseOf` method).

> **Example 3 (Explanations for classical binary CSP) :**
> Explanations for classical binary CSP where constraints are expressed by a list of allowed tuples of values for their variables can be easily stated.
> When applying constraint c_{xy} between variables x and y, we will want to remove value a from the domain of x if and only if all supporting values for a in the domain of y regarding constraint c_{xy} have been removed. This can be expressed this way:
>
> $$\texttt{expl}(x \neq a) = c_{xy} \bigcup_{b \texttt{ supp. } a} \texttt{expl}(y \neq b)$$

Those modifications added to each propagation method rapidly give an *explanation-friendly* constraint solver.

3.2 Explanations for global constraints

It is easy to provide explanations for basic constraints: the existing code explicitly states what the real triggering conditions are (as we saw in example 2). Unfortunately, when dealing with global constraints, it is not so easy. Indeed, triggering occurs when an accumulation of information modifies a data structure enough to reject parts of a domain of a variable[4].

When triggering the propagation rule, not all of those pieces of information are kept. Therefore computing a precise explanation may not be that easy. There is always a generic explanation: the current state of the domains of each variable of the constraints. Such an explanation is useless if used with all the constraints of the system because everything will depend on everything else!

When adding explanations to global constraints, it is highly recommended to get back to the overall algorithm in order to make it explanation-friendly. One should remember that that a precise explanation system will act as a good documentation for the global constraint!

Another point about global constraints is that they absolutely need to get data structures synchronized with the current state of the domain at every moment. As we will see in section 5, we cannot rely upon backtracking for modifying data structures when undoing some previous choice. In order to keep the synchronisation of the current state of the domain with the data structures of the

[4] For example, consider the `alldifferent` constraint [19]. The maintained data structure allows the removal of a value from the domain of a variable as soon as those two (variable and value) appear in different strongly connected components in a specific graph updated throughout the computation. What would be the explanation ? It need thinking about!

constraint, a clear separation between data structure maintenance and propagation is necessary.

3.3 Complexity issues

One of the major interests of what we present here is that both space and time complexities remain polynomial. As the explanation network *traces* the behavior of the solver, when a single value is removed (which happens at most once for each possible value) a single explanation will be generated. Thus, the space complexity for a CSP defined upon n discrete variables of maximum domain size d upon which are posted e constraints is in $O((e+n) \times n \times d)$. There are at most $n \times d$ explanations (one for each value) of maximum size e (all the constraints of the problem) $+n$ (one assignment constraint for each variable).

Experimental results presented in [13] show that computing explanations while filtering lead to a time overhead due to actual computing and storing of explanations, but this overhead remains quite constant whatever the nature of the problem (highly constrained problems *vs.* under-constrained problems). Moreover, that overhead is rewarded when we actively use those explanations for, for example, guiding the search (*see* section 5.3).

3.4 Other computation techniques

We forbid modification of the classical mechanisms of constraint programming (filtering and enumeration). We provide a *trace* of the solver in order to compute explanations. Explanations are therefore neither minimal nor exhaustive and highly dependant on the solver mechanisms and implementation. We see it as a pragmatic point of view.

There are alternatives. For example:

- finding mechanisms or enumeration ordering that are efficient for producing both solutions and explanations;
- computing explanations in a lazy way (keeping less information – as in [3]);
- *investing* in explanations and radically modifying solving mechanisms.

In addition, a promising approach is to develop non intrusive techniques in which we do not need to modify the existing propagation code. We are currently investigating the use of Aspect Oriented Programming (AOP [16]) for monitoring a constraint solver from the outside and still providing precise explanations for its behavior.

4 Exploiting explanations

The aim of this section is to introduce different usages of explanations within constraint programming.

4.1 Understanding the current situation

Explanations can be directly used to help the user (the application developer, the solver developer, etc.) to understand the current situation within the constraint solver:

- *Why does my problem have no solution ?*
  When the domain of a variable is empty (no value exists that will respect all the constraints on that variable), constraint systems usually notify users that there is no solution. Checking the explanation of this situation will help to understand why it is so by narrowing the problem to relevant constraints. In our implementation, the command `becauseOf(theDom(`v`))`, if v is the emptied variable, will give an explanation computed following equation 3.
- *Why variable x cannot take the value a ?*
  Checking the associated removal explanation will help to answer the question.
  In our implementation, `becauseOf(theHole(`x`, `a`))` will provide the required explanation (it sends back `expl(`$x \neq a$`)`).
- *What if constraint c gets back into the system ?*
  In this situation, constraint c has been retracted. Keeping track of explanations that contained constraint c before its retraction will allow a glimpse (a necessarily partial but still valid one) on the effects of the activation of c. Indeed, events attached to those explanations would become valid.
  Notice that keeping such invalid explanation can be done using what is called k-relevance in [2].

Using explanations in a debugging context is still an open area (how to *translate* explanations into user-friendly terms, or how to use them effectively in a debugging system).

However, we experienced their use in the PTIDEJ system [1]: an automated system for identifying distorted micro-architectures in object-oriented source code (the idea is to find *quasi* solutions to an over-constrained problem by identifying set of constraints that cannot be simultaneously verified).

4.2 Dynamic constraint retraction

Explanations can be very useful when dealing with dynamic problems. Incremental constraint addition to a problem is a well known issue but incremental constraint *retraction* is not so easy. Christian Bessière already used a simplified explanation system (using *justifications*[5]) to perform such a retraction [3].

[5] A *justification* for a removal consists in the actual constraint that performed the removal. From that information, an explanation can be computed by recursively taking into account removal justifications in the variables that intervene in the constraint that justifies the concerned value removal. In general, the explanation that we would have computed is a strict subset of the explanations computed using justifications.

The retraction process When an explanation-based system is used, incremental constraint retraction of a given constraint c can be achieved this way:

1. **Disconnecting** The first step is to cut c from the constraint network. c needs to be completely disconnected (no further propagation).
2. **Setting back values** The second step, is to undo the past effects (direct and indirect ones) of the constraint. This can be done by considering all the explanations containing the removed constraint: all the associated events (value removal) are no longer *valid* and those values need to be put back in their respective domains.
3. **Controlling what has been done** There are several different explanations for a given removal. Some values that has been put back in the domain of variable need to be removed if another explanation can be found.
4. **Repropagation** In order to have a consistent state, those new removals need to be propagated.

At the end of this process, the system is in a consistent state. It is exactly the state that would have been obtained if the retracted constraint (c) would not have been introduced into the system[6].

This process is like the one described in [3] but we do not need to *compute* the past effects of a constraint (exploring backwards the *justification* system) because each of our *explanations* contains all the information at once (we just need to compute the set of explanations that do contain the retracted constraint). Moreover, it is worth noticing that our set of undone events is a subset (often a strict one) of undone events of [3].

Notice that if we only want to retract constraints incrementally, it is not worth to use an explanations system. A *justification* system would be more appropriate. However, an explanation system can be used for many other purposes (problem analysis, new search algorithms, etc.).

A new event A new kind of event comes to mind when analyzing the constraint retraction process: value restoration (or bound restoration) in a domain of a variable. In order to achieve the third step (control) for the constraint retraction process, the related constraint is called in order to confirm the validity of the value restoration. This is what methods `awakeOnRestoreInf`, `awakeOnRestoreSup` and `awakeOnRestoreVal` are about.

[6] Notice that the state of the domain is the state that would have been obtained if the constraint originally was not in the system. But, the set of current explanations is only one of the several different states that are associated to the initial set of constraints without c.

Example 4 (Value restoration for $x >= y + c$) :
Let us consider a `GreaterOrEqualxyc` constraint. The `awakeOnRestoreXXX` methods can be easily written considering the associated `awakeOnXXX` methods.

```
[awakeOnRestoreInf(c: PalmGreaterOrEqualxyc, idx: integer) : void
 -> if (idx = 1) awakeOnInf(c, 2)]

[awakeOnRestoreSup(c: PalmGreaterOrEqualxyc, idx: integer) : void
 -> if (idx = 2) awakeOnSup(c, 1)]
```

Like in the example 1, `idx` is the index of the variable whose lower (*resp.* upper) bound has been restored.

A constraint, in an explanation-based solver, is characterized by its reaction to all its domain modification events: value removal (`awakeOnInf`, `awakeOnSup`, and `awakeOnRem` methods) **and** value restorations (`awakeOnRestoreInf`, `awakeOnRestoreSup`, and `awakeOnRestoreVal` methods).

4.3 Solving over-constrained problems

As we saw, we can use the same tool for both computing an explanation for contradiction and for removing a constraint incrementally from a given constraints system. Using explanation for solving over-constrained problems naturally comes to mind [12]: consider solving the constraints system as *usual* and, if necessary (*i.e.* once the problem is proved to be over-constrained), use explanations to identify the constraint(s) to be removed (using a *comparator* that will take the user's preferences into account) and remove it (or them) incrementally. This process being iterated as much necessary to obtain a suitable solution for the resulting constraint system. The use of the comparator guarantees the optimality of the solution [12].

The originality of this approach is two-fold:

- All constraints of the problem can be propagated from the beginning of the search. The use of comparison criteria can be delayed to the last moment [12]. Indeed, choice criteria (reflecting the user opinion about relaxing constraints) are used only for selecting a set of constraints among the ones appearing in the current contradiction explanation. Thus, classifying constraints is only needed when a contradiction is encountered and only between relevant constraints.
- Search is made in the space of possible relaxations in opposition to the space of possible affectations in more classical approaches ([6] or [21]).

5 Explanation-based search for constraint programming

The usage of explanations within a constraint programming language leads to new programming gimmicks and new approaches. Most of CSP solving algorithms are derived from a backtrack-based complete search. The drawbacks of this approach have been known for a long time: thrashing and inappropriate behavior. The previously developed fancy backtrackers were not really convincing to address that issue (time or space overhead, no real advantages) [4]. Nevertheless, more recently, the `mac-dbt` algorithm [13] showed that explanation-based algorithms could compete well with backtrack-based algorithms in real-world situations.

5.1 From backtrack-based to explanation-based solving

Algorithm 1 presents the `solve` method. `solve` looks for a solution in a given CSP. In this algorithm, a step of computation consists in the addition of an enumeration constraint[7] and its propagation through the constraint network.

Our proposal consists in the usage of the same overall process it which backtrack (that would occur on line 15 of algorithm 1 when a contradiction is identified) is replaced by a more specific handling.

Algorithm 1 (Solving a CSP) :

```
   [solve(pb: Problem): boolean
    -> unassignedVars := pb.vars,
      try (
       while not(empty(unassignedVars)) (
 5        let idx := nextVarToAssign(pb), // variable choice
              v   := unassignedVars[idx],
              a   := selectValToAssign(pb, v) // value choice
          in (
             try (
10              unassignedVars :delete v,
                post(pb, v == a), // instantiation
                propagate(pb)
             )
             catch (LabelingContradiction) ( // An empty domain found
15              handleContradiction(pb)  // classically: BACKTRACK
             )
         ),
         true                    // A solution exists
      )
20    catch (ProblemContradiction) (
       false                     // No solution
      )  ]
```

[7] For CSP, enumeration constraints are instantiations but other kinds of constraints can be used. For example, in numeric CSP, enumeration constraints are *splitting* constraints [14]; for scheduling problems, enumeration constraints may be precedence constraints [15].

5.2 Generic contradiction handling

Algorithm 2 introduces our general schema for handling contradictions. An identified contradiction can be explained by the system (see line 2 in algorithm 2 and recall that a contradiction is identified when the domain of a variable – the failing variable – becomes empty). In order to overcome the contradiction, at least one constraint in its explanation must be removed from the system (hence the selection in line 7 and the removal on line 12).

In order to ensure that the same context will not get explored twice, we add the negation (using the *opposite* method) of the removed constraint (line 15) into the system[8]. Obviously, that new constraint must not be a permanent one: it remains valid only if each of other constraints in the original explanation (`e` in algorithm 2) remains active. Such a constraint is called an *indirect constraint*[9].

Algorithm 2 (Contradiction handling) :

```
1  [handleContradiction(pb: Problem): void
2  -> let e: Explanation := becauseOf(theDom(getFailingVariable(pb)))
3     in (
4        if (empty(e))      // The problem has no solution
5            raiseProblemContradiction()
6         else (
7             let ct := selectConstraint(e) // select a to be removed choice
8             in (
9                if known?(ct) ( // it exists
10                       unassignedVars :add ct.v1,
11                       try (
12                            remove(ct), // actual constraint removal
13                            e :delete ct,
14                            // the negation is added (indirect constraint)
15                            post(pb, opposite(ct), e),
16                            propagate(pb)) // achieving consistency
17                       )
18                       catch LabelingContradiction (
19                          handleContradiction(pb) // recursive handling
20                         )
21                )
22                else (  // no more enumeration constraint
23                     raiseProblemContradiction()
24                )
25             )
26      )) ]
```

Notice that while removing a constraint or propagating its negation a contradiction might occur. Hence, the recursive handling provided on line 19.

[8] This behavior is the translation into explanation-based constraint programming of switching branch in a backtrack-based approach.

[9] Our implementation, the `PaLM` system [11] provides tools for indirect constraints.

Thus, an explanation-based constraint solver which is capable of incremental and efficient constraint retraction (as shown in section 4.2) can easily replace[10] a backtrack-based approach.

5.3 A versatile framework

Different algorithms are obtained according to the algorithms used for selecting a constraint or computing explanations. For example, the following variations exist:

- **Standard backtracking:** Considering that all the constraints of the system constitute an explanation for any event (this is clearly the case although it is not very precise information) and systematically choosing the most recent enumeration constraint in the contradiction explanation leads to a classical backtrack-based solver.
- **Intelligent backtracking:** Using a more precise explanation schema (such as the one presented in section 3), an intelligent backtracker can be designed (*i.e.* an algorithm that will backtrack higher in the search tree: to a really relevant choice point with no solution loss *à la* CBJ [18]). Not only the most recent constraint in the explanation needs to be removed but also all the other enumeration constraints added since that removed one.
- **Dynamic backtracking:** When using the explanation system of section 3 and only removing the most recent constraint in the contradiction explanation, we obtain an implementation of *dynamic backtracking* [8] that handles constraint propagation in each step of the resolution [13].

Each of the preceding algorithms performs a complete search. Christian Bliek even introduces conditions to verify in order to overcome the *most recent* limitation in the choice of the constraint to relax in *dynamic backtracking* [5]. Notice that if all choices are left for that selection, the obtained search may not remain complete. However, it can give a very efficient local search: this is the `path-repair` algorithm class [15].

5.4 Applications

We have been using explanation-based constraint programming. We developed and tested several search algorithms (complete as heuristic ones). Here are a few applications:

- In [9], we presented an *intelligent backtracker* for solving classical *open-shop* scheduling problems. This first use of explanations lead us to solve an open instance for the first time .
- [13] introduces the `mac-dbt` algorithm which uses explanations to interface constraint propagation and *dynamic backtracking*. Experiments with this algorithm showed that its usage was very useful when problems to be solved are *structured* (which is often the case for real-world problems).

[10] Only the completeness of the search needs to be ensured (see section 5.3).

– Finally, we developed an heuristic search based upon cooperation between constraint propagation and repair techniques: the `path-repair` algorithm [15]. Results obtained for *open-shop* problems were quite interesting: several instances were solved for the first time and many others were improved.

In general, experiments show that using explanations is really interesting as soon as a structure (quasi-independent sub-problems) appears *during* resolution. Explanations allow a quick, efficient, and automatic use of that hidden structure to guide the search.

6 Conclusion and further works

In this paper, we presented an introduction to the use of explanations within constraint programming and showed how to compute and use them. We also illustrated how programming habits are changed when a complete use of explanations is wanted and what were their main interests.

The explanation system and the new search techniques presented in this paper were implemented within the `PaLM` suite [11] in `choco` [17] (the constraint layer of the `claire` language [7]).

Our current work includes decoupling the constraints solver and the explanation system in order to provide a non-intrusive explanation system. We are also working on the usage of explanations for debugging and cooperation between solvers.

References

1. Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *16th IEEE conference on Automated Software Engineering (ASE'01)*, San Diego, USA, November 2001.
2. Roberto J. Bayardo Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI'96*, 1996.
3. Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.
4. Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problem. In *CP'96*, Cambridge, MA, 1996.
5. C. Bliek. Generalizing partial order and dynamic backtracking. In *Proceedings of AAAI*, 1998.
6. Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint hierarchies and logic programming. In Giorgio Levi and Maurizio Martelli, editors, *ICLP'89: Proceedings 6th International Conference on Logic Programming*, pages 149–164, Lisbon, Portugal, June 1989. MIT Press.
7. Yves Caseau and François Laburthe. CLAIRE: Combining objects and rules for problem solving. *Proceedings of JICSLP, workshop on multi-paradigm logic programming*, 1996.

8. Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
9. Christelle Guéret, Narendra Jussien, and Christian Prins. Using intelligent backtracking to improve branch and bound methods: an application to open-shop problems. *European Journal of Operational Research*, 127(2):344–354, 2000.
10. Ulrich Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, Seattle, WA, USA, August 2001.
11. Narendra Jussien and Vincent Barichard. The palm system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.
12. Narendra Jussien and Patrice Boizumault. Best-first search for property maintenance in reactive constraints systems. In *International Logic Programming Symposium*, pages 339–353, Port Jefferson, N.Y., USA, October 1997. MIT Press.
13. Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.
14. Narendra Jussien and Olivier Lhomme. Dynamic domain splitting for numeric csp. In *European Conference on Artificial Intelligence*, pages 224–228, Brighton, United Kingdom, August 1998.
15. Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Seventh National Conference on Artificial Intelligence – AAAI'2000*, pages 169–174, Austin, TX, USA, August 2000.
16. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th Europeen Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.
17. François Laburthe. Choco: implementing a cp kernel. In *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, Singapore, September 2000.
18. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993. (Also available as Technical Report AISL-46-91, Stratchclyde, 1991).
19. Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI 94, Twelth National Conference on Artificial Intelligence*, pages 362–367, Seattle, Washington, 1994.
20. Thomas Schiex and Gérard Verfaillie. Nogood Recording fot Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
21. Thomas Schiex and Gérard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, August 1995.